Extended System Programming Laboratory

Lecture 8 – AWK script language, SED

Dr. Marina Kogan-Sadetsky

AWK
script language

GNU AWK

- scripting language that works with streams of textual data (files)
- created at Bell Labs in 1977

LEARN AWK

interpreted programming language

AWK Tutorial

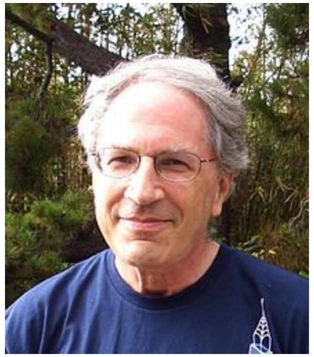
nice tutorial

also nice tutorial

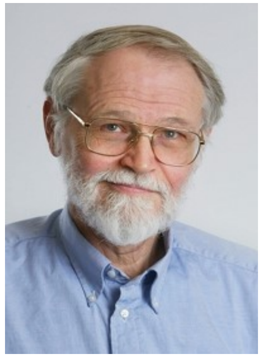
awk(1) - Linux man page

full tutorial

Alfred
Aho



Peter
Weinberger



Brian
Kernighan

file.csv

```
1,2,3
4,5,6
7,8,9
```

'csv' means comma separated values

awk executes given code instructions on each line of the input file

```
marina@vm:~/SPLab $  cat file.csv
1,2,3
4,5,6
7,8,9
marina@vm:~/SPLab $  awk '{print $0}'  file.csv
1,2,3
4,5,6
7,8,9
marina@vm:~/ SPLab $
```

$0 prints the whole line

test.awk

```
{ print $0 }
```

test.awk contains script written in awk

```
marina@vm:~/SPLab $  awk –f /test.awk  file.csv
1,2,3
4,5,6
7,8,9
marina@vm:~/SPLab $
```

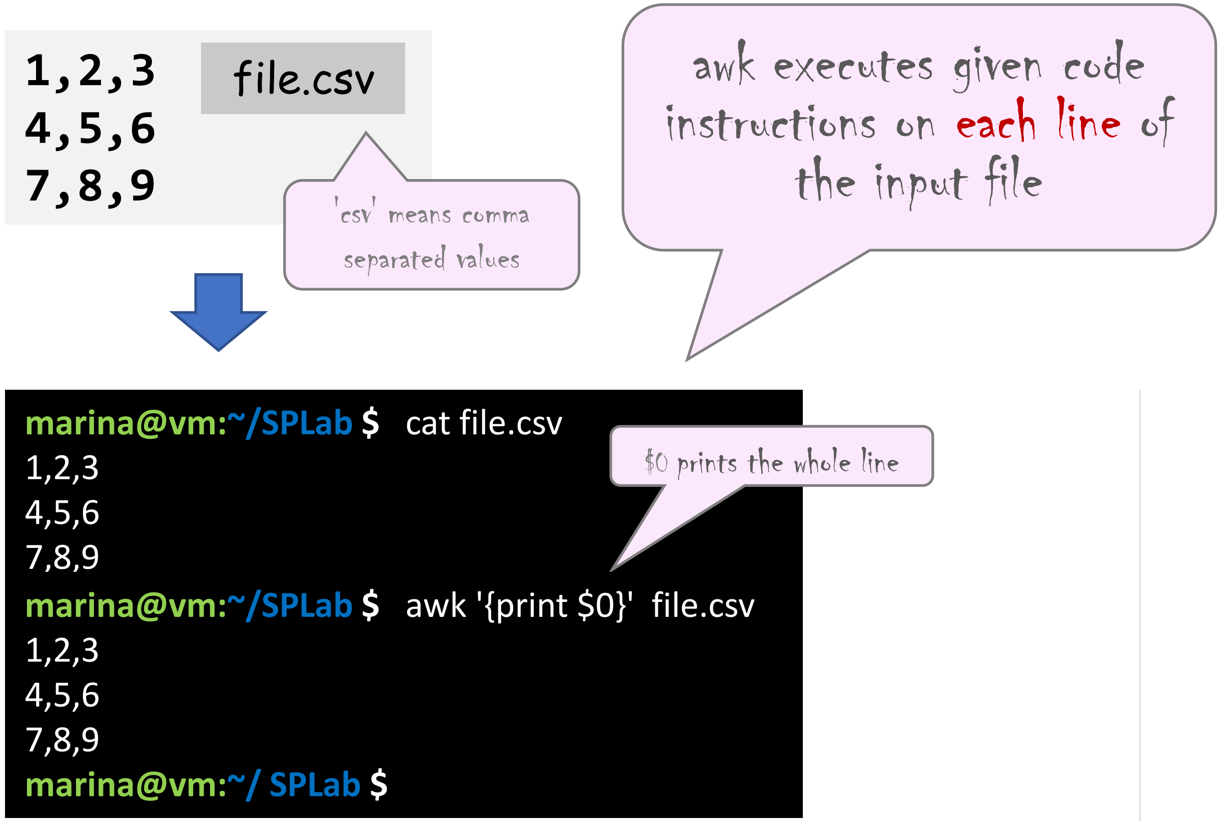
-f means that the script is given in a file

```
marina@vm:~/SPLab $ echo 1,2,3 > file.csv
marina@vm:~/SPLab $ echo 4,5,6  >> file.csv
marina@vm:~/SPLab $ echo 7,8,9  >> file.csv
marina@vm:~/SPLab $ cat file.csv
1,2,3
4,5,6
7,8,9
marina@vm:~/SPLab $
```

> redirects output of echo into file.csv. If file.csv does not exist, it would be created. If file.csv exists, its content would be discarded.

>> also redirects output, but the output would be appended to file.csv

it is possible to create input file in the following way via terminal



file.csv

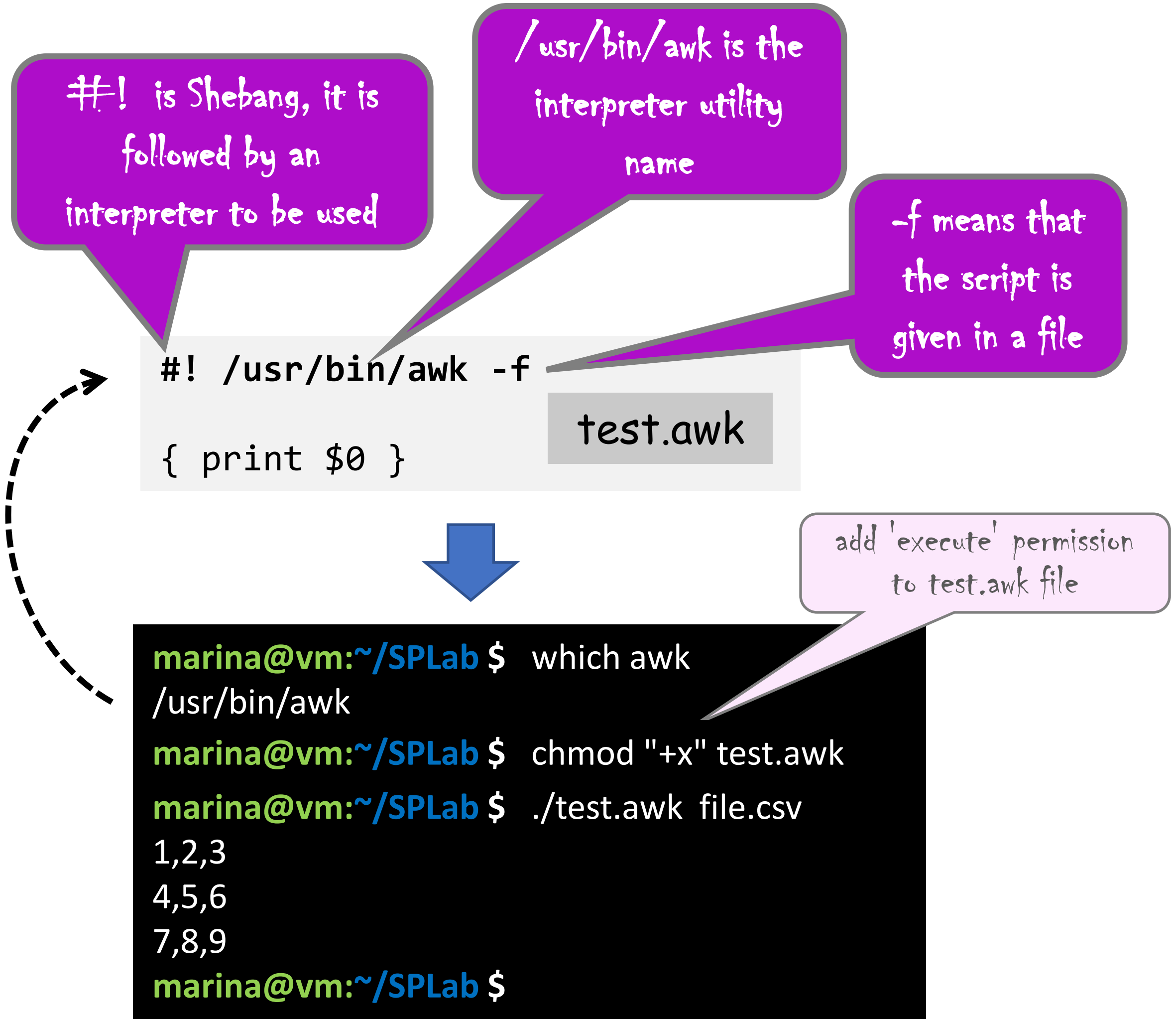
```
1,2,3
4,5,6
7,8,9
```

'csv' means comma separated values

awk executes given code instructions on each line of the input file

```
marina@vm:~/SPLab $  cat file.csv
1,2,3
4,5,6
7,8,9
marina@vm:~/SPLab $  awk '{print $0}'  file.csv
1,2,3
4,5,6
7,8,9
marina@vm:~/ SPLab $
```

$0 prints the whole line



test.awk

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

{ print $0 }
```

#! is Shebang, it is followed by an interpreter to be used

/usr/bin/awk is the interpreter utility name

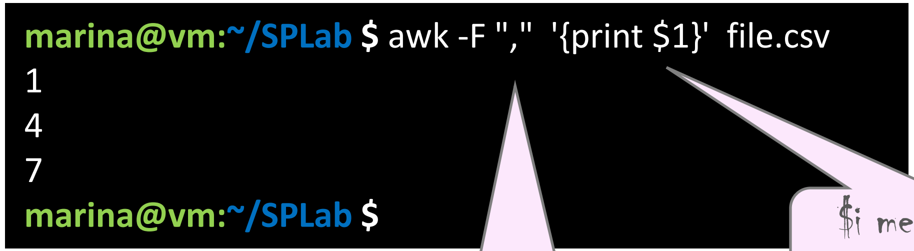
-f means that the script is given in a file

```
marina@vm:~/SPLab $  which awk
/usr/bin/awk
marina@vm:~/SPLab $  chmod "+x" test.awk
marina@vm:~/SPLab $  ./test.awk  file.csv
1,2,3
4,5,6
7,8,9
marina@vm:~/SPLab $
```

add 'execute' permission to test.awk file

file.csv

```
1,2,3
4,5,6
7,8,9
```



```
marina@vm:~/SPLab $ awk -F "," '{print $1}' file.csv
1
4
7
marina@vm:~/SPLab $
```

field separator is ","

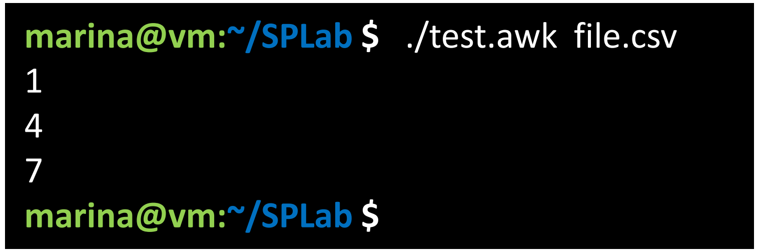
$i means i'th field value in the line

BEGIN block contains instruction to be executed before reading the input file

test.awk

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

BEGIN { FS = ","}
      {
        print $1
      }
```



```
marina@vm:~/SPLab $  ./test.awk file.csv
1
4
7
marina@vm:~/SPLab $
```

file.csv

```
1,2,3
4,5,6
7,8,9
```

```
marina@vm:~/SPLab $ awk -F "," '{print $1}'  file.csv
1
4
7
marina@vm:~/SPLab $
```

field separator is ","

$i means i'th field value in the line

test.awk

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

BEGIN { FS = ","
        print "first\tsecond"
      }
      {
        print $1 "\t" $2
      }
```

BEGIN block contains instruction to be executed before reading the input file

```
marina@vm:~/SPLab $  ./test.awk  file.csv
first           second
1               2
4               5
7               8
marina@vm:~/SPLab $
```

Print vs. Printf

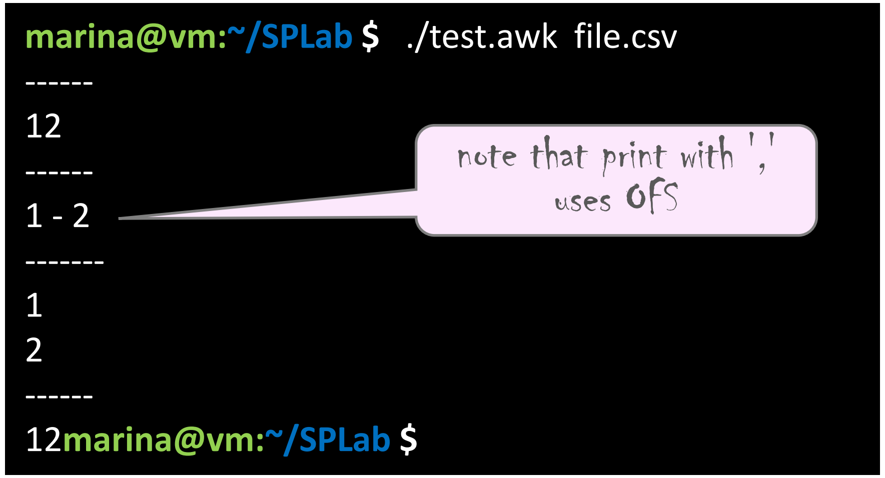
```
1,2,3
```

file.csv

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

BEGIN { OFS = " - "
        FS = ","
      }
      {
        print "------"
        print $1 $2
        print "------"
        print $1, $2
        print "------"
        print $1
        print $2
        print "------"
        printf $1 $2
      }
```

OFS means output field separator



```
marina@vm:~/SPLab $  ./test.awk  file.csv
------
12
------
1 - 2
-------
1
2
------
12marina@vm:~/SPLab $
```

note that print with ',' uses OFS

file.csv

```
1,2,3
4,5,6
7,8,9
```

test.awk

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

BEGIN { FS = ","
        print "first\tsecond\tthird"
      }
      {
        print $1 "\t" $2 "\t" $3
        total = total +1
      }
END   { print "--------------------"
        print total " lines"
      }
```

total is a variable that we define

END block contains instruction to be executed after reading the input file

```
marina@vm:~/SPLab $  ./test.awk  file.csv
first        second        third
1            2             3
4            5             6
7            8             9
--------------------------------------
3 lines
marina@vm:~/SPLab $
```

file.csv

```
1,2,3
4,5,6,7,8,9
10
11,,,14
```

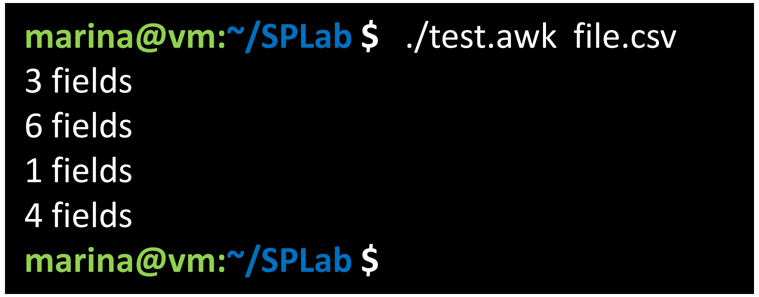
note that empty fields are also counted

test.awk

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

BEGIN { FS = "," }
      {
        print NF " fields"
      }
```

NF means number of fields (in line)



```
marina@vm:~/SPLab $  ./test.awk  file.csv
3 fields
6 fields
1 fields
4 fields
marina@vm:~/SPLab $
```

file.csv

```
1,2,3
4,5,6,7,8,9
10
11,,,14
```

test.awk

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

BEGIN { FS = "," }
      {
        print "line " NR " : " NF " fields"
      }
```

NR means current line number

```
marina@vm:~/SPLab $  ./test.awk  file.csv
line 1 : 3 fields
line 2 : 6 fields
line 3 : 1 fields
line 4 : 4 fields
marina@vm:~/SPLab $
```

a.csv

```
Hello World
Hi
I Love SPLab
```

b.csv

```
1
2334
```

test.awk

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

{
     print NR " : " $0
     getline
     print NR " : " "[next] " $0
 }
```

getline means get the next line from the input file

```
marina@vm:~/SPLab $  ./test.awk  a.csv
1 : Hello World
2 : [next] Hi
3 : I Love SPLab
3 : [next] I Love SPLab
marina@vm:~/SPLab $
```

getline increments NR value

while reaching EOF, getline continues reading the same (last) line

test.awk

```
#! /usr/bin/awk -f
BEGIN {
            while(( getline line < "b.csv" ) > 0 )
                   print line
       }
```

read additional file

```
marina@vm:~/SPLab $  ./test.awk
1
2334
marina@vm:~/SPLab $
```

test.awk

```
#! /usr/bin/awk -f
BEGIN {
            while((i=getline line < "b.csv") > 0)
                   print line " [" i "]"
            print i
       }
```

On success, getline returns 1. When b.csv reaches EOF, getline returns 0.

```
marina@vm:~/SPLab $  ./test.awk
1 [1]
2334 [1]
0
marina@vm:~/SPLab $
```

a.csv

```
Hello World
Hi
I Love SPLab
```

b.csv

```
1
2334
```

test.awk

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

{
     print NR " : " $0
     getline
     print NR " : " "[next] " $0
 }
```

getline means get the next line from the input file

```
marina@vm:~/SPLab $  ./test.awk  a.csv
1 : Hello World
2 : [next] Hi
3 : I Love SPLab
3 : [next] I Love SPLab
marina@vm:~/SPLab $
```

getline increments NR value

while reaching EOF, getline continues reading the same (last) line

test.awk

```
#! /usr/bin/awk -f
BEGIN {
            while(( getline line < "b.csv" ) > 0 )
                  print line
        }
```

read additional file

```
marina@vm:~/SPLab $  ./test.awk
1
2334
marina@vm:~/SPLab $
```

test.awk

```
#! /usr/bin/awk -f
{
   getline line < "b.csv"
   print $0 " [" line "]"
}
```

when b.csv reaches EOF, getline returns 0, but line variable still contains the value of the last line of b.csv

```
marina@vm:~/SPLab $  ./test.awk a.csv
Hello World [1]
Hi [2334]
I Love SPLab [2334]
marina@vm:~/SPLab $
```

a.csv

```
Hello World
Hi
I Love SPLab
```

b.csv

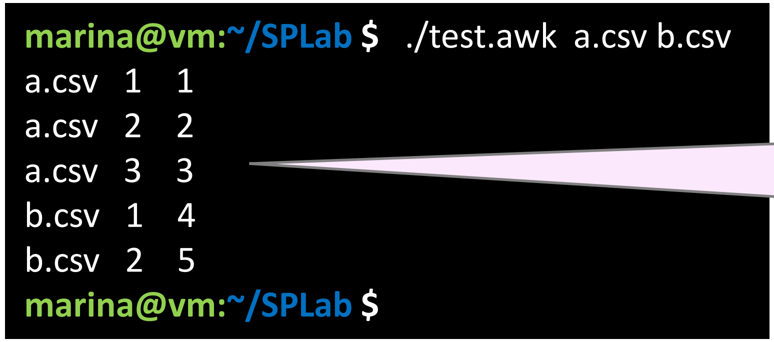
```
1
2334
```

test.awk

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

{
    print FILENAME, FNR, NR
}
```

FNR means number of line in the current input file



```
marina@vm:~/SPLab $  ./test.awk  a.csv b.csv
a.csv  1   1
a.csv  2   2
a.csv  3   3
b.csv  1   4
b.csv  2   5
marina@vm:~/SPLab $
```

FNR is private line counter of each input file, NR is shared line counter

file.csv

```
hi
hello
h
world
```

print only lines with length (i.e., number of chars in $0) bigger than 3

```
marina@vm:~/SPLab $  awk ' length($0) > 3 {print $0}' file.csv
hello
world
marina@vm:~/SPLab $
```

Regular Expression

file.csv

```
cat is fun
refund
fan
fun
future
flan
```

```
marina@vm:~/SPLab $  awk '/fun/' file.csv
cat is fun
refund
fun
marina@vm:~/SPLab $
```

'/fun/' is a regular expression, means line that contains a word (or part of it with pattern 'fun'

```
marina@vm:~/SPLab $  awk '/^fun/' file.csv
fun
marina@vm:~/SPLab $
```

^ stands for <at the beginning of the line>

```
marina@vm:~/SPLab $  awk '/fun$/' file.csv
fun
cat is fun
marina@vm:~/SPLab $
```

$ stands for <at the end of the line>

Regular Expression

file.csv

```
cat is fun
refund
fan
fun
future
flan
```

```
marina@vm:~/SPLab $  awk '/f.n/' file.csv
cat is fun
refund
fan
fun
marina@vm:~/SPLab $
```

. stands for <any character>

```
marina@vm:~/SPLab $  awk '/^f[ua]n/' file.csv
fun
fan
marina@vm:~/SPLab $
```

[ua] means 'u' or 'a'

```
marina@vm:~/SPLab $  awk '/f[^uk]n$/' file.csv
fan
marina@vm:~/SPLab $
```

[^uk] stands for any character except 'u' and 'k'

Regular Expression

file.csv

```
cat is funny
dog is cute
fan
fun
future
flan
```

```
marina@vm:~/SPLab $  awk '/cat|ure/' file.csv
cat is funny
future
marina@vm:~/SPLab $
```

/cat|ure/ stands for lines that contain strings 'cat' or 'ure'

```
marina@vm:~/SPLab $  awk '/fl?an/' file.csv
fan
flan
marina@vm:~/SPLab $
```

'l?' means 'l' should appear one or zero times

```
marina@vm:~/SPLab $  awk '/fun*/' file.csv
cat is funny
fun
future
marina@vm:~/SPLab $
```

fun* means 'n' may appear zero or more times

```
marina@vm:~/SPLab $  awk '/fun+/' file.csv
cat is funny
fun
marina@vm:~/SPLab $
```

```
marina@vm:~/SPLab $  awk '/is (funny|cute)/' file.csv
cat is funny
dog is cute
marina@vm:~/SPLab $
```

(funny|cute) stands for 'funny' or 'cute'

```
marina@vm:~/SPLab $  awk '$1 ~ /[uo]/ {print $0}' file.csv
dog is cute
fun
future
marina@vm:~/SPLab $
```

line is printed only if it first field contains 'u' or 'o' character

```
marina@vm:~/SPLab $  awk '$1 !~ /[cf]/ {print $0}' file.csv
dog is cute
marina@vm:~/SPLab $
```

line is printed only if its first field does not contains 'u' or 'o' characters

Arrays

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

BEGIN {
            A["mango"] = "yellow"
            A[2] = "blue"
            A["red"] = 3

            for(i in A)
                print i " - " A[i]
        }
```

array index may be integer number or string

```
marina@vm:~/SPLab $  ./test.awk
red - 3
2 - blue
mango – yellow
marina@vm:~/SPLab $
```

Multi-dimensional Arrays

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

BEGIN {   A[1]=3;
          A[1,2]=5;
          A[2,4,"Hi"]=7;
         for (i in A)
             print "index = " i  ",  value = " A[i]
      }
```

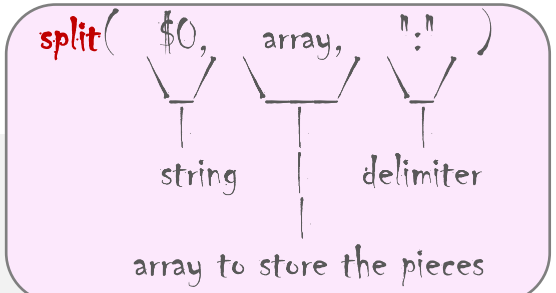
multi-dimensional array

```
marina@vm:~/SPLab $  ./test.awk
index = 24Hi,  value = 7
index = 12,  value = 5
index = 1,  value = 3
marina@vm:~/SPLab $
```

note that index in multi-dimensional array is the concatenation of all the flat indices

AWK converts the multiple indices into strings and concatenates them together, with a separator SUBSEP (built-in variable) between them. The combined string is used as a single index into an ordinary, one-dimensional array.

https://www.gnu.org/software/gawk/manual/html_node/Multidimensional.html



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

BEGIN {  A[1]=3;
         A[1,2]=5;
         A[2,4,"Hi"]=7;
         for (i in A) {
             n = split(i,sep,SUBSEP)
             for (j = 1; j <= n; j++)
                 printf sep[j] " "
             print  "\n-----------"
         }
      }
```

```
marina@vm:~/SPLab $  ./test.awk
2 4 Hi
-----------
1 2
-----------
1
-----------
marina@vm:~/SPLab $
```

AWK Script Example

file.csv

```
John Thomas
Julie Andrews
Alex Tremble
John Tomas
Alex Gordon
Alex Jordan
```

test.awk

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

BEGIN { FS = " " }
   {
      A[$1]++;
   }
END {
      for (i in A)
         printf "%d people named %s\n", A[i], i
   }
```

loop of all indices in A

?

What is the output of the script ?

AWK Script Example

file.csv

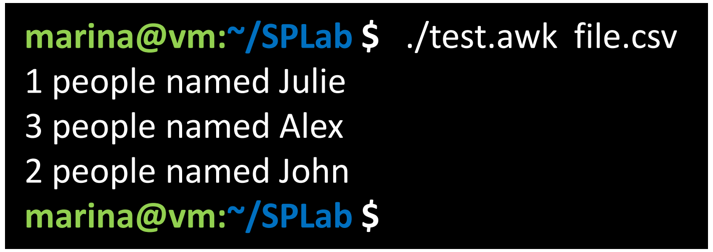
```
John Thomas
Julie Andrews
Alex Tremble
John Tomas
Alex Gordon
Alex Jordan
```

test.awk

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

BEGIN { FS = " " }
   {
      A[$1]++;
   }
END {
      for (i in A)
         printf "%d people named %s\n", A[i], i
   }
```

loop of all indices in A



```
marina@vm:~/SPLab $  ./test.awk  file.csv
1 people named Julie
3 people named Alex
2 people named John
marina@vm:~/SPLab $
```

What is the output of the script ?

Counters of the first field values

AWK Script Example

test.awk

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

BEGIN {
      A[1] = 3;
      A[1,2] = 5;
      A[2,4,"Hi"] = 7;
      A["a"] = "b";
      A["wk",8] = "hello";

      asort(A)
      for (i in A)
         printf i ":" A[i] " "
      print "\n"
}
```

asort sorts the contents of array using GAWK's normal rules for comparing values, and replaces the indexes of the sorted array with sequential integers, starting with 1

```
marina@vm:~/SPLab $  ./test.awk
1:3 2:5 3:7 4:b 5:hello
marina@vm:~/SPLab $
```

note that after sort, the indexes are replaced to 1,2,3,4,5, ...

test.awk

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

BEGIN  {
      B["c"] = "value1"
      B["a"] = "value2"
      B["b"] = "value3"

      asorti(B)
      for (i in B)
         printf i ":" B[i] " "
      print ""
}
```

asorti sorts the array indexes and not the array values

```
marina@vm:~/SPLab $  ./test.awk
a:e b:f c:d  --> 1:a 2:b 3:c
marina@vm:~/SPLab $
```

AWK Script Example

test.awk

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

BEGIN {
        str = "Hello, emanuel"
        printf str " --> "

        gsub("e", "E", str)
        print str
        }
```

```
marina@vm:~/SPLab $  ./test.awk
Hello, emanuel --> HEllo, EmanuEl
marina@vm:~/SPLab $
```

gsub (global substitution) replaces every occurrence of regex with the given string.

test.awk

```
#! /usr/bin/awk -f
{
  gsub("cat", "dog")
  print $0
}
```

If third parameter is omitted, then $0 is used.

```
marina@vm:~/SPLab $  echo "cat is cat" | ./test.awk
dog is dog
marina@vm:~/SPLab $
```

test.awk

```
#! /usr/bin/awk -f
{
  sub("cat", "dog")
  print $0
}
```

sub replaces only the first occurrence

```
marina@vm:~/SPLab $  echo "cat is cat" | ./test.awk
dog is cat
marina@vm:~/SPLab $
```

AWK Script Example

file.csv

```
hi         hello h
    world   !   !             !
```

What is the output of these scripts ?

test.awk

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

{
  gsub(/[[:blank:]]+/, " ", $0)
  print "[" $0 "]"
}
```

[[:blank:]] means space or tab

?

test.awk

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

{
  gsub(/^[[:blank:]]+/, "", $0)
  print "[" $0 "]"
}
```

?

test.awk

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

{
  gsub(/[[:blank:]]+$/, "", $0)
  print "[" $0 "]"
}
```

?

AWK Script Example

file.csv

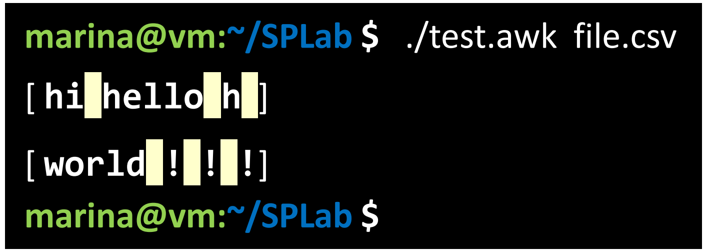
```
hi           hello h
    world   !   !                !
```

What is the output of these scripts ?

test.awk

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

{
  gsub(/[[:blank:]]+/, " ", $0)
  print "[" $0 "]"
}
```



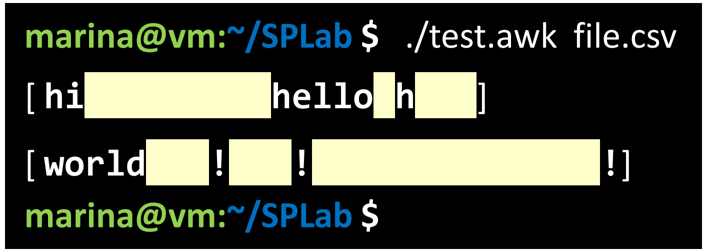
```
marina@vm:~/SPLab $  ./test.awk  file.csv
[hi hello h ]
[world ! ! !]
marina@vm:~/SPLab $
```

all the double whitespaces are removed

test.awk

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

{
  gsub(/^[[:blank:]]+/, "", $0)
  print "[" $0 "]"
}
```



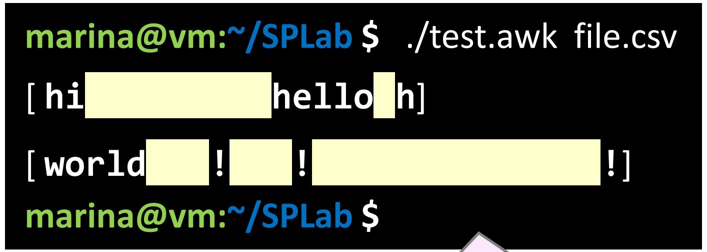
```
marina@vm:~/SPLab $  ./test.awk  file.csv
[hi           hello h   ]
[world   !   !               !]
marina@vm:~/SPLab $
```

whitespaces at the beginning of the lines are removed

test.awk

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

{
  gsub(/[[:blank:]]+$/, "", $0)
  print "[" $0 "]"
}
```



```
marina@vm:~/SPLab $  ./test.awk  file.csv
[hi           hello h]
[world   !   !               !]
marina@vm:~/SPLab $
```

whitespaces at the end of the lines are removed

AWK Script Example

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

function sum(a, b) {
   return a + b
}

function main(a, b){
    print "sum =", sum(a, b)
}

BEGIN {
               main(10, 20)
         }
```

test.awk

What is the output of these scripts ?

?

AWK Script Example

test.awk

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

function sum(a, b) {
   return a + b
}

function main(a, b){
    print "sum =", sum(a, b)
}

BEGIN {
                main(10, 20)
         }
```

```
marina@vm:~/SPLab $  ./test.awk
sum = 30
marina@vm:~/SPLab $
```

test.awk

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

function sum(a, b) {
   return a + b
}

function main(a, b){
    print "sum =", sum(a, b)
}

BEGIN {
                main("Hi", "Bye")
         }
```

?

What is the output of these scripts ?

AWK Script Example

test.awk

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

function sum(a, b) {
   return a + b
}

function main(a, b){
    print "sum =", sum(a, b)
}

BEGIN {
            main(10, 20)
        }
```

```
marina@vm:~/SPLab $  ./test.awk
sum = 30
marina@vm:~/SPLab $
```

test.awk

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

function sum(a, b) {
   return a + b
}

function main(a, b){
    print "sum =", sum(a, b)
}

BEGIN {
            main("Hi", "Bye")
        }
```

```
marina@vm:~/SPLab $  ./test.awk
sum = 30
sum = 0
marina@vm:~/SPLab $
```

What is the output of these scripts ?

the answer is meaningless...
what can be done ?

AWK Script Example

test.awk

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

function sum(a, b) {
   return a + b
}

function main(a, b){
    print "sum =", sum(a, b)
}

BEGIN {
            main(10, 20)
        }
```

```
marina@vm:~/SPLab $  ./test.awk
sum = 30
marina@vm:~/SPLab $
```

test.awk

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

function sum(a, b) {
  return a + b
}
function main(a, b){
  print typeof(a), typeof(b)
  if (typeof(a) == "number" && typeof(b) == "number")
    print "sum =", sum(a, b)
  else
    print "error: at least one argument is not a number"
}
BEGIN {
  main(10, 20)
  main("Hi", "Bye")
}
```

since AWK is typeless, we should better check the data types of the function arguments before calculating they sum

We can do this check also with regular expression. If a and b are strings, but contain only digits, then we can convert them to numbers.

```
marina@vm:~/SPLab $  ./test.awk
number number
sum = 30
string string
error: at least one argument is not a number
marina@vm:~/SPLab $
```

AWK Script Example

test.awk

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

BEGIN { printf "" > "output.txt" }

 {
     for(i=1;i<=NF; i++) {
        if(i == 1)
            printf $(i) >> "output.txt"
        else
            printf "+" $(i) >> "output.txt"
        sum +=$(i)
     }

     print " = " sum >> "output.txt"
     sum = 0
 }
```

$(i) means i'th field of the line

the output of the script is redirected to "output.txt" file

What is the output of these scripts ?

file.csv

```
1 2
3 4 5
7 4 8
9 6 5 2 3
```

?

AWK Script Example

test.awk

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

BEGIN { printf "" > "output.txt" }

 {
      for(i=1;i<=NF; i++) {
         if(i == 1)
             printf $(i) >> "output.txt"
         else
             printf "+" $(i) >> "output.txt"
         sum +=$(i)
      }

      print " = " sum >> "output.txt"
      sum = 0
 }
```

$(i) means i'th field of the line

the output of the script is redirected to "output.txt" file

file.csv

```
1 2
3 4 5
74 8
9 6 5 2 3
```

```
marina@vm:~/SPLab $  ./test.awk file.csv
marina@vm:~/SPLab $  cat output.txt
1+2 = 3
3+4+5 = 12
74+8 = 82
9+6+5+2+3 = 25
marina@vm:~/SPLab $
```

AWK Script Example

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

 {
     print "---------------"
     print $0  | "tr [a-z] [A-Z]"
     print $0  | "rev"
     system("echo "$0" | wc -w ")
 }
```

tr means translate, i.e., change string

rev means reverse string

system() is system call

?

What is the output of these scripts ?

```
Hello World
Hi
I Love SPLab
```

file.csv

AWK Script Example

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

 {
     print "---------------"
     print $0  | "tr [a-z] [A-Z]"
     print $0  | "rev"
     system("echo "$0" | wc -w ")
  }
```

tr means translate, i.e., change string

rev means reverse string

system() is system call

```
marina@vm:~/SPLab $  ./test.awk file.csv
---------------
HELLO WORLD
dlroW olleH
2
---------------
HI
iH
1
---------------
I LOVE SPLAB
baLPS evoL I
3
marina@vm:~/SPLab $
```

```
Hello World
Hi
I Love SPLab
```

file.csv

AWK – summary

self-read

regexp	meaning
[ad]	'a' or 'd'
[c-k]	any character in range [c-k]
[^a-d]	any character except [a-d]
[A-Za-z0-9]	any letter or digit
[[:alnum:]]	any letter or digit (**posix**)
\w	any word
\s	space or tab
[[:blank:]]	space or tab (**posix**)
\d	any digit
[[:digit:]]	any digit (**posix**)
\.	dot
^	at the beginning of line
$	at the end of line

regexp	meaning
.	any character
+	one or more times
*	zero or more times
?	zero or one time
{n}	exactly n times
{n,}	n or more times
{n, m}	between n and m times

IO statement	meaning
getline	Set $0 to be next input record
getline < file	Set $0 to be next input record of given file
getline var	Set var to be next input record
getline var < file	Set var to be next input record of given file
command | getline [var]	Run command piping the output either into $0 or var
next	Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, the END block(s), if any, are executed.
nextfile	Stop processing the current input file. The next input record read comes from the next input file. FILENAME and ARGIND are updated, FNR is reset to 1, and processing starts over with the first pattern in the AWK program. If the end of the input data is reached, the END block(s), if any, are executed.
system(cmd-line)	Execute the command cmd-line, and return the exit status (**posix**)

control statement

if (condition)
statement
else
statement

expr1 ? expr2 : expr3 means If expr1 is true, execute expr2, otherwise execute expr3
while (condition) statement

do
statement
while (condition)

for (expr1; expr2; expr3) statement
for (item in array) statement
break
continue
exit [expression]

string functions
asort(s [, d])
asorti(arr [, d [, how]])
gensub(r, s, h [, t])
gsub(r, s [, t])
index(s, t)
length([s])
match(s, r [, a])
split(s, a [, r])
sprintf(fmt, expr-list)
strtonum(str)
sub(r, s [, t])
substr(s, i [, n])
tolower(str)
toupper(str)

operator	meaning
++ --	increment / decrement
+ - * - %	math operators
^ or **	exponentiation
!	logical negation
|| &&	logical operators

https://gnu.huihoo.org/gawk-3.0.3/html_node/gawk_186.html

self-read

AWK Data Types

The value of an awk expression is always either a number or a string.
Some contexts (such as arithmetic operators) require numeric values. They convert strings to numbers by interpreting the text of the string as a number. If the string does not look like a number, it converts to zero.

Other contexts (such as concatenation) require string values. They convert numbers to strings by effectively printing them with sprintf. See section Conversion of Strings and Numbers, for the details.

To force conversion of a string value to a number, simply add zero to it. If the value you start with is already a number, this does not change it.

To force conversion of a numeric value to a string, concatenate it with the null string.
Comparisons are done numerically if both operands are numeric, or if one is numeric and the other is a numeric string. Otherwise one or both operands are converted to strings and a string comparison is performed.
Fields, getline input, FILENAME, ARGV elements, ENVIRON elements and the elements of an array created by split are the only items that can be numeric strings. String constants, such as "3.1415927" are not numeric strings, they are string constants. The full rules for comparisons are described in section Variable Typing and Comparison Expressions.

Uninitialized variables have the string value "" (the null, or empty, string). In contexts where a number is required, this is equivalent to zero.

See section Variables, for more information on variable naming and initialization; see section Conversion of Strings and Numbers, for more information on how variable values are interpreted.

SED

Stream Editor

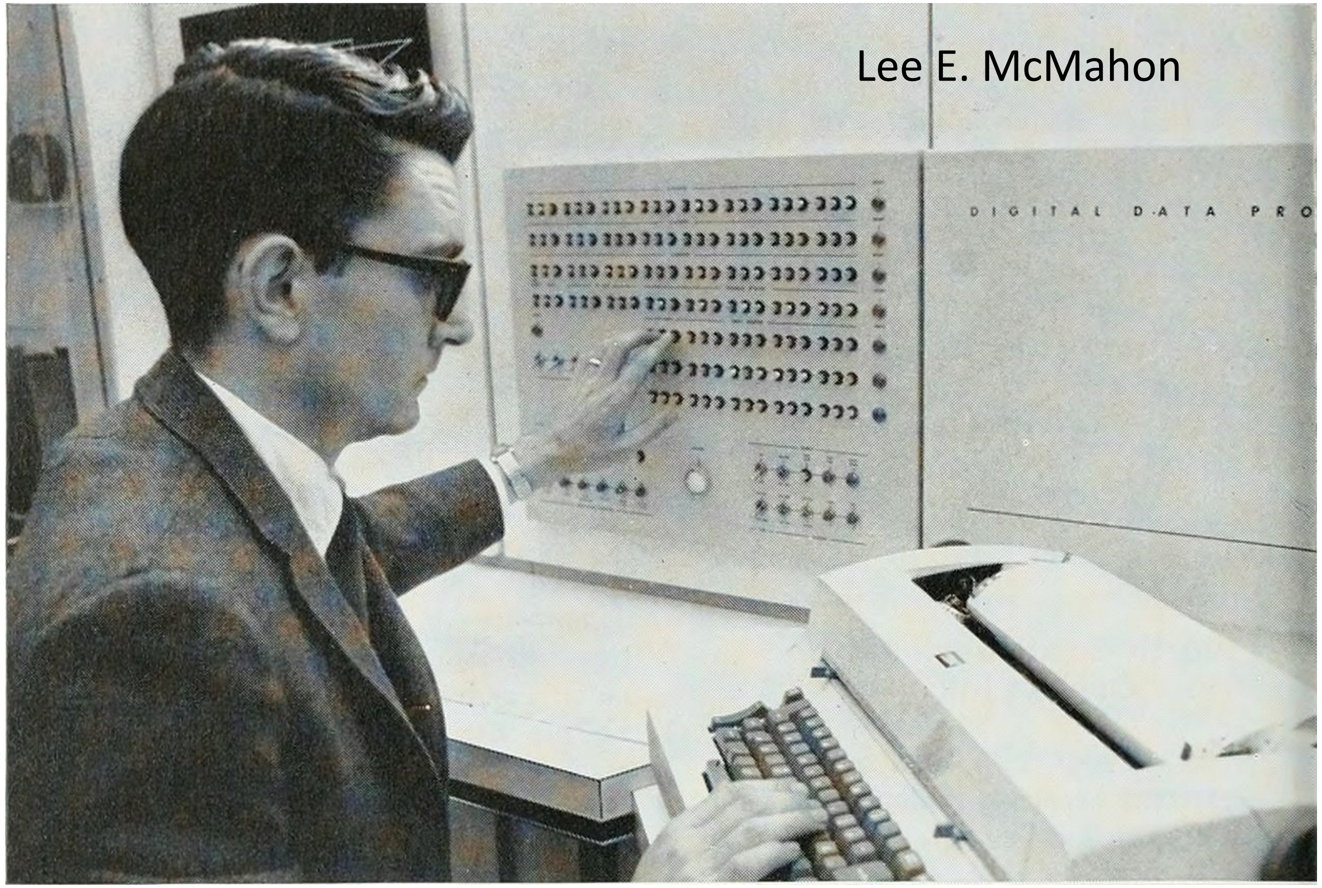
SED – Stream EDitor

- Unix utility for parsing and transforming text
- developed in 1973 by Lee E. McMahon of Bell Labs
- based on ed ("editor", 1971) and qed ("quick editor", 1965)
- was first to support regular expressions

Supported features

- insertion
- deletion
- substitution
- supports regular expression

```
$ sed 's/find/replace/' file
```



Lee E. McMahon

SED – Stream EDitor

file.txt

cat is great
cat is fluffy
cat is gorgeous, I love cat, cat is the best

"s" stands for substitution

pattern to find

replacement string

```
marina@vm:~/SPLab $  sed 's/cat/dog/' file.txt
dog is great
dog is fluffy
dog is gorgeous, I love cat, cat is the best
marina@vm:~/SPLab $
```

by default, only the first occurrence of the pattern in each line is replaced

```
marina@vm:~/SPLab $  sed 's/cat/dog/2' file.txt
cat is great
cat is fluffy
cat is gorgeous, I love dog, cat is the best
```

'2' means to replace only the second occurrence of the pattern in each line

```
marina@vm:~/SPLab $  sed 's/cat/dog/g' file.txt
dog is great
dog is fluffy
dog is gorgeous, I love dog, dog is the best
```

"g" stands for global substitution, i.e., replacement of all the pattern occurrences

```
marina@vm:~/SPLab $  sed 's/cat/dog/2g' file.txt
cat is great
cat is fluffy
cat is gorgeous, I love dog, dog is the best
```

"2g" means global replacement, starting from second occurrence of the pattern

```
marina@vm:~/SPLab $  sed '3 s/cat/dog/g' file.txt
cat is great
cat is fluffy
dog is gorgeous, I love dog, dog is the best
```

"3" means change only the third line

```
marina@vm:~/SPLab $  sed '1,2 s/cat/dog/g' file.txt
dog is great
dog is fluffy
cat is gorgeous, I love cat, cat is the best
```

"1,2" means the range of lines to change

```
marina@vm:~/SPLab $  sed '2,$ s/cat/dog/g' file.txt
cat is great
dog is fluffy
dog is gorgeous, I love dog, dog is the best
```

"$" means the last line of the input file

SED – Stream EDitor

file.txt

```
cat is great
cat is fluffy
cat is gorgeous, I love cat, cat is the best
```

"-n" stands for no output

"p" stands for printing the changed lines

```
marina@vm:~/SPLab $  sed -n '3 s/cat/dog/gp' file.txt
dog is gorgeous, I love dog, dog is the best
marina@vm:~/SPLab $
```

```
marina@vm:~/SPLab $  sed -i -n '3 s/cat/dog/gp'  file.txt
marina@vm:~/SPLab $  cat file.txt
dog is gorgeous, I love dog, dog is the best
marina@vm:~/SPLab $
```

"-i" means in-place editing, i.e., the output would not be printed on the screen, but would be saved into the input file.

```
marina@vm:~/SPLab $  sed -i'.orig' -n '3 s/cat/dog/gp'  file.txt
marina@vm:~/SPLab $  cat file.txt
dog is gorgeous, I love dog, dog is the best
marina@vm:~/SPLab $  cat file.txt.orig
cat is great
cat is fluffy
cat is gorgeous, I love cat, cat is the best
marina@vm:~/SPLab $
```

"-i.orig" means in-place editing, and the original file copy would be saved in <file name>.orig file

SED – Stream EDitor

file.txt

```
first
second
third
fourth
```

"2" stands for line number to be deleted

"d" stands for delete lines

```
marina@vm:~/SPLab $  sed '2d' file.txt
first
third
fourth
marina@vm:~/SPLab $
```

```
marina@vm:~/SPLab $  sed '$d' file.txt
first
second
third
marina@vm:~/SPLab $
```

"$" stands for last line

```
marina@vm:~/SPLab $  sed '1,3d' file.txt
fourth
marina@vm:~/SPLab $
```

"1,3" stands for lines range

```
marina@vm:~/SPLab $  sed '/ir/d' file.txt
second
fourth
marina@vm:~/SPLab $
```

"/ir/" stands pattern to be deleted (in our example, first and third have this pattern)

SED – Stream EDitor

file.txt

```
first


second

third
fourth
```

"/^$/" stands blank (i.e., empty) line

```
marina@vm:~/SPLab $ sed '/^$/d' file.txt
first
second
third
fourth
marina@vm:~/SPLab $
```

```
marina@vm:~/SPLab $ sed '/^$/d;G' file.txt
first

second

third

fourth

marina@vm:~/SPLab $
```

"G" stands for inserting one blank line after each line of the input file

```
marina@vm:~/SPLab $ sed 's/^/line: /' file.txt
line: first
line:
line:
line: second
line:
line: third
line: fourth
marina@vm:~/SPLab $
```

SED – Stream EDitor

file.txt

```
Linux
Solaris
Ubuntu
Fedora
RedHat
```

sed supports multiple commands separated by ';'

```
marina@vm:~/SPLab $ sed 's/u/ /g;s/e/,,/g' file.txt
Lin x
Solaris
Ub nt
F,,dora
R,,dHat
marina@vm:~/SPLab $
```

replace 'u' by space, and also replace 'e' by ',,'

```
marina@vm:~/SPLab $ sed 's/^.//;s/.$//' file.txt
inu
olari
bunt
edor
edHa
marina@vm:~/SPLab $
```

remove first and last characters of each line

```
marina@vm:~/SPLab $ sed –E 's/.{3}//' file.txt
ux
aris
ntu
ora
Hat
marina@vm:~/SPLab $
```

remove first occurrence of three characters

'-E' Interpret regular expressions as extended (modern) regular expressions rather than basic regular expressions

SED – Stream EDitor

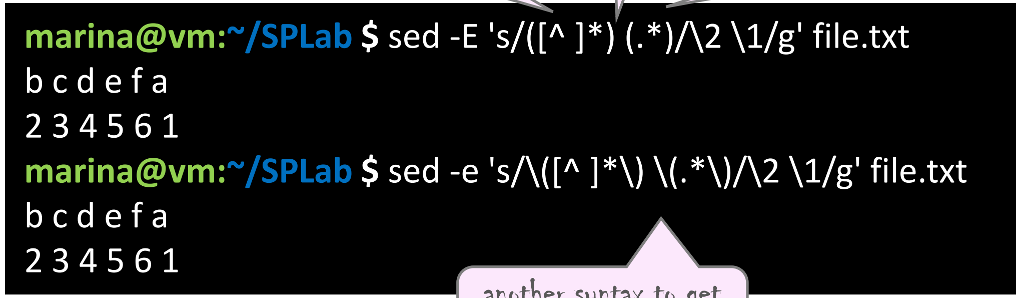
```
a b c d e f
1 2 3 4 5 6
```

file.txt

([^]*) means any character except space, any number of times. This matches the first word of the line. This is the first part of our regular expression.

note that between two parts of regular expressions exists a space character

(.*) means any character, any number of times. This matches the rest of the line. This is the second part of our regular expression.

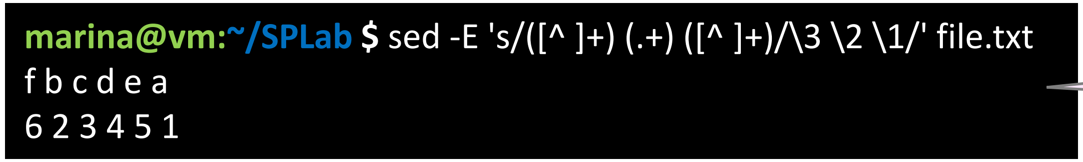


```
marina@vm:~/SPLab $ sed -E 's/([^ ]*) (.*)/\2 \1/g' file.txt
b c d e f a
2 3 4 5 6 1
marina@vm:~/SPLab $ sed -e 's/\([^ ]*\) \(.*\)/\2 \1/g' file.txt
b c d e f a
2 3 4 5 6 1
```

another syntax to get the same output

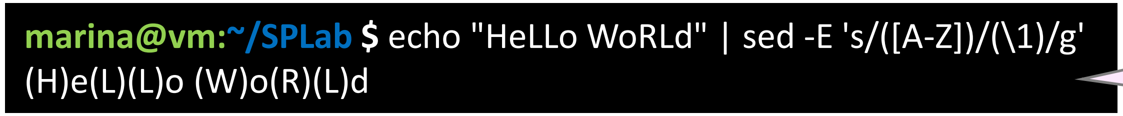
/\2 \1/ means that the output line (that replaces the appropriate input line) consists of the rest of the line, then space, then the first word of the line.

So, we move the first word of each line to its end.



```
marina@vm:~/SPLab $ sed -E 's/([^ ]+) (.+) ([^ ]+)/\3 \2 \1/' file.txt
f b c d e a
6 2 3 4 5 1
```

This script switches the first and the last words of each line.



```
marina@vm:~/SPLab $ echo "HeLLo WoRLd" | sed -E 's/([A-Z])/(\1)/g'
(H)e(L)(L)o (W)o(R)(L)d
```

/(\1)/ means output '(', then the first regular expression value (in our case, it is any capital letter), then ')'. Since we use 'g', we do this for every capital letter in the line.

Extended System Programming Laboratory

Lecture 8 – AWK script language, SED

Dr. Marina Kogan-Sadetsky