SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 267-279 (1979)

Awk — A Pattern Scanning and Processing Language

ALFRED V. AHO, BRIAN W. KERNIGHAN AND PETER J. WEINBERGER
Bell Laboratories, Murray Hill, New Jersey 07974, U. S. A

SUMMARY

This paper describes the design and implementation of *awk*, a programming language which searches a set of files for patterns, and performs specified actions upon records or fields of records which match the patterns. *Awk* makes common data selection and transformation operations easy to express; for example,

length > 72

is a complete *awk* program that prints all input lines whose length exceeds 72 characters. The program

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

prints each input line with the first field replaced by its logarithm. The program

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

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

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

KEY WORDS Report generation

INTRODUCTION

A surprising amount of data processing amounts to little more than minor transformations on information that already exists — selecting parts of it ("print all the records where the second field is positive"); rearranging ("print the second and third fields of each record, in the opposite order"); and performing simple transformations ("add up the numbers in the first field and compute the average").

None of these tasks is difficult, yet all too often it is a nuisance to encode them in a conventional programming language. Most of the programming effort involves trivial details, like declarations and initialization, handling I/O and conversions, and parsing the input records. Yet when the job is done, the program that results frequently has no long term value — only a one-shot program was needed.

Awk is a program designed to make it easy to write programs of the form suggested above —

0038-0644/79/0409-0267$01.00 *Received 25 December 1978*

simple information retrieval, text manipulation, and report generation procedures. Wherever possible, the trappings of conventional programming languages have been eliminated in favor of simple, concise expression and default handling of common cases. Accordingly, *awk* variables are not declared, and are initialized to either the null string or zero by default. Likewise, data types need not be defined; variables take on numeric or string values as appropriate. The language is interpreted, not compiled, so there are no load modules. Input and file handling is done by an implicit input loop.

Program Structure

The basic operation of *awk* is to scan a set of input lines, searching for lines which match patterns specified by the user. An *awk* program has 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 omitted, but not both. If there is no action for a pattern, the matching line is copied to the output. If there is no pattern for an action, then the action is performed for every input line. A line that matches no pattern is ignored.

Input and sequencing through the patterns is entirely automatic; there is no control flow other than naming the patterns and actions. *Awk* is normally given a set of files to process; the sequencing from one file to the next is also implicit. (The name of the current input file is available as the value of the variable **FILENAME**.)

This model of computation — processing a stream of input by selecting and transforming elements which are "interesting" — has proven to be a useful one on the UNIX† system; *awk* is one in a family of such programs. We will discuss the design in Section 4.

Records and Fields

One of the most valuable services that *awk* performs is to break an input stream into records and fields. A *record* is a string of characters followed by a record separator. Normally the record separator is a newline character, so by default *awk* processes its input one line at a time. The number of the current record is available in a variable named **NR**.

Each input record consists of *fields*, which by default are separated by blanks or tabs. Fields are referred to as **$1**, **$2**, and so forth, where **$1** is the first field; **$0** is the whole input record itself. The number of fields in the current record is available in a variable named **NF**.

The input field and record separators are stored in variables called **FS** and **RS**. They may be changed at any time to any single character. If the record separator is made null, then a blank input line terminates a record, and blanks, tabs and newlines are all field separators. This permits convenient handling of multi-line records.

PATTERNS

A pattern in front of an action acts as a selector that determines whether the action is to be executed. There are four basic patterns:

† UNIX is a trademark of Bell Laboratories.

BEGIN
END
regular expressions
relational expressions

More complex patterns can be formed by combining these elements with boolean connectives and by forming ranges, as described below.

BEGIN and END

The 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 }
```

Regular Expressions

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

```
/smith/
```

This is 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.

More generally, an *awk* regular expression can be:

any single character, which matches itself,
a dot ".", which matches any single character,
a character class (e.g., **[0123456789]**), which matches any single character from the class,
an abbreviated character class (e.g., **[0−9]**),
a complemented character class (e.g., **[^0−9]**), which matches any character *not* in the class,
^, a metacharacter which matches the beginning of a string, and
$, a metacharacter which matches the end of a string.

Regular expressions may be combined to form more complex regular expressions in the following manner.

The regular expression **a|b** where **a** and **b** are regular expressions matches a string if either **a** or **b** matches that string.
The regular expression **ab** (the concatenation of **a** and **b**) matches a string **xy** if **a** matches a suffix of **x** and **b** a prefix of **y**.
The regular expressions **a***, **a+**, and **a?** match respectively zero or more, one or more, and zero or one instances of the regular expression **a**.

As is customary, | has the lowest precedence, then concatenation, then the unary operators *, + and ?. Parentheses may be used to group regular expressions.

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 must be enclosed in slashes. Within a regular expression, blanks and the regular expression metacharacters are significant. The special meaning of any character may be temporarily turned off by preceding it with a backslash:

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

matches any string of characters enclosed in /* and */.

Relational Expressions

An *awk* pattern can be a relational expression of the form

expression relop expression

where *relop* is one of the comparison operators <, <=, == (equal to), != (not equal to), >, >=, or one of the matching operators ~ (matches) or !~ (does not match). For example, the program

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

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. ("%" is the remainder operator.)

In comparisons, if both operand expressions are strings, a string comparison is made; otherwise a numeric comparison is made. 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

```
$1 > $2
```

performs a string comparison.

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

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

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

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

Combinations of Patterns

A pattern can also be formed by combining patterns with parentheses and the operators || (or), && (and), and ! (not). For example,

```
$1 ~ /debit/ && $2 < 0
```

selects lines in which the first field contains the string "debit" and the second is negative. && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in

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

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

```
NR == 100, NR == 200
```

prints lines 100 through 200 of the input, while

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

prints all lines between successive occurrences of **start** and **stop**, or to the end of the input if no terminating **stop** occurs.

Pattern ranges reduce the need to use variables to record state information. Without a pattern range, the previous example would be written

```
/start/       { start = 1 }
start == 1
/stop/ { start = 0 }
```

which is certainly less clear.

ACTIONS

An action is a sequence of statements separated by newlines or semicolons.

Printing

Many *awk* programs do nothing more than print all or part of each record. The program

```
{ print }
```

prints each record, thus copying the input to the output intact. More common is to print one or more fields from each record. For instance,

```
{ print $2, $1 }
```

prints the first two fields in reverse order. Expressions separated by a comma in the print statement will be separated by the current output field separator when printed:

```
{ print NR, NF, $0 }
```

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

The current output field separator and output record separator may be changed by setting the variables **OFS** and **ORS** respectively. The output record separator is appended to the output of each **print** statement.

Output may be diverted to multiple files; the program

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

writes the first field of each line on the file **foo1**, and the second field on file **foo2**. (The output files are created if necessary.) The file name can be the value of an expression as well as a constant:

```
{ print $1 >$2 }
```

uses the contents of field 2 as a file name.

Output may also be diverted into another process, using the pipe facility of the UNIX operating system[1] For instance,

```
NF == 2  { print $2, $1 | "sort" }
```

prints those records with two fields into the program **sort**.

The formatting provided by the **print** statement is usually adequate. When finer control over output format is required, however, the **printf** statement may be used:

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

formats the expressions in the list according to the specification in **format** and prints them. With **printf** no output separators are produced automatically; they must be included explicitly into the **format** string. The capabilities of **printf** are the same as those of the standard I/O library used with the programming language C[2] They include formatting of decimal, octal, hexadecimal, floating point (both **f** and **e** notations), strings, etc.

Built-in Functions

Awk provides a number of built-in functions to facilitate string handling and simple arithmetic. The function **length** computes 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" that yields the length of the current record; **length(arg)** is a function that 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**.

The name of one of these built-in functions by itself, with no arguments 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)** returns as value the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring returned goes to the end of **s**. The function **index(s1, s2)** returns the first position in **s1** where **s2** occurs, or zero if it does not. The function **sprintf(format, expr, expr, ...)** formats the expressions in the **printf** format specified by **format**, and returns the resulting string.

Variables, Expressions, and Assignments

The basic elements in expressions are

floating point numbers
strings
variables
field names
array elements
function calls

Within an expression, these elements can be combined using arithmetic and string operators. All

arithmetic is done internally in floating point. In addition to the usual arithmetic operators +, −, *, /, and % (mod), the C increment ++ and decrement −− operators are also available (both prefix and postfix), as are the assignment operators =, +=, −=, *=, /=, and %=. (++n increments n; x+=y stands for x=x+y.) Note that assignment operators *are* operators: assignments may occur in expressions.

Concatenation is the only string operator. In any expression, adjacent operands are concatenated (after being converted to strings if necessary). 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 ".

The most important aspect of *awk* arithmetic and string operations is that variables and expressions take on either numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x receives a numeric value, while in

```
x = "smith"
```

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

```
x = "0" + "4"
```

assigns 4 to x, whereas

```
x = 0 4
```

assigns the string "04". As another instance, one could print even-numbered records by

```
NR ~ /[02468]$/
```

Strings which cannot be interpreted as numbers in a numerical context will have numeric value zero.

By default, variables (other than built-ins) are initialized to the null string, which has numeric value zero. This default initialization 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 }
```

Field Variables

Fields share all of the properties of variables. Thus one can replace the first field of each input line by a sequence number with:

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

or accumulate two fields into a third:

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

or assign a string to a field:

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

Field references may be numerical expressions, as in

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

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.

Arrays

As shown above, array elements are not declared; like variables, they spring into existence by being mentioned. An array subscript may be *any* non-null value, including a non-numeric string. 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 (up to the limit of memory) 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**.

Any expression can be used as a subscript in an array reference. Thus

```
x[$1] = $2
```

uses the first field of a record (as a string) to index the array **x**.

Non-numeric array subscripts give **awk** a capability rather like the associative memory of Snobol tables. Suppose each line of input contains two fields, a name and a non-zero value. Names may be repeated; the task is to print a list of each unique name followed by the sum of all the values for that name. This can be done with the program

```
      { amount[$1] += $2 }
END   { for (name in amount)
            print name, amount[name]
```

To sort the output, one replace the line by

```
            print name, amount[name] | "sort"
```

in place of the last **print** statement.

Flow-of-Control Statements

Flow of control within an *awk* action can be achieved using the statements **if-else**, **while**, **for**, and statement grouping with braces, as in C.

The **if** statement has the syntax

```
if (condition)
        statement
else
        statement
```

The *condition* is evaluated; if it is true, the *statement* following the **if** is executed. The **else** part is optional.

The **while** statement has the form

```
while (condition)
        statement
```

The *condition* is tested; if it is true, *statement* is executed, and the loop repeats. It terminates when *condition* is false. 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 (expression; condition; expression)
        statement
```

Thus

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

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

There is a variant of the **for** statement that allows one to iterate over the subscripts of an array in an undefined order. Thus as shown above, one can write

```
for (i in a)
        print i, a[i]
```

without having to store the subscripts explicitly in another array.

The condition part of an **if**, **while** or **for** statement can be any relational expression (§2.3) or boolean combination of relational expressions. Note that this permits a condition to contain regular expression matches that use the match operators ~ and !~. Of course, there are a variety of ways to write any particular program; the third example of the abstract may also be written as

```
{ if ($1 != prev) {
        print
        prev = $1
  }
}
```

Finally, there are four statements that control the various loops. The **break** statement causes an immediate exit from an enclosing **while** or **for**; **continue** 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
```

DESIGN AND IMPLEMENTATION

As mentioned earlier, the UNIX system already provides several programs that operate by passing input through a selection mechanism. Programs in the *grep* family[3] print all lines that match a single regular expression. *Sed*[3] 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[4] provides general regular expression recognition capabilities. Because it generates C programs, it is essentially open-ended in its capabilities. The use of *lex* requires a knowledge of C programming, however, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is intended 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, and deciding which representation is appropriate as late as possible. This works well; in most cases, it produces exactly the effect that is wanted.

Most of the effort in developing *awk* went into deciding what *awk* should and 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 expressive but easy to use and well adapted to scanning files that contain both textual and numerical information. For example, implicit initializations and the absence of declarations, while probably a bad idea for a general-purpose programming language, are desirable in a language that is meant to be used for tiny programs that are often composed on the command line.

The development of *awk* was significantly shortened by using UNIX tools. The grammar is specified with *yacc*;[5] the lexical analysis is done by *lex*. Using these tools made it easy to vary the syntax of the language during development. The regular expression recognizers are deterministic finite automata constructed directly from the expressions. Currently, an *awk* program is translated into a parse tree which is then directly executed by a simple interpreter. This aspect of the language has changed radically several times, however, and may well do so again.

EXPERIENCE

Patterns of Use

Awk has been in use for more than a year in a variety of UNIX installations at Bell Laboratories. Its usage seems to fall into three broad categories. One is what might be called "report generation" — processing data to extract information and produce simple statistics like counts, averages, sub-totals, etc. For example, we use an *awk* program to summarize the information in the file where *awk* usage data is recorded.

A second major area of use is as a data transformer, converting information from the form produced by one program or person into that expected by another. The simplest examples merely select fields, perhaps with rearrangements. In fact, it appears that if *awk* provided nothing more than the ability to select fields, it would still be a widely-used program. Another example is a program which converts map coordinates in one projection into another so a map can be drawn.

A third area of application is the writing of simple data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly positioned. The combination of textual and numeric processing is invaluable here; most data validation tasks seem to involve both.

Timing

Awk was designed primarily 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 seems adequately fast for most purposes. Table I shows the execution (user + system) time on a PDP-11/70 of the programs *sed*, *lex* and *awk* on the following simple tasks:

1. count the number of lines:

```
END { print NR }
```

2. print all lines containing "doug":

```
/doug/
```

3. print all lines containing "doug", "ken" or "dmr":

```
/ken|doug|dmr/
```

4. print the third field of each line:

```
{ print $3 }
```

5. print the third and second fields of each line, in that order:

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

6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively:

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

7. print each line prefixed by "line-number : ":

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

8. sum the fourth column of a table:

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

In all cases, the input was a file of 450,000 characters, in 10,000 lines. Each line contained 8 fields.

For small input files, the predominant speed problem is simply that *awk* is a big program (50K bytes) and thus takes longer to be loaded than does a small program.

Clearly there is a limit to the size of file for which one can economically use linear search. It is an interesting practical problem to make the program fast at linear search, but it is also a good research problem to make it work in a natural way with files stored in some other way. For instance, it is possible to search for restricted classes of regular expressions[6] without looking at all the characters; we have not attempted to use such an algorithm.

Table 1: Timing Comparisons

Task	awk	lex	sed
1	15.0	65.1	10.2
2	25.6	150.1	11.6
3	29.9	144.2	15.8
4	33.3	67.7	29.0
5	38.9	70.3	30.5
6	46.4	104.0	16.1
7	71.4	81.7	
8	31.1	92.8	

Directions for Future Work

Considerable discussion (and compromise) has gone into the design of *awk*. One obvious defect is the lack of a subroutine facility. This deficiency is not serious for small, one-time applications, but as users become more experienced and sophisticated, the call for subroutines may become more compelling. The trend toward longer, more complex user programs also points out the shortcomings of the terse and uninformative syntax error messages produced by the current implementation.

The ability to handle numbers, strings and regular expressions at the same time seems to have worked out well. In most cases, each variable has only one type, and in those cases where coercion is necessary (most often for comparison), the results are what the user expects. It is true that when the coercion is not the right one, the results can be mystifying to the uninitiated, but this has happened infrequently enough so as to have an acceptable cost.

At the moment, *awk* has no explicit command for text substitution, although some substitutions can be implemented (clumsily) with **substr** and **length**. Incorporating a general text substitution command would affect the type of regular expression matching algorithm used. The algorithm currently used constructs a compact deterministic finite automaton directly from the regular expression. The automaton finds the shortest possible match with no backtracking. This approach is good for fast matching but for textual replacement it is necessary to isolate the leftmost longest substring that matched. We have also considered adding a dynamic regular expression capability, but have not yet found the applications to justify this inclusion.

Also under consideration is some way of making RPG-like "control breaks" easier to program. In this kind of data processing, an action is invoked when the value of some field changes from one record to the next. This can always be done with "state variables " but the resulting programs are messy and unclear. Pattern ranges address only simple aspects of the problem.

Finally, we have already done some experimentation with a language based on *awk* that has normal control flow and input statements. This is a very high level language, which permits arbitrary file manipulations, at the price of somewhat more programming for those cases where *awk* already does the job.

CONCLUSIONS

Awk has been used in a broad variety of tasks, ranging from budget preparation, through telephone network planning, even to finding Chinese restaurants within walking distance of theaters in Manhattan. It has demonstrated that there is a considerable market for a stand-alone language which enables common data processing tasks to be done with little programming effort. Because of its ease of use, *awk* has often been used to perform bookkeeping and data manipulation tasks

that otherwise would have been done manually or not done at all.

The UNIX programming environment is heavily based on the idea that programs should communicate with each other. It has been our experience that *awk* is of considerable help here too, as a data transformer between a program which produces output in one form and a second that requires it in some other form. In a sense, *awk*, like many other UNIX programs, acts as an "impedance matcher," adapting one program to another.

REFERENCES

1. D. M. Ritchie and K. Thompson, 'The UNIX Time-Sharing System,' *Comm. Assoc. Comp. Mach.*, **17**, 7, 365-375 (1974).
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories, 1975. Sixth Edition.
4. M. E. Lesk, 'Lex — A Lexical Analyzer Generator,' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories (1975).
5. S. C. Johnson, 'Yacc — Yet Another Compiler-Compiler,' Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (1975).
6. R. S. Boyer and J S. Moore, 'A Fast String Searching Algorithm,' *Comm. ACM*, **20**, 10, 762-772 (1977).