

Computer Systems

6. SED - AWK



Review

- Computers, informations, number representation, code- writing
- Architecture, client-server role, file systems
- Base commands, foreground and background processes
- I/O redirection, filters, regular expressions
- Variables, command substitution, arithmetical, logical expressions
- Control structures

What comes today?

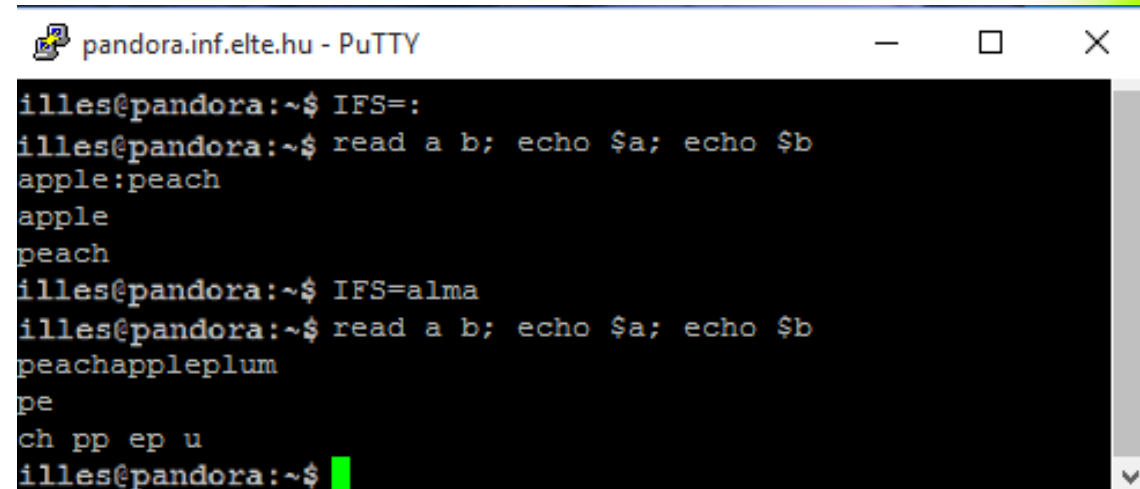
- More filtering!
 - SED
- More program writing possibilities!
 - AWK

Input – Output commands

- Most of the commands write to the standard output
 - echo command, -n there is no new line
- Read from the standard input:
 - read command
 - E.g.: read apple#reads into apple variable till enter
 - read pear; echo \$pear # ok
 - echo hello|read a; echo \$a # a is empty, because read is not a filter
 - line command - filter
 - It reads a whole line from the standard input and writes to the standard output
 - echo hello| line

IFS - Internal Field Separator

- It is part of several Unix implementation!
 - BASH contains it.
- To define a new delimiter character instead of the default (space, tab) one!
 - It is recommended to choose a single character!
 - IFS=: #double-dot is the new delimiter



The screenshot shows a terminal window titled "pandora.inf.elte.hu - PuTTY". The user "illes" is at the "pandora" host. The terminal shows the following commands and output:

```
illes@pandora:~$ IFS=:
illes@pandora:~$ read a b; echo $a; echo $b
apple:peach
apple
peach
illes@pandora:~$ IFS=alma
illes@pandora:~$ read a b; echo $a; echo $b
peachappleplum
pe
ch pp ep u
illes@pandora:~$
```

Make more filtering...

- What can we filter up to now?
 - To cut out characters or fields from lines (cut)
 - Whole lines (grep, regular expressions)
- Is it enough?
- What are we not able to do?
 - A lot of things... e.g. we are not able to search, to replace part of texts inside a line!

SED – Stream EDiTOR

- Filter – it modifies the input lines with the given operations (edit)
- Task: Complex substitutions, replaces working on the lines arriving on the standard input, result is written on to the standard output. It takes each of the lines (cycle), and executes the commands on them one after the other, the modified lines are written on to the standard output!
- Example: `cat class | sed 's/3/9/g'`
 - Find each 3 (g) and replace them to 9.

sed options

- -n, There is no automatic writing to the standard output. Only the sed's own writing instructions write out to the output.
- -f scriptfile, The program, the sed script is in the file given by the parameter. Typically there is only one command in a line, but with ; as a delimiter we can give several commands in the same line.
- -e There are several scripts in the command line of sed. There is no need of it in the case of one script.

sed script commands

- Several commands inside a direct script command:
 - ; Semicolon between the commands!
 - Example: `cat class | sed 's/3/9/g ; s/9/21/'` # Replace in the lines of class file each 3 to 9 and after it the first 9 to 21 in each line!
 - Commands may be written into different lines, we get a second prompt in this case!

```
$ cat class | sed 's/3/9/g  
>s/9/21/' [enter]  
john 21 4 2 5 4  
george 2 21 4 5 9  
kate 4 21 2 9 9  
$
```

Writing a sed script

- In the command line we may write several commands, but it is not suggested!
- Write a script!
 - `chmod +x sedexample`
 - Execution: `sedexample class`

```
teszt@pandora:~/> cat sedexample
#!/bin/sed -f
#
s/3/9/g #3-9 replace all 3 to 9
s/9/10/ #9-10 replace – only the first 9
```

sed important commands I.

- Sed command syntax: [address] s /pattern/new_pattern/[marker]
 - Task: searches for a pattern and replace it with a new one!
- Typically the pattern is a regular expression and we are searching for it.
- The address, to which the sed command is referred may be a regular expression, a number or an interval 1,10 s/.../.
- \$ symbol means the last line # 5,\$s/apple/pear/
- ! symbol, means negotiation
 - 2!s/apple/pear/ # replacement in each line
except the second line

Other address definition

- If it is not given, then the s command refers to each line!
- N – refers to the N. line, e.g.: 4/3/9/g
replacement in the 4th line
- x~y – line number x and each y-th after it!
 - Example: 3~2/3/9/g # from the 3. lines each 2nd one!
- X+y – from the x-th lines y lines!
 - Example: 3+2/3/9/g # 2 lines after the 3rd one
- To learn the full address mechanism let's see the reference!

searching-replacing command marker values

- Marker values:
 - n: ordinal number, the replacement should be started at the n. pattern, if it is not given the default value n=1. If n is greater than the last occurrence, nothing happens!
 - g: All of the patterns must be replaced.
 - p: It lists out the actual line! (pg can be used together)
 - w file: Save the actual line into the file (appends)!
 - r file: It reads in the content of the file to the input!

sed important commands II.

- `/new_pattern &/` The new pattern will be added to the beginning of the pattern
 - Example: `echo RealMadrid | sed 's/RealMadrid/tally-ho &/'` #tally-ho RealMadrid
 - `&` symbol means the „old” pattern, it can be anywhere!
- `\n` usage,
 - `\1` the 1st regular expression,
 - `\` it masks the effect of the following metacharacter (`\.ali`)
 - `\n`, insert a new line (`\ >\2/`), there is no `n` in the example because we insert it directly as a command

```
illes@panda:~$ echo .ali 4 Ali baba|sed 's/\.ali \([0-9][0-9]*\) \(.*)\^1. part\  
> \2/' (enter)  
4. part  
Ali baba  
illes@panda:~$
```

sed important commands III.

- Deleting: [address-interval]d
 - d command deletes the lines given by the address-interval
 - E.g.: `cat class|sed '1d; s/3/9/pg; s/9/21/'` #it deletes the 1. line than the following 2 commands are executed on the other lines.
- Append: a
 - E.g.: `cat class|sed 'a\apple'` # an apple will be inserted as a new line after each line
- Include: i
 - E.g.: `cat class|sed '2,3i\apple'` # apple is inserted before the 2,3 lines

sed important commands IV.

- y – character replacement pattern: `echo pilote|sed ,'y/lo/ra/' #pirate`
 - The number of characters in the patterns must be equal! (similar to tr)
- q – quit, after a given address sed quits
- :label – label making
- b label – jumps without a condition
- t label – if there was a succesful replacement a conditional jump is executed

Sed example I.

- 1. example:

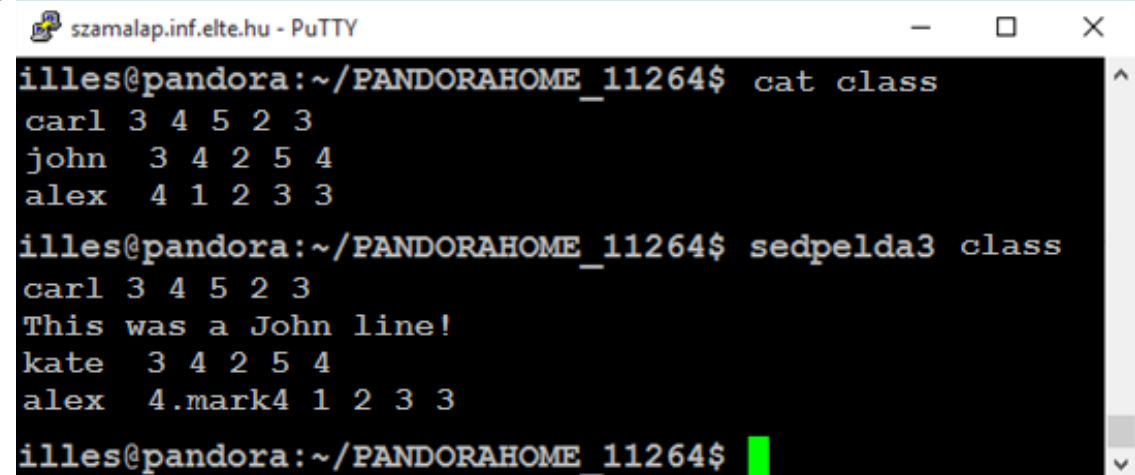
```
#!/bin/sed -f
#
# In the first and second line after numbers we write „wish”
1,2s/\([1-9][1-9]*\)/&.wish\1/g # after all numbers!!!
# After the second line it quits!
2q
```

SED program quotation marks

- You have to give the program between quotation marks for SED!
- It may be a single quote ' or a quote mark "!
- There is a difference!
- `x=John; echo George is skillful! | sed "s/George/$x/"`
 - Result: John is skillful!
- `x=John; echo George is skillful! | sed 's/George/$x/'`
 - Result: \$x is skillful!

Sed example II.

```
#!/bin/sed -f
#
s/john/kate/ # john is replaced to kate
t johnline # conditional jump, if there was a succesful replacement
# in each line we write after the numbers
2,$s/\([1-9][1-9]*\)ate/&.mark\1/
#replacement only at the first number
# 2,$ gives the address-interval, $ means the last line
b end # jump without a condition
#johnline label
:johnline
i This was a John line!
# includes before the actual line!!
:end #end label
```



```
szamalap.inf.elte.hu - PuTTY
illes@pandora:~/PANDORAHOME_11264$ cat class
carl 3 4 5 2 3
john 3 4 2 5 4
alex 4 1 2 3 3
illes@pandora:~/PANDORAHOME_11264$ sedpelda3 class
carl 3 4 5 2 3
This was a John line!
kate 3 4 2 5 4
alex 4.mark4 1 2 3 3
illes@pandora:~/PANDORAHOME_11264$
```

SED references

- The slides do not contain each details!
- GNU SED description- a full reference
 - <https://www.gnu.org/software/sed/manual/>
- On the net you can find a lot more!

What else comes after sed? AWK

- Sed is very good, very useful, but ...
 - Typical usage is pattern replacement.
 - To work within a line is restricted.
 - There is no arithmetical possibility. (There is something but it is not a real good one...)
 - Control structures are missing. (There is jump, conditional jump...)
 - ...
- Out of the woods: AWK

AWK

- Alfred V. **A**ho, Peter J. **W**einberger, Brian W. **K**ernighan
- Deficiency of shell in text processing
- Practically it has similar possibilities as in C program language
- Typical filter
- Often it is used as a shell script element
- Text processing line by line, executable program
- `awk -gawk` (GNU AWK)
 - Reference: <http://www.gnu.org/software/gawk/manual/gawk.html>

AWK working, structure

- Place of command: whereis awk # /usr/bin/awk, ...
- awk is waiting the data as a parameter or from the standard input.
- The data is processed line by line. Initial block before the first line and after the last.
- Command blocks are instructions between {} symbols
- Before the command block a pattern may be defined:
Example: /f.*/
 - /regular expression/
 - Pattern contains a logical expression: \$2 == „alma”

AWK usage

- Program directly, just like a parameter
 - `awk '{ print ;}' datafile`
 - The program refers to each lines, writes them out
- Program is in a file
 - `awk -f programfile datafile`
 - Often the awk programfile itself is the command!
 - `#!/usr/bin/awk -f`
 - This is the first line
 - E.g.: `$ awk_program1 datafile` # typical structure of the command
- Filter
 - `command1 | awk-commandfile`

Elements of input lines

- \$0 – the whole line
- \$1, \$2, ... - the first, second, ... element of the line
- Field separator: FS (default: space or tab)
 - You may use several characters: FS=„abc”
- The number of fields in a line: NF
- Line separator character: RS (default: new line)
- The number of lines read up to now: NR
 - In the case of several input files: FNR, ordinal number of a file

AWK example

- Programcode
:

- Execution

```
$cat class | firstawk
Program is starting!
The content of john line: john 3 4 2 5 4
It is the end!
$
```

```
#!/usr/bin/awk -f
# Starting block!
BEGIN {
    print „Program starts!";
    # here comes further commands which are
    # needed at the beginning, e.g. field separator
    # FS=":"
}
# In each lines it searches for john and writes it when
# it finds it
/john/ {
    print „The content of john line: "$0;
}
# Ending
END {
    print "It is the end!";
}
```

AWK elements of output lines

- print instruction: e.g.: `print $0` #writes out the whole line
- OFS variable
 - Output Field Separator
- ORS variable
 - Output Row Separator

```
BEGIN {  
    OFS=„:”;  
    ORS=„End.\n”;  
}  
  
    print „Apple”, $0;  
}
```

AWK variables, expressions

- Built in variables are named with capital letters! E.g.: NF
- There is no type: name=value, a value can be number, „text”
 - `v="apple"; print v; #apple`
- Text concatenation: there is no operator, simply you have to write the variables one after the other!
 - E.g.: `{v=„apple";f=„tree"; print "5 "v f;} #5 appletree`
- Automatic conversion, if it „feels” it is needed
 - `{v="apple";f=„tree";print v+f} #0`
 - `{v="3apple";f="2tree"; print v+f} # 5`

AWK arrays

- `t[0]=3`, etc. We define the elements with index.
- The type of the elements of the array can be different.
- Actually they are associative arrays:
 - `t[„one”]=1; print t[„one”];`
- Length of an array: `length(t)`
- If there is an index in the array: 4 in `t`
- Deletion of an element: `delete t[4]`
- Multidimensional arrays: `tt[1,2]=3;`

AWK operations, functions

- +, -, ++, --, *, /, %, ... - ordinary operators
- Logical value like in C
- !=, ==, <, > - logical operators
- && logical and, || logical or, ! negation
- ~, !~ pattern fitting, not fitting
 - The right hand side operand may be a regular expression as well /.../.
 - \$0 ~ /reg.exp/ form is shortened this way: /reg.exp/ , that is why you can use it before awk block just so!
- ** or ^ raising to power, e.g.: 2**3 #8

AWK mathematical functions

- Important built in functions:
 - `int(number)` # integer part of a number, print `int(3.7)` #3
 - `sqrt(number)` # square root
 - `sin(x), cos(x)` # sin, cos functions, x is given in radian!
 - `rand()` # gives a random number in the]0,1[interval
 - `x=int(10*rand())` # 0,1,...9
 - `exp(x), log(x)` # $e^{**} x$, $\ln(x)$
 - `atan2(x,y)` # arc. tangens x/y

Other useful awk functions

- `{v=length("appletree");print v}` # 9 the length of text
- `split` – text splitting
 - E.g.: `split(„alan:paul:robert”, n, „:”); print n[1] #alan`
- `sprintf(sz, „pattern”,variables);` # like `sprintf` in C
- `system(„command”)` – op. System command execution
 - E.g.: `{system("date > date.txt") }`
- File reading:
 - `getline <"/home/data"; print $0;`
 - Next `getline`, next line.

AWK control structures

- Branch
 - `if (x % 2 == 0) print "x is even"; else print "x is odd";`
- Multi-branch
 - `Switch (c) { case „a”:... }; # like in C`
- Cycles are similar to ones in C
 - `while (exp) instruction`
 - `do ... while (exp)`
 - `for(exp1;exp2;exp3) instructions`
 - `for (i in array)`
 `array[i] processing`

Example

```
#!/usr/bin/awk -f
#
# Task: to calculate the averages in a
class
# execution: awk_average class
# BEGIN pattern is executed firstly
BEGIN {
    print „usage of cycle-branch!";
    if (ARGC !=2)
    {
        print „Give a filename!";
        exit 1;
    }
}
```

```
{
    print „Data of a student:" $0;      # writes out the actual line
    # calculation of the average of the student
    # firstly we calculate the sum of the marks
    sum=0; # starting value is 0
    for (i=2;i<=NF;i++)                # we take all the marks one by one
    {
        sum+=$i;                        # actual mark is given to the sum
    }
    # remember the i. average
    averages[NR]=sum/(NF-1);
    print $1 " average is: " averages[NR];
}
# END pattern is executed at the end of the program
END {
    print NR " students average was calculated!";
}
```

AWK summary

- Help: Google: awk, gawk
- BEGIN block, it is executed before the line by line block execution
- END block, it is executed after the line by line execution block
- Pattern {line by line block}
- These slides summarized the most important elements of the command!
- Full reference: <https://www.gnu.org/software/gawk/manual/>

Thank you!

