Data Processing: Formats and Tools

(part 2) a topic in

DM565 – Formal Languages and Data Processing

Kim Skak Larsen

Department of Mathematics and Computer Science (IMADA)
University of Southern Denmark (SDU)

kslarsen@imada.sdu.dk

September, 2023

Some Relatively Simple Command-Line Tools SDU

sort

uniq

tr

cut

paste

join

head/tail

These are all linux filters, i.e., they do not change their input, but produce output on **stdout**, and can be used in pipes, just like the more complicated grep, sed, and (g)awk.

Command-Line Tools: sort SDU

Options for common issues (selected)

- ignore blanks
- ignore case
- sort numerically, alphabetically, by month, version numbers, . . .
- specify which field to sort on
- specify delimiters
- reverse

Example

Sort numerically on the 5th column, showing the larger numbers (file sizes) first:

```
> ls -l | sort -n -r - k5
```

Command-Line Tools: uniq

“filter out adjacent matching lines” – often used after sort

Options for common issues (selected)

ignore case

print only unique or duplicate lines

consider only the first or last some number of characters

consider only some fields

count duplicates

Example

Remove duplicates, ignoring the first field:

```
> cat myfile
41 1 2 3
42 1 2 3
43 3 2 1
> cat myfile | uniq - f1
41 1 2 3
43 3 2 1
>
```

Command-Line Tools: tr

“translate or delete characters”

Options for common issues (selected)

delete characters in a given set

delete consecutive duplicates of a character, leaving one occurrence translate by specifying two character sequences of the same length

Example

Change and delete some characters:

```
> cat myfile
41 Forty + one
42 Forty + two
43 Forty + three
> cat myfile | tr 'F + ' 'f - ' | tr -d ' [: digit :] '
forty - one
forty - two
forty - three
>
```


Command-Line Tools: cut

"remove sections from each line of files"

Options for common issues (selected)

- select numbered bytes
- only keep certain characters
- select some fields
- specify delimiter

specify output delimiter

Example

Change input/output delimiters and keep columns 2 and 3:

```
> cat myfile
x :41: one
y :42: two
> cat myfile | cut -d : -- output - delimiter = ' ' -f2 ,3
41 one
42 two
>
```


Command-Line Tools: paste

"merge lines of files"

Options for common issues (selected)

specify delimiter (default is **\t**)

serial mode (the lines in each file will be concatenated into one line)

Example

Paste lines using space instead of the default tab:

```
> cat myfile1
41
42
> cat myfile2
forty - one
forty - two
> paste -d ' ' myfile1 myfile2
41 forty - one
42 forty - two
>
```


Command-Line Tools: join

"join lines of two files on a common (sorted) field" – similar to dbms

equi-join **Options for common issues (selected)**

ignore case

specify delimiters

specify join field

Example

Join lines on a common field (first field is default):

```
> cat myfile1
42 A
42 C
43 B
> cat myfile2
41 X
42 Y
> join myfile1 myfile2
42 A Y
42 C Y
>
```


Command-Line Tools: head/tail

“output the first/last part of files”

Options for common issues (selected)

- specify the number of lines
- specify start line
- specify bytes instead of lines

Example

Print the first two lines of the last 10 (default) lines:

```
> seq 50 | tail | head -2
41
42
>
```

stream-oriented, non-interactive, text **ed**itor

Specify patterns (similar to grep),

but change (edit) the matching lines,

not interactively, but via a script – a sequence of *commands*.

Changes are applied to a line successively, i.e., after one modification, the next change (to the same line) is applied to the modified line.

A command consists of *address* information and an *action*; the address information can restrict the lines affected to some subset.

Command-Line Tools: sed 

Using standard specification syntax, a command has the form

[address[,address]][!]command[arguments]

(Unfortunately, syntax varies a lot; in manuals and similar documents, **[...]** is used instead of **(...)?**, i.e., zero or one occurrence.)

An address can be a *line number* (**$** can be used to mean “the last line”) or a *pattern*, which is simply a regular expression (grep-style) surrounded by slashes.

Two addresses can be used to specify an interval and **!** negates the address information.

Command-Line Tools: sed

Examples: On Address Specification

d delete all lines

42d delete line 42

1,10!d delete all lines except lines 1–10

1,/^$/d delete from line 1 through first blank line

/^$/,$d delete from the first blank line through the last line **/42/,42d** delete from the first line containing the number 42 through line 42

/^Proof/,/qed$/d

delete from the first line starting with “Proof”

through the first line ending with “qed”

So, on command-line, one writes, for instance,

```
cat myfile | sed ’ 42! d ’
```


Command-Line Tools: sed

Useful Options

A filename can be given as argument to **sed**; if omitted, input comes from **stdin**.

-n suppress output, unless an explicit print command is issued (see later) **-f** the next argument is a file name containing a script

-e the next argument is a command

(if necessary to avoid confusion with a file name argument

-E use regular expression syntax as for **grep -E**

#n as the first line of a script is an alternative to **-n**

The print command **p** is used exactly like **d**.

Unless **-n** is used, printed lines will come out twice.

However, **p** together with **-n** can be useful when

there are several commands.

Command-Line Tools: sed

Substitute

The substitute command **s** takes arguments and optionally flags. Leaving out the address specification, the syntax is

s/pattern/replacement/[flags]

A flag can be a number **i**, indicating that it is the **i**th occurrence that should be replaced. The flag **g** (global) indicates that all occurrences should be replaced, and **p** prints.

The pattern is just a regular expression (grep-style).

The replacement string can contain special characters:

\d the **d**th group from the pattern (grep-style)

& the entire string matched by the regular expression

**** backslash

\& ampersand

Examples

We can fix a spelling error by

sed -E 's/iether/either/g'

We can translate the second occurrence of "datalogi" on each line by **sed -E 's/datalogi/computer science/2'**

We can change "datalogi" to "datalogistudiet" by

sed -E 's/datalogi/&studiet/g'

If a file contains two columns, separated by one colon, we can switch the two columns using

sed -E 's/(.*):(.*)/\2:\1/'

Regular expressions match strings as long as

Command-Line Tools: sed SDU

Transform

The sed-equivalent of **tr** is the command **y**, performing one-to-one character-to-character replacement (can be prefixed by addresses).

Example

sed 'y/123/234/' will increment all the digits 1, 2, and 3.

Quit

The quit command **q** stops processing when (if) the single address specification is reached.

Example

sed ’10q’ will terminate after the first 10 lines have been processed.

Command-Line Tools: sed

Other Commands

There also command for inserting and appending before or after a line, changing lines in a fixed manner, and more, but the syntax becomes more

cumbersome and it might be nicer to use other tools.

Multiple Commands

sed is most convenient for simple changes, possibly by piping into another **sed**. However, multiple commands are possible, with the following somewhat odd requirements.

Format:

```
[ address [ , address ]][!]{
command [ arguments ]
.
.
command [ arguments ]
}
```

where the opening brace must be last on a line and the closing alone on a line.

Command-Line Tools: awk

Aho, **W**einberger, **K**ernighan

We will use gawk (GNU awk), but usually just say “awk”.

- a full programming language
- can be used on command-line or via script
- handles fields nicely (not just lines)
- understands numbers (not just text)
- C-like syntax, but also grep-like patterns
- an awk script is a sequence of **pattern {action}**

Command-Line Tools: awk

Patterns

BEGIN and **END** are special patterns that only match at the begining or end of a file, respectively, used for initialization and announcement of results regular expressions enclosed in **/.../**

C-like conditionals with comparisons and logical connectives: **&&**, **||**, **! s ~ r** is true if the string **s**

matches the regular expression **r** (asymmetric!); use double quotes around the arguments if they contain special symbols arithmetic and built-in mathematical functions

Actions

C-like statements

if there is no action, lines matching the pattern

are printed (sed-style)

Command-Line Tools: awk

Example

```
ls | gawk '
    BEGIN { print " List of tex files :" ; count = 0}
    /\. tex$ / { print ; count += 1 }
    END { print " Total :" , count , " files " }
    '
```

gives output (in my example)

```
List of tex files :
2023 lecture . tex
def - colors . tex
lecture . tex
preamble . tex
Total : 4 files
```

Alternatively, one can print with C's **printf**.

Command-Line Tools: awk

Field Manipulation

RS is the record separator (default newline)

FS is the field separator (default other maximal whitespace sequences) **OFS** is the output field separator (default space)

NR is the number of the current record (line)

NF is the number of fields in the current record

$1, **$2**, . . . , **$0** are the fields and the entire record

Examples

cat myfile | gawk '{ print NR, $2 * $3, $(NF-2)}' prints the line number, the value of fields 2 and 3 multiplied together, and the third to last field.

We can assign to the field variables as in

cat myfile | gawk '{ $1 = $2; $2 = ""; print }'

Command-Line Tools: awk

Built-In Functions

The linux command **wc** can be realized as follows:

```
gawk '
    BEGIN { OFS = "\ t " }
    { chars += length ( $0 ) + 1; words = words + NF }
    END { print NR , words , chars , FILENAME }
    ' myfile
```

+1 since **$0** does not include newline.

Variables are initialized to the empty string or zero as

appropriate. Also

- string concatenation – placing strings next to each other separated by blanks **substr(s, m, n)** – **n** characters from position **m** in **s**
- advanced **split** operations
- **system** calls and **exit**
- dictionaries, including **ARGV** (**ARGC** for the length of **ARGV**)

Command-Line Tools: awk

Control Structures

- if-then-else
- while and do-while
- for-loops C-style and **for (key in array) { ... }**

Command-Line Tools: awk

Example

Print input in reverse order:

```
cat myfile | gawk '
    { line [ NR ] = $0 }
    END {
        for ( i = NR ; i > 0; i -= 1) {
            print line [ i ]
        }
    }
    '
```


Command-Line Tools: awk

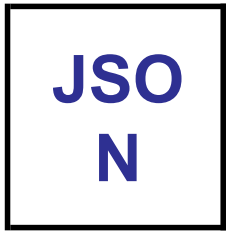
Useful Options

Filenames can be given as argument to **gawk**; if omitted, input comes from **stdin**.

-f the next argument is a file name containing the program **-F** the next argument is to be used as input field separator **-v var=val** initialize a variable prior to execution

JSON-Like Formats: JSON

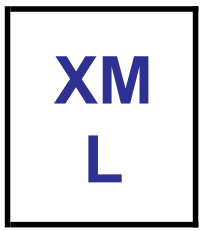
Java**S**cript **O**bject **N**otation



```
{ " animals " : [
      {
          " name " : " Panda " ,
          " cuteness " : 1.0 ,
          " colors " : [ " white " , " black " ]
      } ,
      {
          " name " : " Panther " ,
          " cuteness " : 0.7 ,
          " colors " : [ " black " ]
      }
   ]
}
```


JSON-Like Formats: XML

e**X**tensible **M**arkup **L**anguage



```
< animals >
    < animal >
        < name > Panda </ name >
        < cuteness >1.0 </ cuteness >
        < color > white </ color >
        < color > black </ color >
    </ animal >
    < animal >
        < name > Panther </ name >
        < cuteness >0.7 </ cuteness >
        < color > black </ color >
    </ animal >
</ animals >
```


JSON-Like Formats

There is more to both formats.

The essence is that it is named parentheses structures expressing records (attribute/value pairs) and sequences (arrays, lists).

There are many variants of XML (HTML) with similar structure.
Command-Line tools can to some extent be used for data discovery, and possibly simple code execution.

To get full power, use a programming language with an appropriate package. Packages read json/xml files and deliver data in native formats.

JSON-Like Formats: Python Example

```
> cat animals . json
{ " animals " : [
      {
          " name " : " Panda " ,
          " cuteness " : 1.0 ,
          " colors " : [ " white " , " black " ]
      } ,
      {
          " name " : " Panther " ,
          " cuteness " : 0.7 ,
          " colors " : [ " black " ]
      }
   ]
}
>
```

JSON-Like Formats: Python Example

```
# Prints
# {
# " a ": 1 ,
# " b ": 2 ,
# " c ": 3 ,
# " d ": 4 ,
# " e ": 5
# }
# Panda

import json

# Data in program for testing
json_data = ' {" c ": 3 , " d ": 4 , " a ": 1 , " b ": 2 , " e ": 5} '

parsed_json = json . loads ( json_data )
print ( json . dumps ( parsed_json , indent =4 , sort_keys = True ))

# It is just dictionaries and lists
with open ( ' animals . json ' , 'r ') as f :
        animals_dict = json . load ( f )
print ( animals_dict [ " animals " ][0][ " name " ])
```