Introduction to awk

Part 1 of 3

Original Author : Brian Brown, CIT

Revision : 1.0

Date : 05/05/94

INTRODUCTION

awk is a programming language designed to search for, match patterns, and perform actions on files. awk programs are generally quite small, and are interpreted. This makes it a good language for prototyping.

THE STRUCTURE OF AN AWK PROGRAM

awk scans input lines one after the other, searching each line to see if it matches a set of patterns or conditions specified in the awk program.

For each pattern, an action is specified. The action is performed when the pattern matches that of the input line.

Thus, an awk program consists of a number of patterns and associated actions. Actions are enclosed using curly braces, and separated using semi-colons.

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

INPUT LINES TO awk

When awk scans an input line, it breaks it down into a number of fields. Fields are separated by a space or tab character. Fields are numbered beginning at one, and the dollar symbol ($) is used to represent a field.

For instance, the following line in a file

```
I like money.
```

has three fields. They are

```
$1      I
$2      like
```

```
        $3      money.
```
Field zero ($0) refers to the entire line.

awk scans lines from a file(s) or standard input.

Your first awk program

Consider the following simple awk program.
```
        { print $0 }
```

There is no pattern to match, only an action expressed. This means that for every line encountered, perform the action.

The action *prints* field 0 (the entire line).

Using a text editor, create a file called *myawk1* and place the above statement in it. Save the file and return to the Unix shell prompt.

Running an awk program

To run the above program, **type** following command
```
        awk   -f myawk1  /etc/group
```

awk interprets the actions specified in the program file *myawk1*, and applies this to each line read from the file */etc/group*. The effect is to print out each input line read from the file, in effect, displaying the file on the screen (same as the Unix command *cat*).

Searching for a string within an input line

To search for an occurrence of a string in an input line, specify it as a pattern and enclose it using a forward slash symbol. In the example below, it searches each input line for the string *brian*, and the action prints the entire line.
```
        /brian/   { print  $0 }
```

Edit *myawk1* and change the search string to your username. Run the program on the files */etc/group* and */etc/passwd*
```
        awk  -fmyawk1  /etc/group

        awk  -fmyawk1  /etc/passwd
```

Compared to the previous example where there was no pattern specified, what is the difference in the output of this program.
```
        ..........................................................
..............................
```

. . . . . .

. . . . . .

Type the following command. This runs the program *who* and sends its output of who is logged on the system to the awk program which scans each line for the search string. It will thus list out a line containing your login name, terminal number and login date/time.

```
who  |  awk -f myawk1
```

Change the contents of *myawk1* to read (replace the search string with your login name)

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

What do you expect the output of the program to be? (what fields will it print out?)

. . . . . .

. . . . . .

. . . . . .

Now type the command

```
who | awk -f myawk1
```

What happened? How is the output different than before.

. . . . . .

. . . . . .

. . . . . .

Using awk programs with form files

awk programs are particularly suited to generating reports or forms. In the following examples, we shall use the following textual data as the input file. The file is available from your tutor, can be typed in using a UNIX editor, or is available on the ftp host *brian.cit.ac.nz*, in the *OS202* subdirectory as the file *awktext*. A heading has been provided here for clarity, there is no header in the data file.

Type	**Memory (Kb)**	**Location**	**Serial #**	**HD Size (Mb)**
XT	640	D402	MG0010	0
386	2048	D403	MG0011	100
486	4096	D404	MG0012	270
386	8192	A423	CC0177	400
486	8192	A424	CC0182	670
286	4096	A423	CC0183	100
286	4096	A425	CC0184	80

```
Mac     4096            B407            EE1027          80
Apple   4096            B406            EE1028          40
68020   2048            B406            EE1029          80
68030   2048            B410            EE1030          100
$unix   16636           A405            CC0185          660
"trs80" 64              Z101            EL0020          0
```

In addition, all examples (awk program files myawk*nn*) are available from your tutor or by **ftp** from the ftp host *brian.cit.ac.nz* in the *OS202* subdirectory (username=*guest*, password=*os2*). A public domain MSDOS awk program (*awk.exe*) is also located in this subdirectory.

Simple Pattern Selection

This involves specifying a pattern to match for each input line scanned. The following awk program (myawk2) compares field one ($1) and if the field matches the string "386", the specific action is performed (the entire line is printed).

```
$1 == "386"  { print $0 }
```

Note: The == symbol represents an equality test, thus in the above pattern, it compares the string of field one against the constant string "386", and performs the action if it matches.

Create the program

```
$ cat  -  >   myawk2
$1 == "386"  { print $0 }
< ctrl-d>
$
```

Note: < ctrl-d> is a keypress to terminate input to the shell. Hold down the ctrl key and then press d. User input is shown in bold type.

Run The Program

```
$ awk  -f  myawk2   awktext
```

Sample Program Output

```
386     2048            D403            MG0011    100
386     8192            A423            CC0177    400
```

The program prints out all input lines where the computer type is a "386".

Write an awk program which prints out all input lines where a computer has 4096 Kb of memory. After running the program successfully, enter it in the space provided below.

```
.....
.....
```

Using Comments In awk Programs

Comments begin with the hash (#) symbol and continue till the end of the line. The awk program below adds a comment to a previous awk program shown earlier

```
        #myawk3, same as myawk2 but has a comment in it
        $1 == "386"  { print $0 }
```

Comments can be placed anywhere on the line. The example below shows the comment placed after the action.

```
        $1 == "386"  { print $0 }   # print all records where the
computer is a 386
```

Remember that the comment ends at the end of the line. The following program is thus wrong, as the closing brace of the action is treated as part of the comment.

```
        $1 == "386  { print $0    #print out all records  }
```

Relational Expressions

We have already seen the equality test. Detailed below are the other relational operators used in comparing expressions.

```
        <       less than
        < =     less than or equal to
        ==      equal to
        !=      not equal
        > =     greater than or equal to
        >       greater than
        ~       matches
        !~      does not match
```

Some Examples Of Using Relational Operators

```
        # myawk4, an awk program to display all input lines for
computers
        # with less than 1024 Kb of memory
        $2 <  1024  { print $0 }
```

myawk4 Program Output

```
        XT      640             D402            MG0010  0
        "trs80" 64              Z101            EL0020  0
```

```
        ============================================================
======
        # myawk5
        # an awk program to print the location/serial number of 486
computers
        $1 == "486"  { print $3, $4 }
```

myawk5 Program Output

```
        D404  MG0012
        A424  CC0182
```

```
        ============================================================
======
        # myawk6
```

```
        # an awk program to print out all computers belonging to
management.
        /MG/  { print $0 }
```

myawk6 Program Output

```
XT      640             D402            MG0010  0
386     2048            D403            MG0011  100
486     4096            D404            MG0012  270
```

The awk program *myawk6* scans each input line searching for the occurrence of the string **MG**. When found, the action prints out the line. The problem with this is it might be possible for the string **MG** to occur in another field, but the serial number indicate that it belongs to another department.

What is necessary is a means of matching only a specific field. To apply a search to a specific field, the match (~) symbol is used. The modified awk program shown below searches field 4 for the string **MG**.

```
        # myawk6A
        # improved awk program, print out all computers belonging to
management.
        $4 ~ /MG/  { print $0 }
```

myawk6a Program Output

```
XT      640             D402            MG0010  0
386     2048            D403            MG0011  100
486     4096            D404            MG0012  270
```

What do the following examples do?

```
        $2 != "4096"  { print $0 }
```

```
        ..........................................................
.....................
```

```
        ..........................................................
.....................
```

```
        $5 >  100    { print $4 }
```

```
        ..........................................................
.....................
```

```
        ..........................................................
.....................
```

```
        $4 !~ /CC/    { print $0 }
```

```
        ..........................................................
.....................
```

. . . . . .

Write an awk program to display the location of all computers belonging to the computer centre (code CC). Test the program, and after running the program successfully, enter the program in the space provided below.

. . . . . .

Making the output a bit more meaningful

In all the previous examples, the output of the awk program has been either the entire line or fields within the line. Lets add some text to make the output more meaningful. Consider the following awk program,

```
# myawk7
# list computers located in D block, type and location
$3 ~ /D/  { print "Location = ", $3, "  type = ", $1 }
```

myawk7 Program Output

```
Location =   D402  type =   XT
Location =   D403  type =   386
Location =   D404  type =   486
```

Text And Formatted Output Using printf

We shall tidy the output information by using a built in function of awk called *printf*. C programmers will have no difficulty using this, as it operates the same way as in the C programming language.

Printing A Text String

Lets examine how to print out some simple text. Consider the following statement,

```
printf( "Location : " );
```

The *printf* statement is terminated by a semi-colon. Brackets are used to enclose the argument, and the text is enclosed using double quotes. Now lets combine it into an actual awk program which displays the location of all 286 type computers.

```
#myawk8
$1 == "286" {  printf( "Location : ");   print $3 }
```

myawk8 Program Output

```
Location : A423
Location : A425
```

Printing A Field Which Is A Text String

Lets now examine how to use printf to display a field which is a text string. In the previous program, a separate statement (print $3) was used to write the room location. In the program below, this will be combined into the *printf* statement also.

```
#myawk9
$1 == "286"  {  printf( "Location is %s\n", $3 );  }
```

myawk9 Program Output

```
Location is A423
Location is A425
```

Note: The symbol \n causes subsequent output to begin on a new line. The symbol %s informs printf to print out a text string, in this case it is the contents of the field $3.

Consider the following awk program which prints the location and serial number of all 286 computers.

```
#myawk10
$1=="286" { printf( "Location = %s, serial # = %s\n", $3, $4
); }
```

myawk10 Program Output

```
Location = A423, serial # = CC0183
Location = A425, serial # = CC0184
```

Write an awk program which lists the serial numbers of all computers belonging to the management school. After running the program successfully, enter it in the space provided below.

```
.....
.....
```

Printing A Numeric Value

Lets now see how to print a numeric value. The symbol %d is used for numeric values. The following awk program lists the location and disk capacity of all 486 computers.

```
#myawk11
$1=="486" { printf("Location = %s, disk = %dKb\n", $3, $5 );
}
```

myawk11 Program Output

```
Location = D404, disk = 270Kb
Location = A424, disk = 670Kb
```

Write an awk program which lists the memory size and serial number of all computers which have a hard disk greater than 80Mb in size. After running the program successfully, enter it in the space provided below.

```
.....
.....
```

Formatting Output

Lets see how to format the output information into specific field widths. A modifier to the %s symbol specifies the size of the field width, which by default is right justified.

```
#myawk12
# formatting the output using a field width
$1=="286" {printf("Location = %10s, disk = %5dKb\n",$3,$5);}

myawk12 Program Output
Location =       A423, disk =   100Kb
Location =       A425, disk =    80Kb
```

10%s specifies to print out field $3 using a field width of 10 characters, and %5d specifies to print out field $5 using a field width of 5 digits.

Summary of printf so far

Below lists the options to printf covered above. [n] indicates optional arguments.

```
%[n]s          print a text string
%[n]d          print a numeric value
\n             print a new-line
```

The BEGIN And END Statements Of An awk Program

The keywords BEGIN and END are used to perform specific actions relative to the programs execution.

```
BEGIN   The action associated with this keyword is executed before the
        first input line is read.

END     The action associated with this keyword is executed after all
        input lines have been processed.
```

The **BEGIN** keyword is normally associated with printing titles and setting default values, whilst the **END** keyword is normally associated with printing totals.

Consider the following awk program, which uses BEGIN to print a title.

```
#myawk13
BEGIN   { print "Location of 286 Computers" }
$1 == "286"  { print $3 }

myawk13 Program Output
Location of 286 Computers
A423
A425
```

Introducing awk Defined Variables

awk programs support a number of pre-defined variables.

```
NR      the current input line number
NF      number of fields in the input line


#myawk14
```

```
        # print the number of computers
        END     { print "There are ", NR, "computers" }

        myawk14 Program Output
        There are  13 computers
```

User Defined Variables In An awk Program

awk programs support the use of variables. Consider an example where we want to count the number of 486 computers we have. Variables are explicitly initialised to zero by awk, so there is no need to assign a value of zero to them.

The following awk program counts the number of 486 computers, and uses the **END** keyword to print out the total after all input lines have been processed. When each input line is read, field one is checked to see if it matches *486*. If it does, the awk variable *computers* is incremented (the symbol ++ means increment by one).

```
        #myawk15
        $1 == "486"  { computers++ }
        END     {  printf("The number of 486 computers is %d\n",
computers);  }

        myawk15 Program Output
        The number of 486 computers is 2

        Note: There is no need to explicitly initialise the variable
'computers' to
        zero. awk does this by default.
```

Write an awk program which counts the number of computers which have 8192Kb or greater amounts of memory, then prints the number found at the end of the program. After running the program successfully, enter it in the space provided below.

```
        ..........................................................
..............


        ..........................................................
..............
```

Write an awk program which sums the disk space of all computers, then prints the total disk space at the end of the program. After running the program successfully, enter it in the space provided below.

```
        ..........................................................
..............


        ..........................................................
..............
```

Introduction to awk

Part 2 of 3

Original Author : Brian Brown, CIT

Revision : 1.0

Date : 05/05/94

ref awk.doc

Regular Expressions

awk provides pattern searching which is more comprehensive than the simple examples outlined previously. These patterns are called *regular expressions*, and are similar to those supported by other UNIX utilities like **grep**.

The simplest regular expression is a string enclosed in slashes,

```
/386/
```

In the above example, any input line containing the string *386* will be printed. To restrict a search to a specific field, the match (or not match) symbol is used. In the following example, field one of the input line is searched for the string *386*.

```
$1  ~  /386/
```

In regular expressions, the following symbols are metacharacters with special meanings.

```
\  ^  $  .  [  ]  *  +  ?  (  )  |

^       matches the first character of a string
$       matches the last character of a string
.       matches a single character of a string
[ ]     defines a set of characters
( )     used for grouping
|       specifies alternatives
```

A group of characters enclosed in brackets matches to any one of the enclosed characters. In the example below (myawk16), field one is matched against either "8" or "6".

```
#myawk16, display all x8x computer types
$1 ~ /[86]/  { print $0 }

myawk16 Program Output
386     2048            D403            MG0011  100
486     4096            D404            MG0012  270
386     8192            A423            CC0177  400
486     8192            A424            CC0182  670
286     4096            A423            CC0183  100
286     4096            A425            CC0184  80
```

```
68020   2048            B406            EE1029  80
68030   2048            B410            EE1030  100
"trs80" 64              Z101            EL0020  0
```

Note: In this example, field one is searched for the character '8' and '6', in any order of occurrence and position.

If the first character after the opening bracket ([) is a caret (^) symbol, this complements the set so that it matches any character NOT IN the set. The following example (myawk17) shows this, matching field one with any character except "2" "3" "4" "8" or "6".

```
#myawk17
# display all which do not contain 2, 3, 4, 6 or 8 in first
field
$1  ~  /[^23468]/  { print $0 }
```

myawk17 Program Output

```
XT      640             D402            MG0010  0
Mac     4096            B407            EE1027  80
Apple   4096            B406            EE1028  40
68020   2048            B406            EE1029  80
68030   2048            B410            EE1030  100
$unix   16636           A405            CC0185  660
"trs80" 64              Z101            EL0020  0
```

Why are the lines containing "68020", "68030" and "trs80" also displayed?

.....
.....

.....
.....

.....
.....

```
#myawk18
# display all lines where field one contains A-Z, a-z
$1  ~  /[a-zA-Z]/  { print $0 }
```

myawk18 Program Output

```
XT      640             D402            MG0010  0
Mac     4096            B407            EE1027  80
Apple   4096            B406            EE1028  40
$unix   16636           A405            CC0185  660
"trs80" 64              Z101            EL0020  0
```

Parentheses are used to group options together, and the vertical bar is used for alternatives. In the following example (myawk19), it searches all input lines for the string "Apple", "Mac", "68020" or "68030".

```
#myawk19
# illustrate multiple searching using alternatives
/(Apple|Mac|68020|68030)/   { print $0 }
```

myawk19 Program Output

```
Mac     4096            B407            EE1027  80
Apple   4096            B406            EE1028  40
```

```
68020   2048            B406            EE1029  80
68030   2048            B410            EE1030  100
```

To use metacharacters as part of a search string, their special meaning must be disabled. This is done by preceding them with the backslash (\) symbol. The following example prints all input lines which contain the string "b$".

```
/b\$/   { print $0 }
```

Write an awk program which prints out all input lines for computers which belong to the school of management (using metacharacters). After running the program successfully, enter it in the space provided below.

```
. . . . . .
```

Write an awk program which prints out all input lines for computer types which begin with a dollar ($) symbol (using metacharacters). After running the program successfully, enter it in the space provided below.

```
. . . . . .
```

Special symbols recognised by awk

In addition to metacharacters, awk recognises the following C programming language escape sequences within regular expressions and strings.

```
\b      backspace
\f      formfeed
\r      carriage return
\t      tab
\"      double quote
```

The following example prints all input lines which contain a tab character

```
/\t/  { print $0 }
```

Consider also the use of string concatenation in pattern matching. The plus (+) symbol concatenates one or more strings in pattern matching. The following awk program (myawk16a) searches for computer types which begin with a dollar ($) symbol and are followed by an alphabetic character (a-z, A-Z), and the last character in the string is the symbol x.

```
#myawk16a
$1  ~  /^\$+[a-zA-Z]+x$/  { print $0 }
```

myawk16 Program Output

```
$unix   16636           A405            CC0185  660
```

Write an awk program which prints out all input lines for computer types which are enclosed in double quotes (using metacharacters). After running the program successfully, enter it in the space provided below.

```
. . . . . .
```

awk interprets any string or variable on the right side of a ~ or !~ as a regular expression. This means the regular expression can be assigned to a variable, and the variable used later in pattern matching. An earlier awk program (myawk17) searched for input lines where field one did not contain the digits 2, 3, 4, 6 or 8.

```
#myawk17
# display all which do not contain 2, 3, 4, 6 or 8 in first
field
$1  ~  /[^23468]/  { print $0 }
```

The awk program below shows how to rewrite this (myawk17) using a variable which is assigned the regular expression.

```
#myawk20
BEGIN  { matchstr = "[^23468]" }
$1  ~  matchstr  { print $0 }
```

myawk20 Program Output

```
XT      640             D402            MG0010  0
Mac     4096            B407            EE1027  80
Apple   4096            B406            EE1028  40
68020   2048            B406            EE1029  80
68030   2048            B410            EE1030  100
$unix   16636           A405            CC0185  660
"trs80" 64              Z101            EL0020  0
```

Consider the following example, which searches for all lines which contain the double quote character (").

```
#myawk21
BEGIN  {  matchstr = "\"" }
$1 ~ matchstr  {  print  $0  }
```

myawk21 Program Output

```
"trs80" 64              Z101            EL0020  0
```

Combining Patterns

Patterns can be combined to provide more powerful and complex matching. The following symbols are used to combine patterns.

```
||              logical or, either pattern can match
&&              logical and, both patterns must match
!               logical not, patterns not matching
```

Lets suppose we want a list of all "486" computers which have more than 250Mb of hard disk space. The following awk pattern uses the logical and to construct the necessary pattern string.

```
#myawk22
$1 == "486"  &&   $5 >  250  { print  $0 }
```

myawk22 Program Output

```
486     4096            D404            MG0012  270
486     8192            A424            CC0182  670
```

Write and awk program which lists all computers of type "286" which have 2Mb or more memory. After running the program successfully, enter it in the space provided below.

```
        ..........................................................
..................
```

Write and awk program which lists all computers of type "286", "386" and "486" which have a hard disk fitted. After running the program successfully, enter it in the space provided below.

```
        ..........................................................
..................
```

---

awk Pattern Ranges

A pattern range is two patterns separated by a comma. The action is performed for each input line between the occurrence of the first and second pattern.

```
        #myawk23
        # demonstrate the use of pattern ranges
        /XT/, /Mac/   { print $0 }

        myawk23 Program Output
        XT      640             D402            MG0010  0
        386     2048            D403            MG0011  100
        486     4096            D404            MG0012  270
        386     8192            A423            CC0177  400
        486     8192            A424            CC0182  670
        286     4096            A423            CC0183  100
        286     4096            A425            CC0184  80
        Mac     4096            B407            EE1027  80
```

The awk program *myawk23* prints out all input lines between the first occurrence of "XT" and the next occurrence of "Mac".

Write an awk program using a pattern range to print out all input lines beginning with the first computer fitted with 8192Kb of memory, up to the next computer which has less than 80Mb of hard disk. After running the program successfully, enter it in the space provided below.

```
        ..........................................................
..................
```

---

awks Built In Variables

awk provides a number of internal variables which it uses to process files. These variables are accessible by the programmer. The following is a summary of awk's built-in variables.

```
        ARGC            number of command-line arguments
        ARGV            array of command-line arguments
        FILENAME        name of current input file
        FNR             record number in current file
        FS              input field separator (default= space and tab
characters)
```

```
NF              number of fields in input line
NR              number of input lines read so far
OFMT            output format for numbers (default=%.6)
OFS             output field separator (default=space)
ORS             output line separator (default=newline)
RS              input line separator (default=newline)
RSTART          index of first character matched by match()
RLENGTH         length of string matched by match()
SUBSEP          subscript separator (default="\034")
```

```
#myawk24
# print the first five input lines of a file, bit like head
FNR == 1, FNR == 5  {  print $0 }
```

myawk24 Program Output

```
XT      640             D402            MG0010  0
386     2048            D403            MG0011  100
486     4096            D404            MG0012  270
386     8192            A423            CC0177  400
486     8192            A424            CC0182  670
```

```
============================================================
======
#myawk25
# print each input line preceded with a line number
# print the heading which includes the name of the file
BEGIN  {  print "File:", FILENAME }
{ print NR, ":\t", $0 }
```

myawk25 Program Output

```
File:  awktext
1 :     XT      640             D402            MG0010  0
2 :     386     2048            D403            MG0011  100
3 :     486     4096            D404            MG0012  270
4 :     386     8192            A423            CC0177  400
5 :     486     8192            A424            CC0182  670
6 :     286     4096            A423            CC0183  100
7 :     286     4096            A425            CC0184  80
8 :     Mac     4096            B407            EE1027  80
9 :     Apple   4096            B406            EE1028  40
10 :    68020   2048            B406            EE1029  80
11 :    68030   2048            B410            EE1030  100
12 :    $unix   16636           A405            CC0185  660
13 :    "trs80" 64              Z101            EL0020  0
```

```
============================================================
======
#myawk26
# demonstrate use of argc and argv parameters
BEGIN  {  print "There are ",ARGC, "parameters on the command
line";
                print "The first argument is ", ARGV[0];
                print "The second argument is ", ARGV[1]
        }
```

myawk26 Program Output

(invoked using *awk -fmyawk26 awktext*)

```
There are  2 parameters on the command line
The first argument is  awk
```

```
        The second argument is  awktext


        ============================================================
======
        #myawk27
        # print out the number of fields in each input line
        {  print "Input line", NR, "has", NF, "fields" }
```

myawk27 Program Output

```
        Input line 1 has 5 fields
        Input line 2 has 5 fields
        Input line 3 has 5 fields
        Input line 4 has 5 fields
        Input line 5 has 5 fields
        Input line 6 has 5 fields
        Input line 7 has 5 fields
        Input line 8 has 5 fields
        Input line 9 has 5 fields
        Input line 10 has 5 fields
        Input line 11 has 5 fields
        Input line 12 has 5 fields
        Input line 13 has 5 fields
```

Using the **BEGIN** statement, it is often desirable to change both **FS** (the symbol used to separate fields) and **RS** (the symbol used to separate input lines). The following text file (*awktext2*) is used for the program *myawk28*. The test file separates each field using a dollar symbol ($), and each input line by a carat symbol (^). The program reads the file and prints out the username and password for each users record. A heading is shown only for clarity.

awktext2 data format

```
        (username$address$password$privledge$downloadlimit$protocol^)
        Joe Bloggs$767 Main Rd Tawa$smidgy$clerk$500$zmodem^Sam
Blue$1023
        Kent Drive Porirua$yougessedit$normal$100$xmodem^Bobby
Williams$96
        Banana Grove$mymum$sysop$3000$zmodem^


        #myawk28
        # a program which shows use of FS and RS, scans awktext2
        BEGIN  { FS = "\$"; RS = "\^" }
        { print "User = ", $1, " Password:", $3 }
```

myawk28 Program Output

```
        User =  Joe Bloggs  Password: smidgy
        User =  Sam Blue  Password: yougessedit
        User =  Bobby Williams  Password: mymum
        User =    Password:
```

Write an awk program which works with the text file *awktext2*. The program is to print out all names of users who have a privilege level of "sysop" or "clerk". After running the program successfully, enter it in the space provided below.

```
        ..........................................................
.................
```

```
        ..........................................................
..................

        ..........................................................
..................
```

---

awks Assignment Operators

The following is a summary of awk's assignment operators.

```
        +               add
        -               subtract
        *               multiply
        /               divide
        ++              increment
        --              decrement
        %               modulus
        ^               exponential
        +=              plus equals
        -=              minus equals
        *=              multiply equals
        /=              divide equals
        %=              modulus equals
        ^=              exponential equals
```

Now some examples,

```
        sum = sum + 3           # same as sum += 3
        sum = x / y
        n++                     # same as n = n + 1
```

The following awk program displays the average installed memory capacity for an IBM type computer (XT - 486). Note the use of **%f** within the *printf* statement to print out the result in floating point format. The use of **.2** between the % and f symbols specify two decimal places.

```
        #myawk29
        /(XT|286|386|486)/   { computers++,  ram += $2 }
        END  { avgmem = ram / computers;
                       printf(" The average memory per PC = %.2f",
avgmem )
                    }
```

myawk29 Program Output

```
        The average memory per PC = 4480.00
```

Write an awk program to print out all the total cost (to the nearest cent) of disk space for computers belonging to the school of management. Assume that disk space has been costed at $10.20 per megabtye. After running the program successfully, enter it in the space provided below.

```
        ..........................................................
..................

        ..........................................................
..................
```

. . . . . .

Write an awk program to print out the percentage (to one decimal place) of computers which have 2048Kb or less of memory. After running the program successfully, enter it in the space provided below.

. . . . . .

. . . . . .

. . . . . .

---

awks Built In Arithmetic Operators and Functions

The following is a summary of awk's built-in arithmetic operators and functions. All operations are done in floating point format.

```
atan2(y,x)     arctangent of y/x in radians
cos(x)         cosine of x, with x in radians
exp(x)         exponential function of x
int(x)         integer part of x truncated towards 0
log(x)         natural logarithm of x
rand()         random number between 0 and 1
sin(x)         sine of x, with x in radians
sqrt(x)        square root of x
srand(x)       x is new seed for rand()
```

Consider the following awk program (myawk30) which prints the square root of an inputted value. This program also shows interactive use, by entering the file that awk processes directly from the keyboard. If no data file is specified (as in the example below, awk reads from the keyboard).

```
#myawk30, to print the square root of a number
{  print  sqrt( $1 ) }
```

Running myawk30

```
awk  -fmyawk30
```

myawk30 Sample Program Output (user entry shown in bold)

```
2
1.41421
3
1.73205
4
2
```

Note: The user pressed CTRL-D (F6 for MSDOS) to signify the end of data input.

Write an awk program to calculate and print out (to three decimal places) the natural logarithm of a value entered from the keyboard. After running the program successfully, enter it (and the command used to invoke it) in the space provided.

```
command:
        ..........................................................
.................

program:
        ..........................................................
.................
```

---

awks Built In String Functions

The following is a summary of awk's built-in string functions. An awk string is created by enclosing characters within quotes ("). A string can contain C language escape sequences. The following awk string contains the escape sequence for a new-line character.

```
        "hello\n"


        gsub(r,s)       substitutes s for r globally in current input
line, returns the
                        number of substitutions
        gsub(r,s,t)     substitutes s for r in t globally, returns
number of substitutions
        index(s,t)      returns position of string t in s, 0 if not
present
        length(s)       returns length of s
        match(s,r)      returns position in s where r occurs, 0 if not
present
        split(s,a)      splits s into array a on FS, returns number of
fields
        split(s,a,r)    splits s into array a on r, returns number of
fields
        sprintf(fmt, expr-list)       returns expr-list formatted
according to format string
                        specified by fmt
        sub(r,s)        substitutes s for first r in current input
line, returns number of
                        substitutions
        sub(r,s,t)      substitutes s for first r in t, returns number
of substitutions
        substr(s,p)     returns suffix s starting at position p
        substr(s,p,n)   returns substring of s length n starting at
position p
```

The following awk program (myawk31) uses the string function **gsub** to replace each occurrence of *286* with the string *AT*.

```
        #myawk31
        { gsub( /286/, "AT" ); print $0 }

        myawk31 Program Output
        XT      640             D402            MG0010  0
```

```
386     2048            D403            MG0011  100
486     4096            D404            MG0012  270
386     8192            A423            CC0177  400
486     8192            A424            CC0182  670
AT      4096            A423            CC0183  100
AT      4096            A425            CC0184  80
Mac     4096            B407            EE1027  80
Apple   4096            B406            EE1028  40
68020   2048            B406            EE1029  80
68030   2048            B410            EE1030  100
$unix   16636           A405            CC0185  660
"trs80" 64              Z101            EL0020  0
```

Write an awk program to find and print out the longest computer name (hint: use the *length* function as a pattern). After running the program successfully, enter it in the space provided below.

. . . . . .

. . . . . .

. . . . . .

Introduction to awk

Part 3 of 3

Original Author : Brian Brown, CIT

Revision : 1.0

Date : 05/05/94

ref awk.doc

---

awk Control Flow Statements

awk provides a number of constructs to implement selection and iteration. These are similar to C language constructs.

if (***expression*** **)** ***statement1*** **else** ***statement2***

The *expression* can include the relational operators, the regular expression matching operators, the logical operators and parentheses for grouping.

expression is evaluated first, and if NON-ZERO then *statement1* is executed, otherwise *statement2* is executed.

In the following awk program (myawk32), each input line is scanned and field $5 is compared against the value of the awk user defined variable *disksize* (awk initialises it to 0). When field $5 is greater, it is assigned to *disksize*, and the input line is saved in the other user defined variable *computer*. Note the use of the braces { } to group the program statements as belonging to the **if** statement (same syntax as in the C language).

```
#myawk32
#demonstrate use of if statement, find biggest disk
{    if( disksize <  $5 )
      {
           disksize = $5
           computer = $0
       }
}
END  { print computer }
```

myawk32 Program Output

```
486     8192           A424           CC0182  670
```

Write an awk program to print out only those computers which are type "486" with 4096Kb or more memory. Use an **if** statement to perform this. After running the program successfully, enter it in the space provided below.

```
.....
.....

.....
.....

.....
.....

.....
.....

.....
.....
```

---

while (***expression*** **)** ***statement***

expression is evaluated, and if NON-ZERO then *statement* is executed, then *expression* is re-evaluated. This continues until *expression* evaluates as ZERO, at which time the **while** statement terminates.

```
#myawk33
# a while statement to print out each second field only for
"286" computers
BEGIN  { printf("Type\tLoc\tDisk\n")  }
/286/  {  field = 1
               while(  field < = NF )
               {
                     printf("%s\t", $field )
                     field += 2
                }
                print ""
```

```
                }
```

myawk33 Program Output

```
Type    Loc     Disk
286     A423    100
286     A425    80
```

The following data file (awktext3) contains a list of computers per department in an organisation.

```
Management              22      Electronics     46      Engineering
        12
Health_Science          5       Tourism         20      Registry
        18
Computing_Centre        300     Library         4       Halls
        2
```

Write an awk program to print out the total number of computers held by the organisation using the data file *awktext3*. Use a **while** statement to perform this (Hint: look at *myawk33*). After running the program successfully, enter it in the space provided below.

```
        ..........................................................
..................

        ..........................................................
..................

        ..........................................................
..................

        ..........................................................
..................
```

---

for (*expression*1; *expression*; *expression*2) *statement*

The for statement provides repetition of a statement. *expression1* is executed first, and is normally used to initialise variables used within the for loop. *expression* is a re-evaluation which determines whether the loop should continue. *expression2* is performed at the end of each iteration of the loop, before the re-evaluation test is performed.

1. *expression1*
2. *expression* is evaluated. If non-zero got step 3 else exit
3. *statement* is executed
4. *expression2* is executed
5. goto step 2

Consider the following awk program (*myawk34*) which is the same as *myawk33* shown earlier.

```
        #myawk34
        # a for statement to print out each second field only for
"286" computers
        BEGIN  { printf("Type\tLoc\tDisk\n")  }
        /286/  {  for( field = 1; field < = NF; field += 2)
printf("%s\t", $field )
                        print ""
                  }
```

myawk34 Program Output

```
Type    Loc     Disk
```

```
286     A423    100
286     A425    80
```

Write an awk program to print out the total number of computers held by the organisation using the data file *awktext3*. Use a **for** statement to perform this (Hint: look at your solution using the while statement previously). After running the program successfully, enter it in the space provided below.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

---

do *statement* while(*expression*)

The *statement* is executed repeatedly until the value of expression is ZERO. *statement* is executed at least once.

```
#myawk35
# print out every second field for "286" computers
BEGIN { field = 1 }
$1 == "286" { do {
                printf("%s\t", $field)
                field += 2
        }  while( field < = NF )
}
```

myawk35 Program Output

```
286     A423    100
```

---

break, continue, next, exit

The **break** statement causes an immediate exit from within a **while** or **for** loop.

The **continue** statement causes the next iteration of a loop.

The **next** statement skips to the next input line then re-starts from the first pattern-action statement.

The **exit** statement causes the program to branch to the END statement (if one exists), else it exits the program.

```
#myawk36
#print out computer types  "286" using a next statement
  { while( $1 != "286" ) next;  print $0 }
```

myawk36 Program Output

```
286     4096            A423            CC0183  100
286     4096            A425            CC0184  80
```

Arrays in awk programs

awk provides single dimensioned arrays. Arrays need not be declared, they are created in the same manner as awk user defined variables.

Elements can be specified as numeric or string values. Consider the following awk program (*myawk37*) which uses arrays to hold the number of "486" computers and the disk space totals for all computers.

```
#myawk37
# diskspace[] holds sum of disk space for all computers
# computers[] holds number of computers of specified type
$1 == "486"  { computers["486"]++ }
$5 >  0  { diskspace[0] += $5 }
END  { print "Number of 486 computers =", computers[486];
            print "Total disk space = ",diskspace[0]
          }
```

myawk37 Program Output

```
Number of 486 computers = 2
Total disk space =  2580
```

Note that the previous program (*myawk37*) uses TWO pattern action statements for each input line. The first pattern action statement handles the number of "486" type computers, whilst the second handles the total disk space for all computer types.

Write an awk program to print out the total disk space for computer types "286", "386" and "486". Use arrays to hold the disk space totals. After running the program successfully, enter it in the space provided below.

```
       ..........................................................
..............

       ..........................................................
..............

       ..........................................................
..............

       ..........................................................
..............

       ..........................................................
..............

       ..........................................................
..............
```

Consider the following awk program (*myawk38*) which uses the **in** statement associated with processing areas. The program .....

```
#myawk38
{ computers[$1]++ }
END  {  for ( name in computers )
```

```
                        print  "The number of ",name,"computers
is",computers[name]
                }
```

myawk38 Program Output

```
The number of  "trs80" computers is 1
The number of  $unix computers is 1
The number of  286 computers is 2
The number of  386 computers is 2
The number of  486 computers is 2
The number of  68020 computers is 1
The number of  68030 computers is 1
The number of  Apple computers is 1
The number of  Mac computers is 1
The number of  XT computers is 1
```

---

awk User Defined Functions

awk supports user defined functions. The syntax is

```
function   name( argument-list )  {
        statements
}
```

The definition of a function can occur anywhere a pattern-action statement can. *argument-list* is a list of variable names separated by commas. There must be NO space between the function name and the left bracket of the argument-list.

The **return** statement is used to return a value by the function.

Consider the following awk program (*myawk39*) which calculates the factorial of an inputted number.

```
#myawk39
function factorial( n ) {
    if( n < = 1 )  return 1
    else  return  n *  factorial( n - 1)
}
{  print "the factorial of ", $1, "is ", factorial($1) }
```

Sample myawk39 Program Output (awk -fmyawk39)

```
10
the factorial of  10 is  3628800
3
the factorial of  3 is  6
1
the factorial of  1 is  1
4
the factorial of  4 is  24
```

---

awk Output

The statements **print** and **printf** are used in awk programs to generate output. awk uses two variables, **OFS** (output field separator) and **ORS** (output record separator) to delineate fields and output lines. These can be changed at any time.

The special characters used in printf, which follow the % symbol, are,

```
c       single character
d       decimal integer
e       double number, scientific notation
f       floating point number
g       use e or f, whichever is shortest
o       octal
s       string
x       hexadecimal
%       the % symbol
```

The default output format is **%.6g** and is changed by assigning a new value to **OFMT**.

---

awk Output To Files

awks output generated by **print** and **printf** can be redirected to a file by using the redirection symbols > (create/write) and > > (append). The names of files MUST be in quotes.

```
#myawk40
# demonstrates sending output to a file
$1 == "486"  { print "Type=",$1, "Location=",$3 >
"comp486.dat"
```

Sample output contained in 'comp486.dat'

```
Type= 486 Location= D404
Type= 486 Location= A424
```

Write an awk program to print out the total disk space for computer types "286", "386" and "486". Use arrays to hold the disk space totals. The output will be stored in the file 'dspace.dat'. After running the program successfully, enter it in the space provided below.

```
.....
.....

.....
.....

.....
.....

.....
.....

.....
.....

.....
.....
```

---

awk Output To Pipes

The output of awk programs can be piped into a UNIX command. The statement

```
print "    ",$1  |  "sort"
```

causes the output of the **print** command to be piped to the UNIX **sort** command.

Write an awk program to print out a sorted list of all "286", "386" and "486" computers sorted according to disk size. After running the program successfully, enter it in the space provided below.

. . . . . .

. . . . . .

. . . . . .

---

awk Input

Data Files

We have seen TWO methods to give file input to an awk program. The first specified the filename on the command line, the other left it blank, and awk read from the keyboard (examples were *myawk30* and *myawk39*).

Program Files

We have used the **-f** parameter to specify the file containing the awk program. awk programs can also be specified on the command-line enclosed in single quotes, as the following example shows.

```
awk  '/286/  {print $0 }'  awktext
```

Note: For MSDOS systems, a double quote must be used to enclose the awk program when specified on the command line.

The getline function

awk provides the function **getline** to read input from the current input file or from a file or pipe.

getline reads the next input line, splitting it into fields according to the settings of NF, NR and FNR. It returns 1 for success, 0 for end-of-file, and -1 on error.

The statement

```
getline data
```

reads the next input line into the user defined variable *data*. No splitting of fields is done and NF is not set.

The statement

```
getline < "temp.dat"
```

reads the next input line from the file "temp.dat", field splitting is performed, and NF is set.

The statement

```
getline data < "temp.dat"
```

reads the next input line from the file "temp.dat" into the user defined variable *data*, no field splitting is done, and NF, NR and FNR are not altered.

Consider the following example, which pipes the output of the UNIX command *who* into *getline*. Each time through the *while* loop, another line is read from *who*, and the user defined variable *users* is incremented. The program counts the number of users on the host system.

```
while ( "who" | getline )
    users++
```

Write an awk program to list all details of type "286" computers. Prefix the list with the current date (Hint: see the previous example, and the UNIX command *date*). After running the program successfully, enter it in the space provided below.

```
. . . . . .
. . . . . .
```

---

awk Summary

The following is a summary of the most common awk statements and features.

Command Line

```
awk  program  filenames
awk  -f  program-file  filenames
awk -Fs
(sets field separator to string s, -Ft sets separator to tab)
```

Patterns

```
BEGIN
END
/regular expression/
relational expression
pattern && pattern
pattern || pattern
(pattern)
!pattern
pattern, pattern
```

Control Flow Statements

```
        if ( expr)  statement [ else statement]
        if ( subscript  in  array)  statement [ else  statement]
        while ( expr)  statement
        for ( expr ;  expr ;  expr )  statement
        for (  var  in  array )  statement
        do statement  while ( expr)
        break
        continue
        next
        exit [ expr]
        return [ expr]
```

Input Output

```
        close(  filename )              close file
        getline                         set $0 form next input line,
set NF, NR, FNR
        getline <  file                 set $0 from next input line of
file, set NF
        getline var                     set var from next input line,
net NR, FNR
        getline var <  file             set var from next input line of
file
        print                           print current input line
        print expr-list                 print expressions
        print expr-list >  file                 print expressions to
file
        printf fmt, expr-list           format and print
        printf fmt, expr-list  >  file format and print to file
        system( cmd-line )              execute command cmd-line,
return status
```

In **print** and **printf** above, > > appends to a *file*, and the | *command* writes to
a pipe. Similarly, *command* | **getline** pipes into **getline**. The function **getline** returns
0 on the end of a file, -1 on an error.

Functions

```
        func  name(  parameter list ) {  statement }
        function  name (  parameter list ) {  statement }
        function-name ( expr, expr, ... )
```

String Functions

```
        gsub(r,s,t)     substitutes s for r in t globally, returns
number of  substitutions
        index(s,t)      returns position of string t in s, 0 if not
present
        length(s)       returns length of s
        match(s,r)      returns position in s where r occurs, 0 if not
present
```

split(*s,a,r*) splits *s* into array *a* on *r*, returns number of fields

sprintf(*fmt, expr-list*) returns *expr-list* formatted according to format string specified by *fmt*

sub(*r,s,t*) substitutes *s* for first *r* in *t*, returns number of substitutions

substr(*s,p,n*) returns substring of *s* length *n* starting at position *p*

Arithmetic Functions

atan2(*y,x*)	arctangent of *y/x* in radians
cos(*x*)	cosine of *x*, with *x* in radians
exp(*x*)	exponential function of *x*
int(*x*)	integer part of *x* truncated towards 0
log(*x*)	natural logarithm of *x*
rand()	random number between 0 and 1
sin(*x*)	sine of *x*, with *x* in radians
sqrt(*x*)	square root of *x*
srand(*x*)	*x* is new seed for **rand()**

Operators (increasing precedence)

= += -= *= /= %= ^=	assignment
?:	conditional expression
||	logical or
& &	logical and
~ !~	regular expression match, negated match
< < = > > = != ==	relationals
blank	string concatenation
+ -	add, subtract
* / %	multiply, divide, modulus
+ - !	unary plus, unary minus, logical negation
^	exponentional
++ --	increment, decrement
$	field

Regular Expressions (increasing precedence)

c	matches no-metacharacter *c*
c	matches literal character *c*
.	matches any character except newline
^	matches beginning of line or string
$	matches end of line or string
[*abc...*]	character class matches any of *abc...*
[^*abc...*]	negated class matches any but *abc...* and newline
r1 | r2	matches either r1 or r2
r1r2	concatenation: matches r1, then r2
r+	matches one or more r's
r*	matches zero or more r's
r?	matches zeor or more r's
(r)	grouping: matches r

Built-In Variables

```
        ARGC            number of command-line arguments
        ARGV            array of command-line arguments (0..ARGC-1fR)
        FILENAME        name of current input file
        FNR             input line number number in current file
        FS              input field separator (default blank)
        NF              number of fields in input line
        NR              number of input lines read so far
        OFMT            output format for numbers (default=%.6g)
        OFS             output field separator (default=space)
        ORS             output line separator (default=newline)
        RS              input line separator (default=newline)
        RSTART          index of first character matched by match()
        RLENGTH         length of string matched by match()
        SUBSEP          subscript separator (default=\034")
```

Limits

```
Each implementation of awk imposes some limits. Below are typical
limits

        100 fields
        2500 characters per input line
        2500 characters per output line
        1024 characters per individual field
        1024 characters per printf string
        400 characters maximum quoted string
        400 characters in character class
        15 open files
        1 pipe
```

Converting files between MSDOS and UNIX format

MSDOS uses a CR and LF to separate each line of a file. The carriage return character appears as a ^M symbol in the editor *vi*. In addition, some MSDOS editors mark the end of a file using the CTRL-Z character.

UNIX uses a LF to separate each line of a file. There is no end of file character.

SCO UNIX provides a mechanism for converting between MSDOS and UNIX file formats.

dtox

```
        converts a MSDOS file to UNIX format (does not strip the end-
of-file
        character).
```

dtox awktext > awktext.unx

```
        It may also be necessary to load the file into an editor and
remove the end-
        of-file character.
```

xtod

```
        converts a UNIX file to MSDOS format.
```

```
        xtod  awktext.unx >  awktext
```