SIGPLAN Notices, V18 #12, December 1983

A Walk Through AWK

Leon S. Levy
Bell Laboratories
Whippany, New Jersey 07981

ABSTRACT

This tutorial on AWK is intended for readers who have a general familiarity with UNIX*, and who have at least a rudimentary knowledge of programming in some procedural language.

AWK is an interpretive programming language, which makes it useful for prototyping. Primary applications of AWK are information retrieval, report writing, and data manipulation.

AWK is often used as a special purpose information processing language - many of whose capabilities are easily accessible to non-programmers. We present it as a programming language - with added capabilities - particularly useful for rapid prototyping.

(Address through 8/84:
c/o Prof. Aryeh Levy
Ben Gurion University
Dept. of Mathematics
& Computer Science
Beer Sheva, ISRAEL)

CONTENTS

A Walk Through AWK

Leon S. Levy

Bell Laboratories
Whippany, New Jersey 07981

ABSTRACT

This tutorial on AWK is intended for readers who have a general familiarity with UNIX*, and who have at least a rudimentary knowledge of programming in some procedural language.

AWK is an interpretive programming language, which makes it useful for prototyping. Primary applications of AWK are information retrieval, report writing, and data manipulation.

AWK is often used as a special purpose information processing language - many of whose capabilities are easily accessible to non-programmers. We present it as a programming language - with added capabilities - particularly useful for rapid prototyping.

1. Introduction

AWK is a UNIX* system programming tool. In this tutorial we introduce AWK and describe features and typical applications. No prior knowledge of AWK is assumed, but programming in some algorithmic language is assumed. Also, a basic knowledge of editing in the UNIX system is assumed - ed [Kernighan] or vi [Joy]. In many AWK applications, a knowledge of UNIX shell programming methods is useful and the requisite concepts will be presented in situ as required. The ability to read programs in the C language [Kernighan and Ritchie] is assumed; C is sufficiently similar to other high level languages that an otherwise knowledgeable, intrepid reader need not be deterred.

Familiarity with the UNIX shell is useful, too. A useful overview of the UNIX system is contained in [Bourne], which also discusses the shell.

* UNIX is a Trademark of Bell Laboratories

2. What is AWK?

AWK (named for its developers - Al Aho, Peter Weinberger, and Brian Kernighan) is a programming language and system with many characteristics that make it a convenient problem solving tool. Important characteristics of the language are its ability to process files and refer to input records and their fields, and pattern matching on these items. Typically, an input record is one line of an input file and fields are delimited by white space.

The regular expression pattern descriptors of AWK are like those of egrep, [UNIX User's Manual], and many of the control structures are similar to C [Ritchie]. AWK is also similar to interpretive languages in that no declarations are used.

Typical applications of AWK make use of these features. Among such applications are information retrieval and report writing, especially on record oriented data bases, which makes use of the record oriented input. Another use of AWK is for data manipulation, or validation, exploiting the pattern matching features.

AWK is often thought of as a sequence of pattern statements, in which each action is preceded by a pattern guard. Thus the program

```
/AWK/ {
        print NR
      }
```

when run against a file will produce a list of the line numbers of all the lines in which the sequence of letters AWK appears.

3. Hello World

The simplest way to start is to edit the program into a program file called program1.awk. (It is not necessary to name program files with a .awk suffix but a convention helps to identify AWK programs.) This program can then be run by giving the command

```
awk -f program1.awk
```

The AWK program will read data from the standard input if no data files are named in the command. If the program is to process records from data files, in addition to the program file, their names will follow the command, as in:

```
awk -f program1.awk datafile1 datafile2 ...
```

Each AWK program consists of three parts:

- a <BEGIN section>,
- a <pattern statement section>, and
- an <END section>.

We will start with the <BEGIN section> which is run before any input records are processed. (The <END section> is run after all input records have been processed, and the <pattern statement section> is data driven.) The following program prints Hello World!.

Program #1. "Hello World!"

```
BEGIN{
      print "Hello World!"
      exit
}
```

Program #1 is entirely contained in the <BEGIN section> and, therefore, is run before any data are read. (Since AWK programs are normally terminated by the end of input, the exit statement is needed here to cause the program to end.)

NOTE: Brackets, {}, are used to define the limits of the sequence of statements associated with the <BEGIN section>. They are similar to the bracketing pairs of words used in other block structured languages, such as do .. od and begin .. end.

Exercise. Enter Program #1 into a file called hello.awk Then give the command:

A Walk Thru AWK Levy

```
awk -f hello.awk
```

A more elaborate example is the following AWK version of the Euclidean algorithm to compute the greatest common divisor of a pair of numbers. (The greatest common divisor of a pair of numbers is the largest integer that divides each of the numbers.)

Program #2.Euclidean Algorithm

```
BEGIN{
      i = 75 ; j = 30
      while(i != j){
            if (i > j)
                  i -= j
            else
                  j -= i
      }
      print "The greatest common divisor" \
                        "of 75 and 30 is " i
      exit
}
```

NOTE: The statement

```
i -= j
```

is equivalent to the statement

```
i = i - j
```

This type of construct is used in Algol 68 and C.

NOTE: AWK is an interpretive language. There are no declarations, the types of variables being inferred by the AWK processor. Further, a variable may take on different types of values within the same program.

NOTE: More than one statement may appear on a line if the statements are separated by semicolons.

NOTE: Statements may be continued on succeeding lines by terminating the line with a '\'. When a quoted string is to be continued on a succeeding line, it must be treated as the concatenation of component substrings, since '\' may not be used within a quoted string. (See Section 11 for an example of concatenation.)

If Program #2 is in a file euclid.awk, then the command:

```
awk -f euclid.awk
```

will print The greatest common divisor of 75 and 30 is 15.

Exercise. Run the Euclidean algorithm program.

In program 2, the data on which the program operates are built in. More generally programs operate on records in files. Unless otherwise specified:

- a record is a single line in a file,
- fields within a record are strings of non-white-space characters separated by white space characters, i.e. blanks or tabs. (Both record and field delimiters can be specified by the user. See Section 5.)

  $i refers to the ith field of a record.

Program #3 is an AWK version of the Euclidean algorithm that computes the greatest common divisors of pairs of numbers. The data are stored as pairs of numbers, one pair on each line of the input data file. Each pair of numbers thus constitutes two fields of a record.

Program #3. Euclidean algorithm - to be stored in euclid.awk

```
{
        arg1 = $1 ; arg2 = $2
        while (arg1 != arg2) {
                if (arg1 > arg2)
                        arg1 -= arg2
                else
                        arg2 -= arg1
        }
        print "The greatest common divisor of " $1 \
                                        " and " $2 " is " arg1
}
```

Given the following data in a file called datafile1:

```
75      30
24      60
360     224
```

the command:

```
awk -f euclid2.awk datafile1
```

will generate the following output:

```
The greatest common divisor of 75 and 30 is 15
The greatest common divisor of 24 and 60 is 12
The greatest common divisor of 360 and 224 is 8
```


Exercise. Enter the euclid2.awk program and run it on a data file with 5 pairs of numbers.

Notice that Program #3 does not have a <BEGIN section> since there is no computation required before reading the first record. In general, the <BEGIN section> is used as the initialization part of the AWK program. Correspondingly, there is an <END section> when some computation is required after all records have been processed.

Program #4. rms

```
        # A program to compute the square root of the
        # sum of the squares of a set of numbers.
        # The set of numbers is provided as input -
        # one number to a record.
        #
        # NR is the current record number. In the END
        # section it is the number of records.

        BEGIN{
                sum_of_squares = 0
        }
        {
                sum_of_squares += $1 * $1
        }
        END{
                root_mean_square = sqrt(sum_of_squares / NR)
                print root_mean_square
        }
```

NOTE: The use of comments in Program #4. A # is a comment marker - anything following it on a line in the program file is treated as comment.

In Program #4, the <BEGIN section> initializes sum_of_squares to 0. (AWK initializes all numeric variables to 0, so the <BEGIN section> is not really needed here.) The fifth line of Program #4 , ignoring blank and comment lines, is run on each record and adds the square of the first field of the record to sum_of_squares. Since each input number is stored in a separate record and constitutes the first (and only) field of that record, the effect is to add up the squares of all numbers. Finally, the <END section> uses a built-in AWK variable, NR, and a built-in AWK function, sqrt(), to compute the root_mean_square. The root_mean_square of a set of items is defined as the square root of the sum of the squares of the items divided by the number of items. NR is a built-in AWK variable that is a count of the number of input records encountered. When processing the ith record, the value of NR is i. Thus in

the <END section>, NR is a count of the number of records. In Program #4, therefore, it is a count of the number of items. Sqrt() is the square root function.

NOTE: The use of comments in Program #4. A # is a comment marker - anything following it on a line in the program file is treated as comment.

Exercise. Run Program #4. Have the input data consist of both integers and decimal numbers. Note that AWK does not distinguish between these.

4. Some AWK Syntax

The basic AWK syntax can be summarized in the following BNF, Backus-Naur Form, [Carberry] or [Levy], definition:

```
<Program>                        ::=  <BEGIN section>
                                      <pattern statement section>
                                      <END section>

<BEGIN section>                  ::=  BEGIN{
                                            <statement list>
                                      } |
                                      empty

<pattern statement section>      ::=  <pattern statement list> |
                                      empty

<END section>                    ::=  END{
                                            <statement list>
                                      } |
                                      empty

<statement>                      ::=  "A statement in a C like syntax"

<pattern statement>              ::=  <pattern>{
                                            <statement list>
                                      } |
                                      <pattern>
```

The English language rendition of this grammar is:

A program consists of a <BEGIN section> followed by a <pattern section> followed by an <END section>. (The <>'s denote syntactic categories, ::= is read "is defined as", and | is read as "or".)

- A <BEGIN section> consists of the word BEGIN - all caps - followed by a list of statements surrounded by braces, or a <BEGIN section> is empty.
- An <END section> consists of the word END followed by a list of statements surrounded by braces, or is empty.
- The <pattern statement section> consists of a sequence of <pattern statement>'s, or may be empty. If the <pattern statement section> is not empty, then each <pattern statement> consists of a <pattern> followed by a <statement list> enclosed

in braces, or is just a <pattern> and the default action is to print the record.

- A <statement list> is a sequence of one or more <statement>'s. This could be described by the BNF rule:

```
<statement list>        ::=   <statement list>
                              <statement> |
                              <statement>
```

Similarly, <pattern statement list> can be cast in BNF form.

In Programs #1 & #2 only the <BEGIN section> is non-empty. In Program #3 both the <BEGIN section> and the <END section> are empty. In Program #4 all sections are non-empty. In Program #4, the pattern section is:

```
{
        sum_of_squares += $1 * $1
}
```

In this case, the <pattern> is empty. An empty <pattern> matches every record. (BEGIN and END can also be understood as patterns matching the start and end of input, respectively.)

I am indebted to Mike Bianchi for convincing me of the importance of using uniform stylistic conventions in the program text. The layout of the grammar suggests this layout by vertically aligning the {} delimiters and indenting the hierarchically nested statement block.

5. Predefined Variables in AWK

In Program #4, the AWK built-in variable NR was used. The following is a list of the built-in variables in AWK:

```
NR - number of records
NF - number of fields
FS - input field separator
RS - input record separator
$i - ith input field of the current record
$0 - current input record
OFS - output field separator
ORS - output record separator
```

Program #5: number.awk

```
# This program provides line numbering

{
        print NR, $0
}
```

Program #6: fields.awk

```
# print a listing of line numbers
# followed by number of fields per line

{
        print NR , NF
}
```

In Programs #5 and #6, the comma in the print statement causes the output field separator to separate the values of NR and NF. By default, OFS is a blank, but it may be reset as in Program #6a. (Beware that if the commas are not included in Program #6 the values of NR and NF would be concatenated.)

Program #6a.fields1.awk

```
BEGIN{
        OFS = ""
}
{
        print NR , NF
}
```

Exercise. Rewrite Program #4 to allow several numbers on a line by changing RS in the <BEGIN section>.

6. Review

Here are some AWK programs. Try to analyze these programs and determine what they do - even if you have to run them. (The answers are given following the review problems.)

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

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

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

4.
```
{
        print NR " : " $0
}
```

5.
```
{
        sum += $4
}
END{
        print sum
}
```

6.
```
{
        s += $1
}
END{
        print " sum is " s , " average is " s / NR
}
```

7.
```
{
        for ( i = NF ; i > 0 ; i-- )
                print $i
}
```


Answers to Review Problems:

1. Print the number of lines in the input.
2. Print the third field in each record.
3. Print the third and second fields of each record.
4. Print the input with line numbers on the left, # : line.
5. Compute and print the sum of the fourth fields of the input records.
6. Compute and print the sum and average of the first fields of all records.
7. Print the fields of each record in reverse order (i.e. last field first), one field per line.

7. Patterns

AWK may be thought of as a pattern driven language in which the standard paradigm is a sequence of pattern-action pairs. In this view of AWK each pattern acts as a guard for the statement following it. The three types of patterns that AWK comprehends are:

1. regular expression patterns,
2. arithmetic patterns, and
3. range patterns.

Regular expression pattern specification in AWK is similar to regular expression pattern specification in ed, [Kernighan], and is based on the specification of a regular expression defining the set of strings that constitute the pattern. The pattern is delimited by pairs of /'s. Within a pattern, most symbols denote themselves*, with the exception of the special symbols to be described. Thus /a/ denotes the pattern consisting of the single letter a, and /ab/ denotes the pattern consisting of the sequence of two letters ab.

The following symbols have special meanings within regular expression patterns.

* More precisely, most symbols denote the singleton set containing that symbol.

Symbol	Meaning	Examples
|	alternation	a|b means 'a' or 'b' ab|cd means 'ab' or 'cd'
+	one or more	a+ means one or more a's
?	zero or one	a? means zero or one a's
*	zero or more	a* means zero or more a's
[...]	pattern class	[ab] means a or b [aeiou] means any vowel [a-z] means any lower case letter
^	beginning anchor	^[a-z] means a pattern beginning with any lower case letter
$	ending anchor	[0-9]$ means a pattern ending with a decimal digit.
.	any character	

^[a-zA-Z][a-zA-Z0-9]*$ is a pattern whose first symbol is a letter, and whose succeeding symbols, if any, are letters or digits.

Recall that a <pattern statement> is a <pattern expression> followed by a <statement list>. The <pattern expression> may consist solely of a pattern in which case the <pattern expression> matches iff $0 satisfies the pattern. A <pattern expression> may also be formed using the relational operators:

```
==      equals
!=      does not equal
>       is greater than
>=      is greater than or equal to
<       is less than
<=      is less than or equal to
```

or the pattern operators:

```
~       matches
!~      does not match
```

The <pattern expression> $1 ~ /a/ is true whenever field # 1 contains an a, the <pattern expression> $1 ~ /^a/ is true if field #1 starts with an a, and $1 ~ /^a$/ is true if field #1 is just the letter a. Other examples are:

```
$2 !~ /1|a/                 true if field #2 contains
                               neither a '1' nor an 'a'

$3 ~ /^this$|^that$/        true if field #3 is exactly
                                    'this' or 'that'.
```

<pattern expression>'s may also be composed into more complicated <pattern expression>'s by using the Boolean connectives and, or, and not denoted by the symbols &&, ||, and !.

Example: (x ~ /a/ || x ~ /b/) && x !~ /c/

is a pattern that is true iff x is a string variable which has an 'a' or a 'b' in its string value, and does not have any 'c' in its string value.

A <pattern statement> consists of a <pattern expression> followed by a bracketed <statement list> which is executed if the <pattern expression> is true. However, if the bracketed <statement list> is missing, the default action is to print the record if the <pattern expression> is true.

Program #7.

```
NF >= 2 {
        for(i=1 ; i <= NF ; i++)
                sum[i] += $i
        cnt++
}
END{
        for(i = 1 ; i <= NF ; i++)
                printf "%g ", sum[i]
        printf "\n"
        print cnt " records with 2 or more fields"
}
```

The <pattern statement> in Program #7 has a pattern predicate that is true whenever there are two fields in a record. This is an example of an arithmetic pattern. Since the default input record separator is a carriage return, the bracketed part of the statement will be executed whenever there are two or more fields on a line of input. Cnt will maintain a count of the number of lines having two (2) or more fields, since cnt is initially zero, and is incremented once each time the pattern predicate NF >= 2 is satisfied.

Also, an array sum is maintained, whose elements are initially zero. The ith element of the array sum is denoted sum[i]. Each time the pattern predicate is satisfied, field #1 is added to sum[1], field #2 is added to sum[2], etc. The <END statement> will print, in g format, [Kernighan and Ritchie], the elements of sum. The record count, cnt, will appear on a separate line, followed by the words records with two or more fields.

The printf statement in Program #7 is similar to the printf subroutines of C. The first argument of printf is a format specification in which the %g is to be replaced by the value of the argument in 'floating point' format.

Patterns may be combined by logical connectives. The pattern $1 ~ /a/ && $1 ~ /b/ is true if field #1 has both an 'a' and a 'b'; their order is immaterial.

Exercise. Write, and test, a pattern that will be true iff field #1 has an 'a' and a 'b' following it but not necessarily immediately.

If two patterns are separated by commas, the pattern match will be set by the first pattern and will remain true until reset by the second pattern. This is called a range pattern. For example, the program:

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

is a <pattern statement>. The first pattern is matched by any line containing the pattern /start/ and the second pattern being matched by any line containing the pattern /stop/. Since there is no statement following the pattern, the default action of printing the input record is taken and printing is turned on by /start/ and turned off by /stop/. The lines containing the start and the stop are printed.

8. Review #2

Here are some AWK programs. Analyze these programs and determine what they do - even if you have to run them. (The answers are given following the review problems.)

1.
```
length > 72
```

2.
```
/doug/
```

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

4.
```
$1 != prev {
      print
      prev = $1
}
```

5.
```
!/!/
```

6.
```
/hello/,/goodbye/
```

Answers to Review Problems

1. Print all lines whose length exceeds 72.

2. Print all lines containing the sequence 'doug'.

3. Print all lines containing any of the sequences 'ken', or 'doug', or 'dmr'.

4. Print any line whose first field differs from the first field of the previous line. (Will the first line be printed?)

5. Print all lines that do not contain any exclamation points. (Useful if '!' is used as a comment flag.)

6. Start printing at lines that contain hello and stop printing at lines that contain goodbye.

9. Output Statements

AWK has two output statements, the <print statement> and the <printf statement>; the <printf statement> gives the programmer more flexibility by allowing explicit specification of the output formats. The syntax of these statements is:

```
<print statement> ::=
      print [<expression list>]   [<destination part>]

<printf statement> ::=
      printf <format> [, <expression list>]   [<destination part>]
```

The format specification in the <printf statement> follows the rules of the C printf statement, [Ritchie]. Since this is so, the following examples should suffice to get the reader started:

1.
```
print x
```

causes the value of x to be printed on the standard output, followed by a newline.

2.
```
print $1, $2, $3 > "file1"
```

causes the values of fields 1, 2, and 3, separated by the output field separator, OFS (the default is a space), to be written to a file named file1. The output of each <print statement> is followed by the output record separator, ORS (the default is a newline). If file1 does not exist it is created; an existing file is overwritten.

3.
```
print $3 >> "file2"
```

appends field #3 to file2. Thus if file2 exists this output will appear at the end of the file. If file2 does not exist, it is created.

4.
```
print $1 $2 $3 > "file1"
```

prints the values of fields 1, 2, and 3 in file1, as in Example 1, above, except that output field separators do not appear between the fields.

5. printf "%s ", $3 > "file2"

   prints the value of field #3 to file2 followed by a space, but does not generate an ORS at the end of the field. Thus subsequent <printf statement>'s may add to this 'line' of output data.

6. Finally, field widths may be specified as in

```
printf "%-20s%5d\n", $3 " = " $2, $1 >> $4
```

   AWK constructs the string $3 " = " $2 by concatenating field #3, the literal " = ", and field #2, printing it left justified in a field of width 20; prints field #1 in integer format right justified in a field of width 5; concatenates all this and terminates with a newline character, \n. The output of the program is appended to the file whose name is the value of $4. (See [Kernighan & Ritchie] for a discussion of formats in C.)

10. The Command Line

AWK may take its input program from the command line directly as in:

```
awk "length > 72" datafile1 datafile2 ...
```

that runs the AWK program

```
length > 72
```

on all the data files listed in the command line. In this mode of program input, the string following the AWK command is the program and the usual shell quoting conventions are in effect. (See below - shell interactions.)

A second mode of program input is

```
awk -f program.awk datafile1 datafile2 ...
```

where the name of a file containing the AWK program is given following the -f specification on the command line.

In either mode of program input, if the data files are omitted from the command line, then data is read from the standard input.

11. More of the Language

The following are built-in constructs in AWK:

- length() - length(x) is a function whose value is the number of characters in the string, x. If x is a number, then length(x) is the number of characters used in the output representation of x. [length, with no arguments, is equivalent to length($0).]

- sqrt(), log(), exp() - The mathematical functions: square root, natural logarithms, and exponentiation.

- int() - The integer function, int(x), yielding the largest integer less than or equal to x, if x is positive. If x is negative, int(x), is the smallest integer greater than or equal to x.

- next - Discontinue processing the current record, and read the next record, if any.

- exit - If not in the *END* section then invoke the *END* section; otherwise, terminate the AWK program.

- substr(,,) - The substring function. Substr(s,m,n) gives the n character substring of s starting at position m. If s is a number, its print image is used;

  substr(123456789,3,4) is 3456.

  If the third component is omitted, the result is the substring of s starting at position m; i.e. the rest of the string:

  substr("123456789",3) is 3456789.

  Try substr(123456789,3) , without the quotation marks!

- index - index(s,t) is a function whose arguments are strings, and whose value is the starting position of the leftmost occurrence of t in s. If t is not a substring of s, then index(s,t) is 0.

  index("Kernighan", "nigh") is 4.

- FILENAME - The name of the input file currently being read. This is useful since AWK commands are typically of the form

  awk -f programfile datafile1 datafile2 ...

- arrays - Arrays may be used to store aggregates of data. Since AWK is a language in which variables are not declared, the arrays need not be dimensioned before use. w[i] denotes the ith item of the array w. i need not be a numeric variable. The array iterator, see below, can be used to process all elements of an array.

  Example: French["goodbye"] = "adieu".

  Extending this simple example, it is possible to construct a database within AWK and to retrieve data using the keys.

- split - split(x,y) assigns the fields of the string, x, to successive elements of the array, y.

  Example. split("Now is the time",w) assigns the value "Now" to w[1], "is" to w[2], "the" to w[3], and "time" to w[4]. Split initializes an array, so that elements of the array which antedated the split are lost.

  The value of split is the number of components into which the string argument was divided.

  Example. n = split("Fourscore and seven years ago",Lincoln) assigns the value 5 to n.

  split(x,y,"z") assigns the fields of the string, x, to successive elements of the array, y, using the character 'z' as the string x field separator.

- OFMT - The format for printing numbers. The default is "%.6g". (See [Kernighan and Ritchie] - pp. 145-147 for discussion of print formats;also, printf(3S) in [UNIX Users's Manual].)

- while - The usual while construct.

- for - The iterative construct in AWK. There are two versions of this construct in AWK:

  1. The C language construct; example:

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

     prints each field of a record preceded by the number of the field. (See [Ritchie], Section 9.6 .)

2. An array iterator, as in

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

This will print each index value, followed by the array element for that index. (Note that indices need not be integer, or numeric. If the indices are not numeric the output is not sorted.)

3. break, continue - similar to their counterparts in C.

- concatenation - juxtaposition of strings yields their concatenation.

Example: If x = "hello" and y = "goodbye", then x y is "hellogoodbye".

12. Applications

12.1 Data Validation

Typical data validation checks insure that a record has the appropriate number of fields, that each field is in proper format (i.e. matches a given pattern), or that fields satisfy a given relationship. Some examples of validation checks are:

1.
```
NF != 5 {
        print NR, "wrong number of fields"
}
```

2.
```
$1 < $2 {
        print NR, "field 1 < field 2"
}
```

3.
```
$3 !~ /^[a-zA-Z]/ {
        print NR, "field #3 = ", $3
        print "- does not start with a letter"
}
```

Of course, a typical validation would consist of many such checks that can be combined into a single AWK program.

12.2 Data Transformation

Data transformation takes input data and alters it in some way. Typically such data transformations may rearrange fields, delete fields, add syntactic markers, or apply mathematical transformations (e.g polar to Cartesian coordinates).

The following program has been used to shorten lines generated by one stage of a code generator, to accommodate the input line length limitation of a PL/1 compiler:

Program #8:

```
# Assumes that all fields are < 58 characters

{
        image = " "                     # the image of the output
        fldptr = 1                      # the field of the record

        if ($0 ~ /^$/ ){                # print blank lines
              print
              next
        }

        while(fldptr <= NF){
              sofar = length(image) + length($fldptr)
              if ( sofar <= 64 ) {
                    image = image " " $fldptr
                    fldptr++
              }
              else {
                    print image
                    image = "       "
              }
        }

        print image
        next
}
```

Successive fields within a record are concatenated, separated by blanks, to form the image line - up to a length of 64. The first line formed in this way from a record is printed with a single blank at its left-hand side; subsequent lines are indented by a fixed amount. Blank input lines are printed.

13. Shell Interaction

Interaction with shell level programs is via files that may be created by or serve as input to AWK programs, or by means of the AWK program that may be a string subject to the usual shell parameter passing conventions. In the shell, within a string bounded by double quotes, substitution for shell variables occurs. Within a string bounded by single quotes, substitution for shell variables does not occur. The following program segment shows how the shell variables $1 and $2 may be assigned to the AWK variables first and second:

```
awk "BEGIN {
            first = \"$1\"
            second = \"$2\"
      }"'
      $1 == first || $2 == second' $3
```

This shell program has three arguments, the third argument specifying the name of the input file. The program prints out any record whose first field matches the first shell argument or whose second field matches the second shell argument. If this shell program is stored in an executable file called match, then the command

```
match x y z
```

applied to the file z containing the following data:

```
x y 1 2 3
x z 1 3 5
w y 2 4 6
u v 3 5 7
```

will print the following output:

```
x y 1 2 3
x z 1 3 5
w y 2 4 6
```

The role of the backslash, '\', is central here. The backslashes tell the shell to treat the quotation marks immediately following the '\' as just a character to be passed on to AWK - not as a string terminator.

NOTE: While the shell has three input arguments, AWK has only two. The first argument to AWK is a single string starting at BEGIN and ending at second. Two quoting styles are used to construct this string:

double — Within a string bracketed by double quotes the shell will substitute for shell variables. The $1 and $2 which occur between the BEGIN and its closing bracket thus represent shell variables for which the shell substitutes their values. The backslashes on the second and third lines of the program cause the shell not to interpret the characters immediately following them.

single — Within a string bracketed by single quotes the shell does not substitute for string variables.

The AWK program constructed by the shell command

```
match x y z
```

is:

```
BEGIN {
      first = "x"
      second = "y"
}
$1 == first || $2 == second
```


14. Report Writing

No AWK tutorial could be complete without a section on report writing - although this one almost was!

The following program is an AWK report program to scan a file and itemize all lines where a given keyword is found:

```
awk "
                        # Prepare a formatted listing of all occurrences
                        # of a given keyword in a file.
                        #
                        # keyword is $1 - the first shell parameter
                        # filename is $2 - the second shell parameter

BEGIN{
      firstpage = 1
      firsttime = 1             # true the first time that keyword
                                # occurs on a page
      pagesize = 16
      outlines = 0
}

outlines % pagesize == 0 {      # new page initialization

      if ((firstpage != 1) &&                       \
          (length(lines) != 0)) {
                                # if not the 1st page
                                # print the last line of the
                                # preceding page

            printf \"%s\n\",lines
      }
      else
            firstpage = 0

                                 # print the page header

      print \"\n\n  Pages and lines where  $1  occurs \n\n\"
      print \" Page        Lines  \"
      print \" ____        _____   \n\n\"

      outlines = 8
      newoutpage = 1
}
/$1/ "'{                        # if the first shell parameter
                                # occurs in the current record

      oldpage = page
      page = int ( NR / 66 ) + 1
```

```
        lineno = NR % 66
        if (page != oldpage){
                if ((firsttime == 1 ) ||     \
                    (newoutpage == 1)){
                      printf "%5d       ",page
                      firsttime = 0
                      newoutpage = 0
                }
                else {
                      printf "%s\n", lines
                      outlines++
                      sheet += 1
                      printf "%5d       ", page
                }
                lines = ""
        }
        if (lines == "")
                lines = lineno
        else
                lines = lines ", " lineno
}
END {
        printf "%s\n", lines
} '  $2
```

If this program is stored in an executable UNIX file, called printindex, and the print image of the tutorial is in a file tutorial, then the command:

```
printindex AWK tutorial
```

will generate output of the form:

Pages and lines where AWK occurs

Page	Lines
1	4, 15, 20, 21, 26, 27, 28, 31
2	4, 10, 19, 21, 24, 27
3	4, 13, 18, 25, 47
4	4, 10
5	4, 15, 57
6	4, 32, 39, 40, 43, 55
7	4, 10
8	4
9	4

A Walk Thru AWK Levy

Pages and lines where AWK occurs

Page	Lines
10	4, 10
11	4
12	4, 10
13	4
14	4
15	4, 10
16	4, 10
17	4, 19
18	4

This report writer is just another AWK program. The user must count input lines and generate page headers, and use the printf statement to control the format of the report. Note that since the program knows about pages of output, it would be easy to include page numbering in the report.

Using the shell interactions discussed in Section 13, a date variable may be initialized in the <BEGIN section>, using the UNIX date command, and used to print the date on the report.

Acknowledgement

I gratefully acknowledge the many helpful comments of Mike Bianchi, Fran Henig, Jordan Levy, John Mashey, Jim Salisbury, Susan Strauss, and Ted Stump. Also, Joel Sturman who asked me to give an AWK talk two years ago, which was the seed for this tutorial. The faults in this tutorial are my own.

REFERENCES

[1] Aho, A.V., Kernighan, B.W., and Weinberger, P. "AWK - A Pattern Scanning and Processing Language", TM 77-1271-S, September 8, 1977

[2] UNIX User's Manual

[3] YACC grammar for AWK

[4] Aho, A.V., Kernighan, B.W., and Weinberger, P. "AWK - A Pattern Scanning and Processing Language", Software - Practice and Experience, April, 1979, p. 267-280.

[5] Bourne, S.R. The UNIX System, Addison-Wesley, 1983.

[6] Kernighan, B.W. "An Introduction to the Unix Editor - ed" UNIX Starter Package

[7] Joy, William "An Introduction to Display Editing with VI" Berkeley UNIX Programmer's Manual

[8] Kernighan, B.W. and Ritchie, D.M. The C Programming Language, Prentice Hall, 1978

[9] Ritchie, D.M. "C Reference Manual"

[10] Carberry, S., H.M. Khalil, J.F. Leathrum, L.S. Levy, Foundations of Computer Science, Computer Science Press, 1979, p.85-98.

[11] Levy, L.S. Discrete Structures of Computer Science, Wiley, 1980, p.182-196.