LEARNING awk

#awk

Table of Contents

About

Chapter 1: Getting started with awk

Remarks

The name AWK comes from the last initials of its creators Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan.

Resources

- The Illumos AWK man-page
- The Plan9 AWK man-page
- The GNU AWK Users Guide
- The AWK Programming Language

Versions

Name	Initial Version	Version	Release Date
POSIX awk	1992	IEEE Std 1003.1, 2013 Edition	2013-04-19
One True Awk or nawk or BWK awk	198X	-	2012-12-20
GNU awk or gawk	1986	4.1.3	2015-05-19

Examples

Hello world

The Hello world example is as simple as:

```
awk 'BEGIN {print "Hello world"}'
```

The most basic `awk` program consists of a true value (typically `1`) and makes `awk` echo its input:

```
$ date | awk '1'
Mon Jul 25 11:12:05 CEST 2016
```

Since "hello world" is also a true value, you could also say:

```
$ date | awk '"hello world"'
Mon Jul 25 11:12:05 CEST 2016
```

However, your intention becomes much clearer if you write

```
$ date | awk '{print}'
Mon Jul 25 11:12:05 CEST 2016
```

instead.

How to run AWK programs

If the program is short, you can include it in the command that runs awk:

```
awk -F: '{print $1, $2}' /etc/passwd
```

In this example, using command line switch `-F:` we advise awk to use : as input fields delimiter. Is is the same like

```
awk 'BEGIN{FS=":"}{print $1,$2}' file
```

Alternativelly, we can save the whole awk code in an awk file and call this awk programm like this:

```
awk -f 'program.awk' input-file1 input-file2 ...
```

program.awk can be whatever multiline program, i.e :

```
# file print_fields.awk
BEGIN {print "this is a header"; FS=":"}
{print $1, $2}
END {print "that was it"}
```

And then run it with:

```
awk -f print_fields.awk /etc/passwd   #-f advises awk which program file to load
```

Or More generally:

```
awk -f program-file input-file1 input-file2 ...
```

The advantage of having the program in a seperate file is that you can write the programm with correct identation to make sense, you can include comments with # , etc

AWK by examples

AWK is string manipulation language, used largely in UNIX systems. The idea behind AWK was to create a versatile language to use when working on files, which wasn't too complex to understand.

AWK has some other variants, but the main concept is the same, just with additional features. These other variants are NAWK and GAWK. GAWK contains all of the features of both, whilst NAWK is one step above AWK, if you like.

The most simple way to think of AWK, is to consider that it has 2 main parts. The pattern, and the action.

Probably the most basic example of AWK: (See also: Hello World)

```
BEGIN {print "START"}
      {print        }
END   {print "STOP" }
```

Here, the keywords `BEGIN` and `END` are the pattern, whilst the action is inside the {}. This example would be useless, but it would only take minor changes to actually make this into a useful function.

```
BEGIN {print "File\tAuthor"}
      {print $8, "\t", $3}
END {print " - DONE - "}
```

Here, `\t` represents a Tab character, and is used to even up the output line boundaries. $8 and $3 are similar to the use that is seen in `Shell Scripts`, but instead of using the 3rd and 8th arguments, it uses the 3rd and 8th column of the input line.

So, this example would print: File Author on the top line, whilst the second line is to do with the file paths. $8 is the name of the file, $3 is the owner (When looking at the directory path, this will be more clear). Finally, the bottom line would print, as you would expect - DONE -

Credit for the above example goes to http://www.grymoire.com/Unix/Awk.html

Reference file

`coins.txt` from Greg Goebel:

```
gold     1    1986  USA                 American Eagle
gold     1    1908  Austria-Hungary     Franz Josef 100 Korona
silver  10    1981  USA                 ingot
gold     1    1984  Switzerland         ingot
gold     1    1979  RSA                 Krugerrand
gold     0.5  1981  RSA                 Krugerrand
gold     0.1  1986  PRC                 Panda
silver   1    1986  USA                 Liberty dollar
gold     0.25 1986  USA                 Liberty 5-dollar piece
silver   0.5  1986  USA                 Liberty 50-cent piece
silver   1    1987  USA                 Constitution dollar
gold     0.25 1987  USA                 Constitution 5-dollar piece
gold     1    1988  Canada              Maple Leaf
```

Minimal theory

General awk one-liner:

```
awk <awk program> <file>
```

or:

```
<shell-command> | awk <awk program>
```

`<shell-command>` and `<file>` are addressed as *awk input*.

`<awk program>` is a code following this template (single, not double, quotes):

```
'BEGIN   {<init actions>};
 <cond1> {<program actions>};
 <cond2> {<program actions>};
 ...
 END  {<final actions>}'
```

where:

- `<condX>` condition is most often a regular expression `/re/`, to be matched with awk input lines;
- `<* actions>` are sequence of *statements*, similar to shell commands, equipped with C-like constructs.

`` is processed according to the following rules:

1. `BEGIN ...` and `END ...` are optional and executed before or after processing awk input lines.
2. For each line in the awk input, if condition `<condN>` is meat, then the related `<program actions>` block is executed.
3. `{<program actions>}` defaults to `{print $0}`.

Conditions can be combined with standard logical operators:

```
    /gold/ || /USA/ && !/1986/
```

where `&&` has precedence over `||`;

The`print` statement. `print item1 item2` statement prints items on STDOUT.
Items can be variables (`X`, `$0`), strings ("hello") or numbers.
`item1, item2` are collated with the value of the `OFS` variable;
`item1 item2` *are justapoxed*! Use `item1 " " item2` for spaces or printf for more features.

Variables do not need `$`, i.e.: `print myVar;`
The following special variables are builtin in awk:

- `FS`: acts as field separator to splits awk input lines in fields. I can be a single character, `FS="c"` ; a null string, `FS=""` (then each individual character becomes a separate field); a regular expression without slashes, `FS="re"`; `FS=" "` stands for runs of spaces and tabs and is defaults value.
- `NF`: the number of fields to read;
- `$1`, `$2`, ...: 1st field, 2nd field. etc. of the current input line,
- `$0`: current input line;
- `NR`

: current put line number.

- `OFS`: string to collate fields when printed.
- `ORS`: output record separator, by default a newline.
- `RS`: Input line (record) separator. Defaults to newline. Set as `FS`.
- `IGNORECASE`: affects FS and RS when are regular expression;

Examples

Filter lines by regexp `gold` and count them:

```
# awk 'BEGIN {print "Coins"} /gold/{i++; print $0}  END {print i " lines out of " NR}'
coins.txt
Coins
gold     1    1986  USA                 American Eagle
gold     1    1908  Austria-Hungary     Franz Josef 100 Korona
gold     1    1984  Switzerland         ingot
gold     1    1979  RSA                 Krugerrand
gold     0.5  1981  RSA                 Krugerrand
gold     0.1  1986  PRC                 Panda
gold     0.25 1986  USA                 Liberty 5-dollar piece
gold     0.25 1987  USA                 Constitution 5-dollar piece
gold     1    1988  Canada              Maple Leaf
9 lines out of 13
```

Default `print $0` action and condition based on internal awk variable `NR`:

```
# awk 'BEGIN {print "First 3 coins"} NR<4' coins.txt
First 3 coins
gold     1    1986  USA                 American Eagle
gold     1    1908  Austria-Hungary     Franz Josef 100 Korona
silver  10    1981  USA                 ingot
```

Formatting with C-style `printf`:

```
# awk '{printf ("%s \t %3.2f\n", $1, $2)}' coins.txt
gold     1.00
gold     1.00
silver   10.00
gold     1.00
gold     1.00
gold     0.50
gold     0.10
silver   1.00
gold     0.25
silver   0.50
silver   1.00
gold     0.25
gold     1.00
```

Condition Examples

```
awk 'NR % 6'            # prints all lines except those divisible by 6
awk 'NR > 5'            # prints from line 6 onwards (like tail -n +6, or sed '1,5d')
awk '$2 == "foo"'       # prints lines where the second field is "foo"
awk '$2 ~ /re/'         # prints lines where the 2nd field mateches the regex /re/
awk 'NF >= 6'           # prints lines with 6 or more fields
awk '/foo/ && !/bar/'   # prints lines that match /foo/ but not /bar/
awk '/foo/ || /bar/'    # prints lines that match /foo/ or /bar/ (like grep -e 'foo' -e 'bar')
awk '/foo/,/bar/'       # prints from line matching /foo/ to line matching /bar/, inclusive
awk 'NF'                # prints only nonempty lines (or: removes empty lines, where NF==0)
awk 'NF--'              # removes last field and prints the line
```

By adding an action `{...}` one can print a specific field, rather than the whole line, e.g.:

```
awk '$2 ~ /re/{print $3 " " $4}'
```

prints the third and fourth field of lines where the second field mateches the regex /re/.

Some string functions

`substr()` function:

```
# awk '{print substr($3,3) " " substr($4,1,3)}'
86 USA
08 Aus
81 USA
84 Swi
79 RSA
81 RSA
86 PRC
86 USA
86 USA
86 USA
87 USA
87 USA
88 Can
```

`match(s, r [, arr])` returns the position in `s` where the regex `r` occurs and sets the values of `RSTART` and `RLENGTH`. If the argument `arr` is provided, it returns the array `arr` where elements are set to the matched parenthesized subexpression. The 0'th element matches of `arr` is set to the entire regex match. Also expressions `arr[n, "start"]` and `arr[n, "length"]` provide the starting position and length of each matching substring.

More string functions:

```
sub(/regexp/, "newstring"[, target])
gsub(/regexp/, "newstring"[, target])
toupper("string")
tolower("string")
```

Statements

A simple statement is often any of the following:

```
variable = expression
print [ expression-list ]
printf format [ , expression-list ]
next # skip remaining patterns on this input line
exit # skip the rest of the input
```

If `stat1` and `stat2` are statements, the following are also statements:

```
{stat}

{stat1;  stat2}

{stat1
stat2}

if ( conditional ) statement [ else statement ]
```

The following standard C-like are constructs are statements:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break    # usual C meaning
continue # usual C meaning
```

A C-style loop to print the variable length description element, starting with field 4:

```
# awk '{out=""; for(i=4;i<=NF;i++){out=out" "$i}; print out}' coins.txt
USA American Eagle
Austria-Hungary Franz Josef 100 Korona
USA ingot
Switzerland ingot
RSA Krugerrand
RSA Krugerrand
PRC Panda
USA Liberty dollar
USA Liberty 5-dollar piece
USA Liberty 50-cent piece
USA Constitution dollar
USA Constitution 5-dollar piece
Canada Maple Leaf
```

Note that `i` is initialized to 0.

If conditions and calculations applied to nuneric fields:

```
# awk '/gold/ {if($3<1980) print $0 "$" 425*$2}' coins.txt
gold     1    1908  Austria-Hungary     Franz Josef 100 Korona      $425
gold     1    1979  RSA                 Krugerrand                  $425
```

AWK executable script

```
#!/usr/bin/gawk -f
```

```
# This is a comment
(pattern) {action}
...
```

Passing shell variables

```
# var="hello"
# awk -v x="$var" 'BEGIN {print x}'
hello
```

Read Getting started with awk online: https://riptutorial.com/awk/topic/937/getting-started-with-awk

Chapter 2: Arrays

Examples

Array basics

Creating a new array is slightly confusing, as there is no real identifier for an array in awk. So, an array cannot really be initialised with our AWK code.

An array in awk is associative, meaning that any string or number can be a key. This means that the array is more like a key-value pair dictionary, map etc. On another note, the arrays do not have a maximum size.

Creating an array in AWK is really easy, as you take a variable name, a proper key and assign it to a variable. This means when the following code is executed, we already have created an array called `myArray`:

```
BEGIN {
    myArray["key"] = "value"
}
```

We our not bound to creating arrays in the begin only. Lets say we have the following input stream:

```
A b c
D e f
G h i
```

And execute the following code with this:

```
{
    myOtherArray[$1] = $2 "-" $3
}
# The array will look like this:
# myOtherArray["A"] = "b-c"
# myOtherArray["D"] = "e-f"
# myOtherArray["G"] = "h-i"
```

When an array is filled with key value pairs, one can retrieve the value with the key only. This means that if we use key `"A"` in `myOtherArray` we get `"b-c"`.

```
END {
    print(myOtherArray["A"])
}
```

We also have the option to loop through each key to get each value. Looping through each key of an array is a simple thing to do, however it has on downfall: it is unsorted. The following loop is like a for-each loop, which retrieves the key:

```
END {
    for (key in myOtherArray) {
        print "myOtherArray[\"" key "\"] = " myOtherArray[key]
    }
}
# Outputs (literal strings):
myOtherArray["A"] = "b-c"
myOtherArray["D"] = "e-f"
myOtherArray["G"] = "h-i"
```

Read Arrays online: https://riptutorial.com/awk/topic/7209/arrays

Chapter 3: Built-in functions

Examples

length([String])

Returns the number of characters of the given *String*

Considerations

- If a number is given instead a String, the result will be the length of the String representing the given number. I.e. If we execute `length(12345)` the result will be the same as `length("12345")`, that is *5*
- If no value is given, the result will be the length of the actual row being processed, that is `length($0)`
- It can be used inside a pattern or inside code-blocks.

Examples

Here are a few examples demonstrating how `length()` works

```
$ cat file
AAAAA
BBBB
CCCC
DDDD
EEEE
```

Inside a pattern

Filter all lines with a length bigger than 4 characters

```
$ awk ' length($0) > 4 ' file
AAAAA
```

Inside a code block

Will print the size of the current line

```
$ awk '{ print length($0) }' file
5
4
4
4
```

```
4
```

With no data given

Will print the size of the current line

```
$ awk '{ print length }' file
5
4
4
4
4
```

Will print the size of the current line

```
$ awk '{ print length() }' file
5
4
4
4
4
```

Number given instead of String

Will print the size of the String representing the number

```
$ awk '{ print length(12345) }' file
5
5
5
5
5
```

Fixed String given

Will print the size of the String

```
$ awk '{ print length("12345") }' file
5
5
5
5
5
```

Read Built-in functions online: https://riptutorial.com/awk/topic/4922/built-in-functions

Chapter 4: Built-in Variables

Examples

FS - Field Separator

Used by awk to split each record into multiple fields:

```
echo "a-b-c
d-e-f" | awk 'BEGIN {FS="-"} {print $2}'
```

will result in:

```
b
e
```

The variable `FS` can also be set using the option `-F`:

```
echo "a-b-c
d-e-f" | awk -F '-' '{print $2}'
```

By default, the fields are separated by whitespace (spaces and tabs) and multiple spaces and tabs count as a single separator.

RS - Record Separator

Used by awk to split the input into multiple records. For example:

```
echo "a b c|d e f" | awk 'BEGIN {RS="|"} {print $0}'
```

produces:

```
a b c
d e f
```

By default, the record separator is the newline character.

Similarly: echo "a b c|d e f" | awk 'BEGIN {RS="|"} {print $2}'

produces:

```
b
e
```

OFS - Output Field Separator

Used by awk to separate fields output by the `print` statement. For example:

```
echo "a b c
d e f" | awk 'BEGIN {OFS="-"} {print $2, $3}'
```

produces:

```
b-c
e-f
```

The default value is , a string consisting of a single space.

ORS - Output Record Separator

Used by awk to separate records and is output at the end of every `print` statement. For example:

```
echo "a b c
d e f" | awk 'BEGIN {ORS="|"} {print $2, $3}'
```

produces:

```
b c|e f
```

The default value is `\n` (newline character).

ARGV, ARGC - Array of Command Line Arguments

Command line arguments passed to awk are stored in the internal array `ARGV` of `ARGC` elements. The first element of the array is the program name. For example:

```
awk 'BEGIN {
   for (i = 0; i < ARGC; ++i) {
      printf "ARGV[%d]=\"%s\"\n", i, ARGV[i]
   }
}' arg1 arg2 arg3
```

produces:

```
ARGV[0]="awk"
ARGV[1]="arg1"
ARGV[2]="arg2"
ARGV[3]="arg3"
```

FS - Field Separator

The variable `FS` is used to set the *input field separator*. In `awk`, space and tab act as default field separators. The corresponding field value can be accessed through `$1`, `$2`, `$3`... and so on.

```
awk -F'=' '{print $1}' file
```

- `-F` - command-line option for setting input field separator.

```
awk 'BEGIN { FS="=" } { print $1 }' file
```

OFS - Output Field Separator

This variable is used to set the *output field separator* which is a space by default.

```
awk -F'=' 'BEGIN { OFS=":" } { print $1 }' file
```

Example:

```
$ cat file.csv
col1,col2,col3,col4
col1,col2,col3
col1,col2
col1
col1,col2,col3,col4,col5

$ awk -F',' 'BEGIN { OFS="|" } { $1=$1 } 1' file.csv
col1|col2|col3|col4
col1|col2|col3
col1|col2
col1
col1|col2|col3|col4|col5
```

Assigning `$1` to `$1` in `$1=$1` modifies a field (`$1` in this case) and that results in `awk` rebuilding the record `$0`. Rebuilding the record replaces the delimiters `FS` with `OFS`.

RS - Input Record Separator

This variable is used to set input record separator, by default a newline.

```
awk 'BEGIN{RS=","} {print $0}' file
```

ORS - Output Record Separator

This variable is used to set output record separator, by default a newline.

```
awk 'BEGIN{ORS=","} {print $0}' file
```

NF - Number of Fields

This variable will give you a total number of fields in the current input record.

```
awk -F',' '{print NF}' file.csv
```

Example:

```
$ cat file.csv
col1,col2,col3,col4
col1,col2,col3
col1,col2
col1
col1,col2,col3,col4,col5

$ awk -F',' '{print NF}' file.csv
4
3
2
1
5
```

NR - Total Number of Records

Will provide the total number of records processed in the current `awk` instance.

```
cat > file1
suicidesquad
harley quinn
joker
deadshot

cat > file2
avengers
ironman
captainamerica
hulk

awk '{print NR}' file1 file2
1
2
3
4
5
6
7
8
```

A total on 8 records were processed in the instance.

FNR - Number of Records in File

Provides the total number of records processed by the `awk` instance relative to the files `awk` is processing

```
cat > file1
suicidesquad
harley quinn
joker
deadshot

cat > file2
avengers
ironman
```

```
captainamerica
hulk

awk '{print FNR}' file1 file2
1
2
3
4
1
2
3
4
```

Each file had 4 lines each, so whenever `awk` encountered an `EOF` `FNR` was reset to 0.

NF - Number of Fields

Provides the number of columns or fields in each record (record corresponds to each line). Each line is demarcated by `RS` which defaults to newline.

```
cat > file1
Harley Quinn Loves Joker
Batman Loves Wonder Woman
Superman is not dead
Why is everything I type four fielded!?

awk '{print NF}' file1
4
4
4
7
```

`FS` (somewhere up there) defaults to tab or space. So Harley, Quinn, Loves, Joker are each considered as columns. The case holds for the next two lines, but the last line has 7 space separated words, which means 7 columns.

FNR - The Current Record Number being processed

FNR contains the number of the input file row being processed. In this example you will see awk starting on 1 again when starting to process the second file.

Example with one file

```
$ cat file1
AAAA
BBBB
CCCC
$ awk '{ print FNR }' file1
1
2
3
```

Example with two files

```
$ cat file1
AAAA
BBBB
CCCC
$ cat file2
WWWW
XXXX
YYYY
ZZZZ
$ awk '{ print FNR, FILENAME, $0 }' file1 file2
1 file1 AAAA
2 file1 BBBB
3 file1 CCCC
1 file2 WWWW
2 file2 XXXX
3 file2 YYYY
4 file2 ZZZZ
```

Extended example with two files

`FNR` can be used to detect if awk is processing the first file since `NR==FNR` is true only for the first file. For example, if we want to join records from files `file1` and `file2` on their `FNR`:

```
$ awk 'NR==FNR { a[FNR]=$0; next } (FNR in a) { print FNR, a[FNR], $1 }' file1 file2
1 AAAA WWWW
2 BBBB XXXX
3 CCCC YYYY
```

Record `ZZZZ` from `file2` is missing as `FNR` has different max value for `file1` and `file2` and there is no join for differing `FNR`s.

Read Built-in Variables online: https://riptutorial.com/awk/topic/3227/built-in-variables

Chapter 5: Fields

Examples

Looping trough fields

```
awk '{ for(f=1; f<=NF; f++) { print $f; } }' file
```

Read Fields online: https://riptutorial.com/awk/topic/6245/fields

Chapter 6: Patterns

Examples

Regexp Patterns

Patterns can be specified as regular expressions:

```
/regular expression/ {action}
```

For example:

```
echo "user@hostname.com
not an email" | awk '/[^@]+@.+/ {print}'
```

Produces:

```
user@hostname.com
```

Note that an action consisting only of the `print` statement can be omitted entirely. The above example is equivalent to:

```
echo "user@hostname.com
not an email" | awk '/[^@]+@.+/'
```

Read Patterns online: https://riptutorial.com/awk/topic/3475/patterns

Chapter 7: Patterns and Actions

Examples

Introduction

An `awk` consists of patterns and actions, enclosed in curly brackets, to be taken if a pattern matches. The most basic pattern is the empty pattern, which matches any record. The most basic action is the empty action, which is equivalent to `{ print }`, which is, in turn, equivalent to `{ print $0 }`. If both the pattern and the action are empty, `awk` will simply do nothing.

The following program will simply echo its input, for example:

```
awk '{ print }' /etc/passwd
```

Since `{ print }` is the default action, and since a true value matches any record, that program could be re-written as:

```
awk '1' /etc/passwd
```

The most common type of pattern is probably a regular expression enclosed in slashes. The following program will print all records that contain at least two subsequent occurrences of the letter `o`, for example:

```
awk '/oo+/ { print }' /etc/passwd
```

However, you can use arbitrary expressions as patterns. The following program prints the names (field one) of users in group zero (field four), for example:

```
awk -F: '$4 == 0 { print $1 }' /etc/passwd
```

Instead of matching exactly, you can also match against a regular expression. The following program prints the names of all users in a group with at least one zero in its group id:

```
awk -F: '$4 ~ /0/ { print $1 }' /etc/passwd
```

Filter Lines by length

> This pattern will allow you to filter lines depending on its length

```
$cat file
AAAAA
BBBB
CCCC
DDDD
EEEE
```

```
$awk 'length($0) > 4 { print $0 }' file
AAAAA
$
```

Anyway, the pattern will allow the next code block to be executed, then, as the **default action for AWK is printing the current line** `{print}`, we´ll see the same result when executing this:

```
$awk 'length($0) > 4 ' file
AAAAA
```

Read Patterns and Actions online: https://riptutorial.com/awk/topic/3987/patterns-and-actions

Chapter 8: Row Manipulation

Examples

Extract specific lines from a text file

Suppose we have a file

```
cat -n lorem_ipsum.txt
 1    Lorem Ipsum is simply dummy text of the printing and typesetting industry.
 2    Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an
unknown printer took a galley of type and scrambled it to make a type specimen book.
 3    It has survived not only five centuries, but also the leap into electronic typesetting,
remaining essentially unchanged.
 4    It was popularised in the 1960s with the release of Letraset sheets containing Lorem
Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker
including versions of Lorem Ipsum
```

We want to extract lines 2 and 3 from this file

```
awk 'NR==2,NR==3' lorem_ipsum.txt
```

This will print lines 2 and 3:

```
 2    Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an
unknown printer took a galley of type and scrambled it to make a type specimen book.
 3    It has survived not only five centuries, but also the leap into electronic typesetting,
remaining essentially unchanged.
```

Extract specific column/field from specific line

If you have the following data file

```
cat data.csv
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

maybe you need to read the fourth column of the third line, this would be "24"

```
awk 'NR==3 { print $4 }' data.csv
```

gives

    24

Modifying rows on-the-fly (e.g. to fix Windows line-endings)

If a file may contain Windows or Unix-like line endings (or even a mixture of both) then the intended text replacement may not work as expected.

Sample:

```
$ echo -e 'Entry 1\nEntry 2.1\tEntry 2.2\r\nEntry 3\r\n\r\n' \
> | awk -F'\t' '$1 != "" { print $1 }' \
> | hexdump -c
0000000   E   n   t   r   y       1  \n   E   n   t   r   y       2   .
0000010   1  \n   E   n   t   r   y       3  \r  \n  \r  \n
000001d
```

This can be easily fixed by an additional rule which is inserted at the beginning of the awk script:

```
/\r$/ { $0 = substr($0, 1, length($0) - 1) }
```

Because the action does not end with `next`, the following rules are applied as before.

Sample (with fix of line-endings):

```
$ echo -e 'Entry 1\nEntry 2.1\tEntry 2.2\r\nEntry 3\r\n\r\n' \
> | awk -F'\t' '/\r$/ { $0 = substr($0, 1, length($0) - 1) } $1 != "" { print $1 }' \
> | hexdump -c
0000000   E   n   t   r   y       1  \n   E   n   t   r   y       2   .
0000010   1  \n   E   n   t   r   y       3  \n
000001a
```

Read Row Manipulation online: https://riptutorial.com/awk/topic/3947/row-manipulation

Chapter 9: String manipulation functions

Syntax

- index(big, little)
- length or length()
- length(string)
- match(string, regex)
- split(string, array, separator)
- split(string, array)
- sprintf(format, ...)
- sub(regex, subst, string)
- sub(regex, subst)
- gsub(regex, subst)
- gsub(regex, subst, string)
- substr(string, start, end)
- substr(string, start)
- tolower(string)
- toupper(string)

Parameters

Parameter	Details
big	The string which is scanned for "little".
end	The index at which to end the sub-string.
format	A `printf` format string.
little	The string to scan for in "big".
regex	An Extended-Regular-Expression.
start	The index at which to start the sub-string.
string	A string.
subst	The string to substitute in for the matched portion.

Examples

Converting string to upper case

The function `toupper` will convert a string to upper case (capital letters). For example:

```
BEGIN {
    greeting = "hello"
    loud_greeting = toupper(greeting)
    print loud_greeting
}
```

This code will output "HELLO" when run.

String Concatenation

String concatenation is done simply by writing expressions next to one another without any operator. For example:

```
BEGIN {
   user = "root"
   print "Hello "user "!"
}
```

will print: `Hello root!`

Note that expressions do not have to be separated by whitespace.

Computing a hash of a string

While implementing one of the standard hashing algorithm in **awk** is probably a tedious task, defining a *hash* function that can be used as a handle to text documents is much more tractable. A practical situation where such a function is useful is to assign short ids to items given their description, for instance test cases, so that the short id can be given as reference to the item by the user instead of supplying its long description.

The *hash* function needs to convert characters to numeric codes, which is accomplished by using a lookup table initialised at the beginning of the script. The *hash* function is then computed using modular arithmetic transformations, a very classical approach to the computation of hashes.

For demonstration purposes, we add a rule to decorate input lines with their hash, but this rule is not needed to use the function:

```
BEGIN{
  for(n=0;n<256;n++) {
    ord[sprintf("%c",n)] = n
  }
}

function hash(text, _prime, _modulo, _ax, _chars, _i)
{
  _prime = 104729;
  _modulo = 1048576;
  _ax = 0;
  split(text, _chars, "");
  for (_i=1; _i <= length(text); _i++) {
    _ax = (_ax * _prime + ord[_chars[_i]]) % _modulo;
  };
  return sprintf("%05x", _ax)
}

# Rule to demonstrate the function
#  These comments and the following line are not relevant
#  to the definition of the hash function but illustrate
#  its use.

{ printf("%s|%s\n", hash($0), $0) }
```

We save the program above to the file `hash.awk` and demonstrate it on a short list of classical english book titles:

```
awk -f hash.awk <<EOF
Wuthering Heights
Jane Eyre
Pride and Prejudice
The Mayor of Casterbridge
The Great Gatsby
David Copperfield
Great Expectations
The Return of the Soldier
Alice's Adventures in Wonderland
Animal Farm
EOF
```

The output is

```
6d6b1|Wuthering Heights
7539b|Jane Eyre
d8fba|Pride and Prejudice
fae95|The Mayor of Casterbridge
17fae|The Great Gatsby
c0005|David Copperfield
7492a|Great Expectations
12871|The Return of the Soldier
c3ab6|Alice's Adventures in Wonderland
46dc0|Animal Farm
```

When applied on each of the 6948 non-blank lines of my favourite novel this hash function does not generate any collision.

Convert string to lower case

AWK often used for manipulating entire files containing a list of strings. Let's say file *awk_test_file.txt* contains:

```
First String
Second String
Third String
```

To convert all the strings to lower case execute:

```
awk '{ print tolower($0) }' awk_test_file.txt
```

This will result:

```
first string
second string
third string
```

String text substitution

SUB function allows to substitute text inside awk

> sub(regexp, replacement, target)

where regexp could be a full regular expression

```
$ cat file
AAAAA
BBBB
CCCC
DDDD
EEEE
FFFF
GGGG
$ awk '{sub("AAA","XXX", $0); print}' file
XXXAA
BBBB
CCCC
DDDD
EEEE
FFFF
GGGG
```

Substring extraction

`GNU` awk supports a sub-string extraction function to return a fixed length character sequence from a main string. The syntax is

```
*substr(string, start [, length ])*
```

where, `string` is source string and `start` marks the start of the sub-string position you want the extraction to be done for an optional length `length` characters. If the length is not specified, the extraction is done up to the end of the string.

The first character of the string is treated as character number one.

```
awk '
BEGIN {
    testString = "MyTESTstring"
    substring  =  substr(testString, 3, 4)    # Start at character 3 for a length of 4
characters
    print substring
}'
```

will output the sub-string `TEST`.

```
awk '
BEGIN {
    testString = "MyTESTstring"
    substring  =  substr(testString, 3)     # Start at character 3 till end of the string
    print substring
}'
```

this extracts the sub-string from character position 3 to end of the whole string, returning `TESTstring`

Note:-

- If `start` is given a negative value, `GNU` awk prints the whole string and if `length` is given a non-zero value `GNU` awk behavior returns a `null` string and the behavior varies among different implementations of `awk`.

Read String manipulation functions online: https://riptutorial.com/awk/topic/2284/string-manipulation-functions

Chapter 10: Two-file processing

Examples

Check matching fields in two files

Given these two CSV files:

```
$ cat file1
1,line1
2,line2
3,line3
4,line4
$ cat file2
1,line3
2,line4
3,line5
4,line6
```

To print those lines in `file2` whose second column occurs also in the first file we can say:

```
$ awk -F, 'FNR==NR {lines[$2]; next} $2 in lines' file1 file2
1,line3
2,line4
```

Here, `lines[]` holds an array that gets populated when reading `file1` with the contents of the second field of each line.

Then, the condition `$2 in lines` checks, for every line in `file2`, if the 2nd field exists in the array. If so, the condition is True and `awk` performs its default action, consisting in printing the full line.

If just one field was needed to be printed, then this could be the expression:

```
$ awk -F, 'FNR==NR {lines[$2]; next} $2 in lines {print $1}' file1 file2
1
2
```

Print awk variables when reading two files

I hope this example will help everyone to understand how awk internal variables like NR, FNR etc change when awk is processing two files.

```
awk '{print "NR:",NR,"FNR:",FNR,"fname:",FILENAME,"Field1:",$1}' file1 file2
NR: 1 FNR: 1 fname: file1 Field1: f1d1
NR: 2 FNR: 2 fname: file1 Field1: f1d5
NR: 3 FNR: 3 fname: file1 Field1: f1d9
NR: 4 FNR: 1 fname: file2 Field1: f2d1
NR: 5 FNR: 2 fname: file2 Field1: f2d5
NR: 6 FNR: 3 fname: file2 Field1: f2d9
```

Where file1 and file2 look like:

```
$ cat file1
f1d1 f1d2 f1d3 f1d4

$ cat file2
f2d1 f2d2 f2d3 f2d4
```

Notice how `NR` value keeps increasing among all files, while `FNR` resets on each file. This is why the expression `NR==FNR` always refer to the first file fed to awk, since only in first file is possible to have `NR` equal to `FNR`.

Read Two-file processing online: https://riptutorial.com/awk/topic/4486/two-file-processing

Chapter 11: Useful one-liners - calculating average from a CSV etc

Examples

Robust processing tabular data (CSV et al.)

Processing tabular data with **awk** is very easy, provided that the input is correctly formatted. Most software producing tabular data use specific features of this family of formats, and **awk** programs processing tabular data are often specific to a data produced by a specific software. If a more generic or robust solution is required, most popular languages provide libraries accommodating with a lot of features found in tabular data:

- optional column names on the first line
- mixture of quoted and unquoted column values
- various delimiters
- localised formats for floating numbers

While it definitely possible to handle all these features cleanly and generically with **awk** this is probably not worth the effort.

Exchanging two columns in tabular data

Given a file using `;` as a column delimiter. Permuting the first and the second column is accomplished by

```
awk -F';' -v 'OFS=;' '{ swap = $2; $2 = $1; $1 = swap; print }'
```

Compute the average of values in a column from tabular data

Given a file using `;` as a column delimiter. We compute the mean of the values in the second column with the following program, the provided input is the list of grades of a student group:

```
awk -F';' '{ sum += $2 } END { print(sum / NR) }' <<EOF
Alice;2
Victor;1
Barbara;1
Casper;4
Deborah;0
Ernest;1
Fabiola;4
Giuseppe;4
EOF
```

The output of this program is `2.125`.

Remember that `NR` holds the number of the line being processed, in the `END` block it therefore hold the total number of lines in the file.

Remember that in many applications *(monitoring, statistics)*, the median is a much more useful information. See the corresponding example.

Selecting specific columns in tabular data

We assume a file using ; as a column delimiter. Selecting a specific set of columns only requires a *print* statement. For instance, the following program selects the columns 3, 4 and 7 from its input:

```
awk -F';' -v 'OFS=;' '{ print $3, $4, $7 }'
```

It is as usual possible to more carefully choose lines to print. The following program selects the columns 3, 4 and 7 from its input when the first field is `Alice` or `Bob`:

```
awk -F';' -v 'OFS=;' '($1 == "Alice") || ($1 == "Bob") { print $3, $4, $7 }'
```

Compute the median of values in a column from tabular data

Given a file using `;` as a column delimiter. We compute the median of the values in the second column with the following program, written for **GNU awk**. The provided input is the list of grades of a student group:

```
gawk -F';' '{ sample[NR] = $2 }
 END {
   asort(sample);
   if(NR % 2 == 1) {
     print(sample[int(NR/2) + 1])
   } else {
     print(sample[NR/2])
   }
}' <<EOF
Alice;2
Victor;1
Barbara;1
Casper;4
Deborah;0
Ernest;1
Fabiola;4
Giuseppe;4
EOF
```

The output of this program is `1`.

Remember that `NR` holds the number of the line being processed, in the `END` block it therefore hold the total number of lines in the file.

Many implementations of **awk** do not have a function to sort arrays, which therefore need to be defined before the code above could be used.

Selecting a set of lines between two patterns

Pattern matching can be used effectively with `awk` as it controls the actions that follows it i.e. `{ pattern } { action }`. One cool use of the pattern-matching is to select multiple between two patterns in a file say `patternA` and `patternB`

```
$ awk '/patternA/,/patternB/' file
```

Assume my file contents are as follows, and I want to extract the lines only between the above pattern:-

```
$ cat file
This is line - 1
This is line - 2
patternA
This is line - 3
This is line - 4
This is line - 5
patternB
This is line - 6

$ awk '/patternA/,/patternB/' file
patternA
This is line - 3
This is line - 4
This is line - 5
patternB
```

The above command doesn't do any specific `{ action }` other than to print the lines matching, but any specific actions within the subset of lines can be applied with an action block (`{}`).

Read Useful one-liners - calculating average from a CSV etc online: https://riptutorial.com/awk/topic/3331/useful-one-liners---calculating-average-from-a-csv-etc

Chapter 12: Variables

Examples

Command-line variable assignment

To assign variables from the command-line, `-v` can be used:

```
$ awk -v myvar="hello" 'BEGIN {print myvar}'
hello
```

Note that there are no spaces around the equal sign.

This allows to use shell variables:

```
$ shell_var="hello"
$ awk -v myvar="$shell_var" 'BEGIN {print myvar}'
hello
```

Also, this allows to set built-in variables that control `awk`:

See an example with `FS` (field separator):

```
$ cat file
1,2;3,4
$ awk -v FS="," '{print $2}' file
2;3
$ awk -v FS=";" '{print $2}' file
3,4
```

Or with `OFS` (output field separator):

```
$ echo "2 3" | awk -v OFS="--" '{print $1, $2}'
2--3
$ echo "2 3" | awk -v OFS="+" '{print $1, $2}'
2+3
```

Passing parameters to a program using the -v option

The option `-v` followed by an assignment of the form *variable*=*value* can be used to pass parameters to an **awk** program. This is illustrated by the **punishment** program below, whose job is to write *count* times the sentence “I shall not talk in class.” on standard output. The following example uses the value 100, which is very popular among teachers:

```
awk -v count=100 'BEGIN {
  for(i = 1; i <= count; ++i) {
    print("I shall not talk in class.")
  }
  exit
}'
```

It is possible to pass multiple parameters with repeated usage of the `-v` flag:

```
awk -v count=100 -v "sentence=I shall not talk in class." 'BEGIN {
  for(i = 1; i <= count; ++i) {
    print(sentence)
  }
  exit
}'
```

There is no built-in support for array or list parameters, these have to be handled manually. A classical approach to pass a list parameter is to concatenate the list using a delimiter, popular choices are `:`, `|` or `,`. The *split* function then allows to recover the list as an **awk** array:

```
awk -v 'serialised_list=a:b:c:d:e:f' 'BEGIN {
  list_sz = split(serialised_list, list, ":")
  for(i = 1; i <= list_sz; ++i) {
    printf("list: %d: %s\n", i, list[i])
  }
  exit
}'
```

The output of this **awk** program is

```
list: 1: a
list: 2: b
list: 3: c
list: 4: d
list: 5: e
list: 6: f
```

Sometimes it is more convenient to recover list items as keys of an **awk** array, as this allows easy membership verification. For instance, the following program print each line whose first word does not belong to a fixed list of exceptions:

```
awk -v 'serialised_exception_list=apple:pear:cherry' 'BEGIN {
  _list_sz = split(serialised_exception_list, _list, ":")
  for(i = 1; i <= _list_sz; ++i) {
    exception[_list[i]]
  }
}

! ($1 in exception) { print }' <<EOF
apple Apples are yummy, I like them.
pineapple Do you like pineapple?
EOF
```

The output of this program is

```
pineapple Do you like pineapple?
```

As a final example, we show how to wrap the **punishment** program into a shell script, as this illustrates how a shell script conveys parameters to an auxiliary **awk** script:

```
#!/bin/sh

usage()
{
   cat <<EOF
Usage: punishment [-c COUNT][-s SENTENCE]
 Prepare your punishments for you
EOF
}

punishment_count='100'
punishment_sentence='I shall not talk in class.'
while getopts "c:hs:" OPTION; do
  case "${OPTION}" in
    c) punishment_count="${OPTARG}";;
    s) punishment_sentence="${OPTARG}";;
    h) usage; exit 0;;
    *) usage; exit 64;;
  esac
done

awk -v "count=${punishment_count}" -v "sentence=${punishment_sentence}" 'BEGIN {
  for(i = 1; i <= count; ++i) {
    print(sentence)
  }
  exit
}'
```

Local variables

The **awk** language does not directly support variables local to functions. It is however easy emulate them by adding extra arguments to functions. It is traditional to prefix these variables by a _ to indicate that they are not actual parameters.

We illustrate this technique with the definition of a `single_quote` function that adds single quotes around a string:

```
# single_quote(TEXT)
#  Return a string made of TEXT surrounded by single quotes

function single_quote(text, _quote) {
  _quote = sprintf("%c", 39)
  return sprintf("%s%s%s", _quote, text, _quote);
}
```

The simpler approach of using `sprintf("'%s'", text)` leads to practical problems because **awk** scripts are usually passed as single quoted arguments to the **awk** program.

Assignment Arguments

Assignment arguments appear at the end of an `awk` invocation, in the same area as file variables, both `-v` assignments and argument assignments must match the following regular expression. (assuming a POSIX locale)

```
^[[:alpha:]_][[:alnum:]_]*=
```

The following example assumes a file `file` containing the following: `1 2 3` (white space separated)

```
$ awk '{$1=$1}1' file OFS=, file OFS=- file
1 2 3
1,2,3
1-2-3
```

Read Variables online: https://riptutorial.com/awk/topic/1403/variables

Credits

S. No	Chapters	Contributors
1	Getting started with awk	Andrea, antonio, AstraSerg, Community, David, fedorqui, George Vasiliou, kdhp, Michael Vehrs
2	Arrays	engineercoding
3	Built-in functions	Alejandro Teixeira Muñoz, Armali
4	Built-in Variables	Alejandro Teixeira Muñoz, Andrzej Pronobis, FoldedChromatin, George Vasiliou, James Brown, sat
5	Fields	chaos
6	Patterns	Andrzej Pronobis
7	Patterns and Actions	Alejandro Teixeira Muñoz, Michael Vehrs
8	Row Manipulation	pacomet, Scheff, UNagaswamy
9	String manipulation functions	Alejandro Teixeira Muñoz, Andrzej Pronobis, AstraSerg, fedorqui, Inian, kdhp, Michael Le Barbier Grünewald, schot, Thor
10	Two-file processing	fedorqui, George Vasiliou
11	Useful one-liners - calculating average from a CSV etc	Chris Seymour, Inian, karakfa, Michael Le Barbier Grünewald
12	Variables	Andrzej Pronobis, anishsane, fedorqui, karakfa, kdhp, Michael Le Barbier Grünewald, Michael Vehrs