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

Alfred V. Aho
Brian W. Kernighan
Peter J. Weinberger
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

\[ \text{length} > 72 \]

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

\[ \text{NF} \% 2 == 0 \]

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

\[ \{ \text{$1 = log($1)}; \text{print} \} \]

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, \textbf{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.

September 1, 1978
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. The program grep, a unix program manual 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

```
$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 files given; 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 }
```

```
... 
```

Each line of input is matched against each of the patterns in turn. When all the patterns have been tested, the next line is fetched and tested. Readers familiar with the UNIX†

†UNIX is a Trademark of Bell Laboratories.
append the output to the file *foo*. (In each case, the output file name is the same as the input file name, with a slight change in the case of the first letter.)

```
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).

```
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.

```
BEGIN { FS = ":" }
```

sets the field separator to a colon by

2.2. Regular Expressions

If an expression is a literal string of characters enclosed in slashes, like

```
#/Aa/b#W/[w][einber]/[Kk][r]ernighan/
```

then matches it against the entire line, as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are considered "special" and are not whitespace.

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 considered "special" and are not whitespace.

```
/N, *=\N/
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, Awk uses the contents of field 2 as a file name.

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 wrapping.

```
BEGIN { FS = ":" }
```

sets the field separator to a colon by

```
END { print NR }
```

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

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.

```
blacksmithing
```

3. Actions

```
/Aa/b/[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 considered "special" and are not whitespace.

```
/N, *=\N/
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, Awk uses the contents of field 2 as a file name.

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 wrapping.

```
BEGIN { FS = ":" }
```

sets the field separator to a colon by

```
END { print NR }
```

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

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.

```
blacksmithing
```

3. Actions

```
/Aa/b/[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 considered "special" and are not whitespace.

```
/N, *=\N/
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, Awk uses the contents of field 2 as a file name.

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 wrapping.

```
BEGIN { FS = ":" }
```

sets the field separator to a colon by

```
END { print NR }
```

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

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.

```
blacksmithing
```

3. Actions

```
/Aa/b/[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 considered "special" and are not whitespace.

```
/N, *=\N/
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, Awk uses the contents of field 2 as a file name.

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 wrapping.

```
BEGIN { FS = ":" }
```

sets the field separator to a colon by

```
END { print NR }
```

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

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.

```
blacksmithing
```

3. Actions

```
/Aa/b/[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 considered "special" and are not whitespace.

```
/N, *=\N/
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, Awk uses the contents of field 2 as a file name.

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 wrapping.

```
BEGIN { FS = ":" }
```

sets the field separator to a colon by

```
END { print NR }
```

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

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.

```
blacksmithing
```

3. Actions

```
/Aa/b/[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 considered "special" and are not whitespace.

```
/N, *=\N/
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, Awk uses the contents of field 2 as a file name.
The argument may be any expression. 

Awk also provides the arithmetic functions \(\sqrt{n}\), \(\log(x)\), and \(\exp(x)\). These follow the algorithm, exponentiation, and functions also allow the possibility to use 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

\[
\text{length} < 10 \quad \text{||} \quad \text{length} > 20
\]

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

The function \(\text{substr}(s, m, n)\) produces the substring of \(s\) that begins at position \(m\) and is \(n\) characters long. If \(n\) is omitted, the substring goes to the end of \(s\). The function \(\text{index}(s, x)\) returns the position where the string \(x\) occurs in \(s\), or zero if it does not.

The function \(\text{sprint}(f, e_1, e_2, \ldots)\) produces the value of the expression \(f\) and prints lines whose length is less than 10 or greater than 20.

\[
x = \text{sprint}("\%8.2f \%10ld", \$1, \$2)
\]

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. Fields in an awk program are a mix of numeric and string values, as well as an example of a conventional numeric subscript.

\[
x = 1
\]

\(x\) is clearly a number, while in

\[
x = "\text{smith}\"
\]

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

\[
x = \text{"3"} + \text{"4"}
\]

assigns 7 to \(x\). Strings which cannot be interpreted as numbers in a numerical context are generally have numeric values, which gives awk a capability similar to Snobol tables. Suppose the input contains fields with values like \(\text{apple}, \text{orange}, \text{indigo}\). Strings can be interpreted as numbers, but become numbers in a numerical context.

\[
x[\text{NR}] = 0
\]

assigns the current input record to the NR-th element of the array \(x\). In fact, it is easy to process the entire input in a random order with the awk program

\[
\text{f1} = \text{"$1"} \quad \text{f2} = \text{"$2"}
\]

\[
\text{END} \{ \text{print f1, f2}\}
\]

The first action merely records each input line in the array \(x\). The last action prints the two fields separated by " is ". Variables and numeric expressions may be used in expressions.

### 3.3. Field Variables

Fields in awk share essentially all of the properties of variables defined in section 3.3 without describing it. The condition in parentheses in

\[
\text{if} ($1 == \text{"too big")}
\]

does the same job as the for statement above. There is an alternate form of the for statement which is suited for accessing the elements of a string, which are accessed by setting \(i\) in turn to each element of \(x\). The elements are accessed in a manner similar to variables, but they are read during the loop. The expression in the condition part of an if, while or for can include relational operators like \(==\) ("equal to") and \(!=\) ("not equal to"); regular expression matches with the match operators ~

\[
\text{if ($1 == $2) ...}
\]

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

\[
\text{if ($1 \text{ is numeric})}
\]

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

\[
\text{print $i, $i+1, $i+n}
\]

or assign a string to a field:

\[
\text{if ($3 > 1000)}
\]

\[
\text{print $3 = "too big"}
\]

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

\[
\text{print $i, $i+1, $i+n}
\]

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

\[
\text{if ($1 \text{ is numeric})}
\]

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

\[
\text{print $i, $i+1, $i+n}
\]

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

\[
\text{if ($1 \text{ is numeric})}
\]

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

The \texttt{break} statement causes an immediate exit from an enclosing \texttt{for} or \texttt{while} statement; the \texttt{continue} statement causes execution of the statement immediately following (i.e., \texttt{awk}) to begin.

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

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

\begin{verbatim}
print x, y  # this is a comment
\end{verbatim}

4. Design

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

\texttt{Lex} and \texttt{Cstr} provide general regular expression recognition capabilities, and, by serving as a \texttt{C} program generator, is essentially open-ended in its capabilities. The use of \texttt{Lex} however, requires a knowledge of \texttt{C} programming, and a \texttt{Lex} program must be compiled and loaded before use, which discourages its use for one-shot applications.

\texttt{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 \texttt{C}. Finally, \texttt{Awk} provides a convenient way to access fields within lines; it is unique in this respect.

\texttt{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 \texttt{awk} went into deciding what \texttt{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, \texttt{awk} usage seems to fall into two broad categories. One is what might be called “report generation” — processing an input to extract counts, sums, sub-totals, 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 \texttt{awk} uses the language development tools available on the UNIX operating system. The grammar is specified with \texttt{yacc}; \texttt{yacc} Johnson Cstr the lexical analysis is done by \texttt{lex}; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An \texttt{awk} program is translated into a parse tree which is then directly executed by a simple interpreter.

\texttt{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 \texttt{wc}, \texttt{grep}, \texttt{egrep}, \texttt{fgrep}, \texttt{sed}, \texttt{lex}, and \texttt{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 \texttt{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 \texttt{ls -1}; each line has the form

\begin{verbatim}
-rw--rw-- 1 ava 123 Oct 15 17:05 xxx
\end{verbatim}

The total length of this input is 452,960 characters. Times for \texttt{lex} do not include compile or load.
<table>
<thead>
<tr>
<th>Task</th>
<th>Program 1</th>
<th>Program 2</th>
<th>Program 3</th>
<th>Program 4</th>
<th>Program 5</th>
<th>Program 6</th>
<th>Program 7</th>
<th>Program 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>wc</td>
<td>8.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>grep</td>
<td>11.7</td>
<td>13.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>egrep</td>
<td>6.2</td>
<td>11.5</td>
<td>11.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fgrep</td>
<td>7.7</td>
<td>13.8</td>
<td>16.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sed</td>
<td>10.2</td>
<td>11.6</td>
<td>15.8</td>
<td>29.0</td>
<td>30.5</td>
<td>16.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lex</td>
<td>65.1</td>
<td>150.1</td>
<td>144.2</td>
<td>67.7</td>
<td>70.3</td>
<td>104.0</td>
<td>81.7</td>
<td>92.8</td>
</tr>
<tr>
<td>awk</td>
<td>15.0</td>
<td>25.6</td>
<td>29.9</td>
<td>33.3</td>
<td>38.9</td>
<td>46.4</td>
<td>71.4</td>
<td>31.1</td>
</tr>
</tbody>
</table>

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"}`
7. `/doug/ {print "jdoug"}`
8. `/dmr/ {print "jdmr"}`
9. `{print NR ": " $0}`
10. `{sum = sum + $4}`
11. END {print sum}

**SED:**

1. `=`
2. `/doug/p`
3. `/doug/p`
4. `/doug/d`
5. `/ken/p`
6. `/dmr/p`
7. `/dmr/d`
8. `/dmr/ w jdmr`
9. `/doug/w jdoug`
10. `/doug/ w jdoug`
11. `%{ int i;`