# CS6

# Practical System Skills

**Fall 2019 edition**

Leonhard Spiegelberg
lspiegel@cs.brown.edu

# 11 Regular expressions

**CS6** Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

# 11.01 Regular expressions

A *regular expression* is a string search pattern.

Regular expressions (short regex) enable you to do the following:

1. test whether a string matches a pattern
2. replace substrings matching a pattern in a string
3. find the position of or extract a substring which matches a pattern

# 11.01 Regular expressions

Where are regular expressions used?

⇒ text editors to search for text

⇒ validation (e.g., webforms)

⇒ data extraction/manipulation

# 11.02 grep

`grep = global regex print` (with **g**, **r**, **p** in bold)

```
grep [OPTIONS] PATTERN [FILE...]
```

⇒ prints lines matching a pattern, i.e. `grep` goes over each line of a file (or stdin when `-` is specified) and prints the line if the pattern is found within the line

# 11.02 grep - basic examples

⇒ the easiest regular expression which we can write is just made up of regular `[a-zA-Z0-9]` characters (i.e. abc and numbers).

⇒ grep searches whether pattern is contained within each line and prints the line then out!

example.txt

```
hello
world
is a classical
first program
to write
in any language!
```

Examples:

| `grep world example.txt` | `grep a example.txt` | `grep "a classical" example.txt` |
|---|---|---|
| **world** | is **a** cl**a**ssic**a**l<br>first progr**a**m<br>in **a**ny l**a**ngu**a**ge! | is **a classical** |

⇒ matching part of line in red here, lines which contain pattern are printed using grep

# 11.02 grep - options

⇒ grep has exit status 0 if at least one match was found, 1 else

⇒ `-o` or `--only-matching` to print only the matching part of a line

⇒ `-q` or `--quiet` to not display lines

Example:

```
echo "some string" | grep elephant -; echo $?
1
```

no output here, because no match

```
echo "some string" | grep -q string -; echo $?
0
```

no output here, because suppressed via -q

# 11.02 grep - options

```
grep -o program example.txt
program
```

```
grep -o an example.txt
an
an
```

each match printed on separate line

example.txt

```
hello
world
is a classical
first program
to write
in any language!
```

# 11.03 writing regular expressions

⇒ regular expressions can be written in their own, specific mini-language

- → defacto there are multiple ways to specify regular expressions, in CS6 we'll learn **POSIX BRE** (Basic Regular Expressions) and **POSIX ERE** (Extended Regular Expressions) syntax.
- → PCRE (Perl compliant regular expressions) is a superset of POSIX BRE/ERE.

# 11.03 Basic regular expressions

⇒ **first:** basic regular expressions (BRE)

⇒ special characters are `[ ] \ ^ $ . *`

⇒ to escape them use `\`, i.e. `\.` to have `.` instead of special character `.`

# 11.03 Basic regular expressions

**Quantifiers:**

⇒ a quantifier after an item specifies how often the item may occur

| | |
|---|---|
| `*` | preceding item may occur 0 or more times |
| `\{m\}` | preceding item must occur exactly `m` times. |
| `\{m,n\}` | preceding item must occur at least `m` times, but no more than `n` times. |

# 11.03 Quantifiers - examples

| Regex | Matches | Does not match |
|---|---|---|
| ab*c | **ac**, z**abc**, **abbc**, **abbbc**, **abbbbbc** | ab, xyz |
| ab\{2\}c | **abbc**, a**abbc**c | abc, ab, ac |
| ab\{2,3\}c | **abbc**, **abbbc**, a**abbbc**c | abc, abbbbc |

How to test a regex?

⇒ use `grep pattern <(echo test_string)`

⇒ quote pattern when special chars are used!

⇒ use `-x` to match against the entire string (i.e. no substring search)

# 11.03 Bracket expressions

⇒ use `.` to match an arbitrary character

⇒ `[...]` can be used to specify a character class, i.e. match one of the characters within the square brackets

⇒ if the first char within `[...]` is `^`, this inverts the character class. I.e. match any character that is not contained within `[...]`

⇒ ranges `[a-z]` available like for UNIX wildcards

⇒ character classes can be combined with quantifiers!

# 11.03 character classes

⇒ can be only used within square brackets, e.g. [[:upper:]]

⇒ shortcut syntax with backslash might be used like a regular character

| class | shortcut | [...] | meaning |
|---|---|---|---|
| [:alnum:] | | [A-Za-z0-9] | Alphanumeric characters |
| | \w | [A-Za-z0-9_] | word, i.e. alphanumeric characters + "_" |
| | \W | [^A-Za-z0-9_] | non-word characters |
| [:alpha:] | \a | [A-Za-z] | alphabetic characters |
| [:digit:] | \d | [0-9] | digits |
| | \D | [^0-9] | non-digits |
| [:space:] | \s | [ \t\r\n\v\f] | whitespace characters |
| | \S | [^ \t\r\n\v\f] | non-whitespace characters |

to use shortcuts under Linux use -P to put in PCRE mode

# 11.03 More predefined character classes

⇒ there are many more predefined character classes, e.g.

| | | |
|---|---|---|
| [:blank:] | [ \t] | space and tab |
| [:graph:] | [\x21-\x7E] | printable characters + space |
| [:punct:] | escaped versions of<br>!"#$%&'()*+,./:;<=>?@\^_`{\|}~- | punctuation characters |
| [:lower:] | [a-z] | lowercase letters |
| [:upper:] | [A-Z] | uppercase letters |
| [:xdigit:] | [A-Fa-f0-9] | hexadecimal digit |

⇒ depending on shell + operating system, sometimes even more classes available!

# 11.03 Bracket expressions - examples

| Regex | Matches | Does not match |
|---|---|---|
| [Gg]r[ae]y | **Gray**, **grey**, **gray**, **Grey** | great, grray |
| a.c | **abc**, a**axc**c | cba, ab, ac |
| [^xyz]* | **abbc**, **abbbc**, **aabbbcc** | xyz, abbxbbc |

⇒ a more complex regular expression using character classes is e.g.

```
ISBN
[[:digit:]]\{3\}-[[:digit:]]-[[:digit:]]\{2\}-[[:digit:]]\{6\}-[[:digit:]]
```

to match an ISBN numbers of the form

```
ISBN 978-2-98-123456-0
```

# 11.03 \? and \+ quantifier

⇒ `a\?` matches “a” zero or one time. I.e. makes a optional, short for `a\{0,1\}`

⇒ a\+ matches “a” one or more times. I.e. match a at least once, short for `aa*` (or `a*a`)

# 11.03 Choice/or operator |

⇒ `pattern1\|pattern2` to match either pattern before \| or after
Example: `abc\|def` matches `abc, def, adef` but not `bcd`.

⇒ or operator, also known as choice.

Example: course numbers

```
CS[[:digit:]]\{3\}\|CSCI[[:digit:]]\{4\}
```

# 11.03 Line anchors - ^ $

⇒ ^ matches starting position within the string (i.e. after CRLF)

⇒ $ matches end position of string (i.e. before CRLF)

→ **Note:** vim uses ^ and $ as well to jump to first/last non whitespace character of a line!

Example:

CRLF stands for carriage return (\r) line feed (\n).
Basically means a newline token.

`^hello` matches `hello world` **but not** `Tux says hello!`

⇒ `grep -x` basically adds ^ as prefix, `$` as suffix

# 11.03 Subexpressions via \(...\)

⇒ BRE supports marked subexpression defined via \(...\)

→ Also called *block* or *capturing group*

⇒ \1, …, \9 can be used to refer to the first, …, ninth capturing group. (Sometimes support for more capturing groups). Numbering from outside, left to right.

# 11.03 subexpressions example

```
^o\(aa*xbb*\)*x\(aa*xbb*\)
```

```
Matches:

oxaxb

oaaaaaaxbxaaaxbbbb

oxaaaaaxbb

oaxbxaaxbbxaxbcdefghij        (via grep)
```

```
Does not match:

axb          (o,x missing)

aoxaxb       (does not start with o)
```

# 11.03 subexpression another example

US phone numbers with optional +1-:

```
\(+1-\)\+[[:digit:]]\{3\}-[[:digit:]]\{3\}-[[:digit:]]\{4\}
```

referencing capturing groups:

`^\(aa*\)-\1-\1$`      matches `a-a-a`, `aa-aa-aa`, `aaa-aaa-aaa`

but not `a-aa-aaa` or `aaa-a-a`.

⇒ i.e. `\1` references **the substring** captured by the first group to appear again.

⇒ If we removed `^` and `$`, then `aa-a-a`,  `aa-a-aa` would be matched.

# Extended regular expressions

# 10.04 extended regular expressions

⇒ weird to have to escape `(`, `{` to get special meaning

→ in ERE mode, no need to write \ before `(, {, +, ?, |`

→ E.g. use `(...)` to specify subexpression and `{m,n}` to specify min/max repeats. (Escape via `\(,\),\{,\}`)

⇒ to use grep with ERE either run `grep -E` or `egrep`

⇒ egrep has no support for referencing subexpressions via `\1,...\9`

# 10.04 ERE vs. BRE

| ERE | BRE |
|---|---|
| ^o\(aa*xbb*\)*x\(aa*xbb*\) | ^o\(aa*xbb*\)*x\(aa*xbb*\) |
| a? | a\? |
| \(a\+b*\)\{1,3\} | (a+b*){1,3} |

⇒ your choice whether to use grep or egrep.

# 10.04 A note on linux

⇒ depending whether you have BSD or GNU/Linux there are some extensions available.

⇒ to use shortcuts for character classes, i.e. `\d` instead of `[[:digit:]]`, under GNU/Linux use `grep -P pattern file`. Special characters like (, {, … are treated like in ERE mode.

→ defacto puts grep into PCRE mode
(perl compatible regular expressions)

# Using regular expressions in VIM

# 11.05 Regex in VIM

⇒ you can use regular expressions in VIM to quickly search for lines

⇒ type `/` and enter a regular expression (in BRE mode), then press Enter and use n to iterate over the results
Example:

```
curl https://cs.brown.edu/courses/cs0060/lectures.html | vim -
```

⇒ also possible to use it for replacement

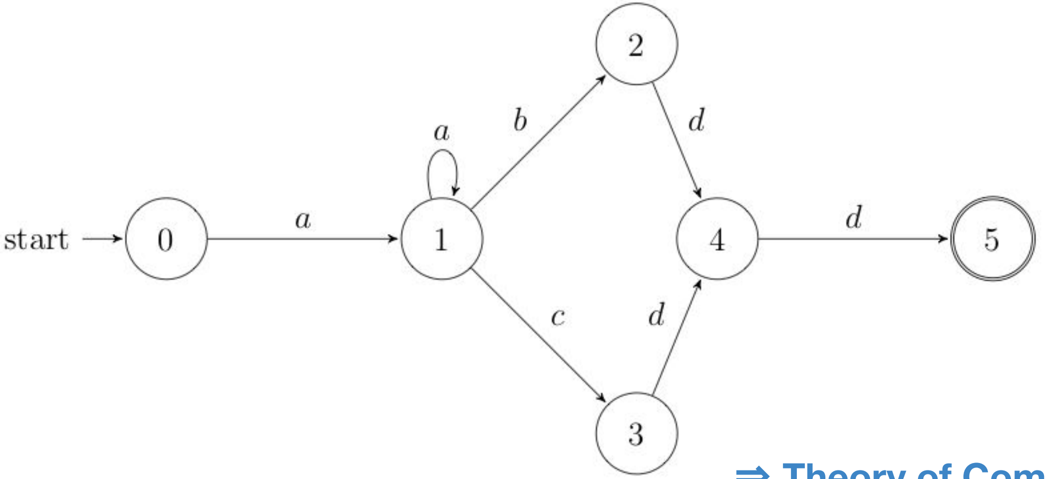
| | |
|---|---|
| `:s/pattern/string/g` | replace pattern with string in the current line |
| `:%s/pattern/string/g` | replace pattern with string in all lines |
| `:s/pattern/string/gc` | same as `:s/.../../g`, but ask for confirmation |

# How to construct regular expressions effectively?

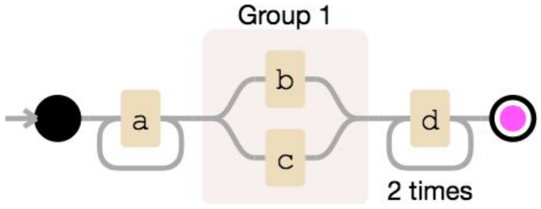
# 11.06 Regular expressions and state machines

regular expressions are closely related to finite state automata. For each regular expression a finite state automaton can be created.

**Example:** `a+(b|c)d{2}`



⇒ Tool to visualize:
https://www.debuggex.com/



**⇒ Theory of Computation CS101 covers this in depth!**

# 11.06 Developing regular expressions

⇒ Websites which host regular expressions, e.g. regexlib.com
**Word of advice:** Do not blindly copy & use regular expressions unless you clearly understand them!

⇒ **Tip:** use a website to develop your regular expression.
Start easy and add features based on test cases.
→ regex101.com
→ regexr.com

⇒ if things fail, try to debug regular expression using debuggex, regex101.com ,... (think of finite state automaton)

# 11.07 Regex vs. wildcards

⇒ Though similar, they're **two different languages**

| Symbol | Regex | UNIX (path) wildcards |
|---|---|---|
| * | quantifier, match preceding character 0 to n times | match any character 0 to n times |
| ? | quantifier, match preceding character optionally | match exactly one arbitrary character |
| . | arbitrary character | dot character . |

⇒ wildcards are semantically a subset of regular expressions.

Regular expressions are more powerful than (simple) wildcards!

Ready, **sed**, go...
...it's getting **awk**ward now!

# 11.08 sed & awk

`sed = streaming editor`

`awk` = named after Alfred Aho, Peter Weinberger & Brian Kernighan.

⇒ both provide a domain specific language (DSL) for processing text data.

⇒ `awk` is a very powerful tool with a fully fledged programming language. There even exist dedicated books to explain how the awk language works.

# 11.08 sed

⇒ non-interactive stream-oriented text processor, typically used as filter in a pipeline

⇒ sed (like awk) reads input stream (stdin, file) line by line, applies specified operations and outputs modified line.

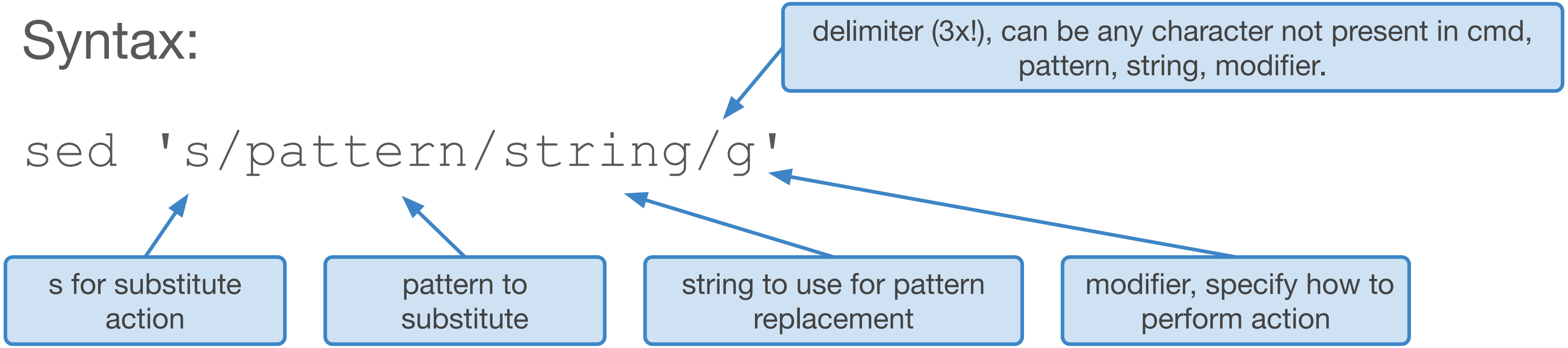
⇒ Basic usage: `sed cmd file` or `sed cmd`

⇒ sed provides ~25 different commands, separate multiple commands using ;

# 11.08 Substitution via sed

⇒ sed allows to search for a pattern and substitute it with a string. When it finds the pattern, it performs the action and repeats.

Syntax:



```
sed 's/pattern/string/g'
```

delimiter (3x!), can be any character not present in cmd, pattern, string, modifier.

s for substitute action

pattern to substitute

string to use for pattern replacement

modifier, specify how to perform action

⇒ use `g` as modifier to replace all occurrences, `1` to replace the first, `2` to replace the second, `3` the third, …

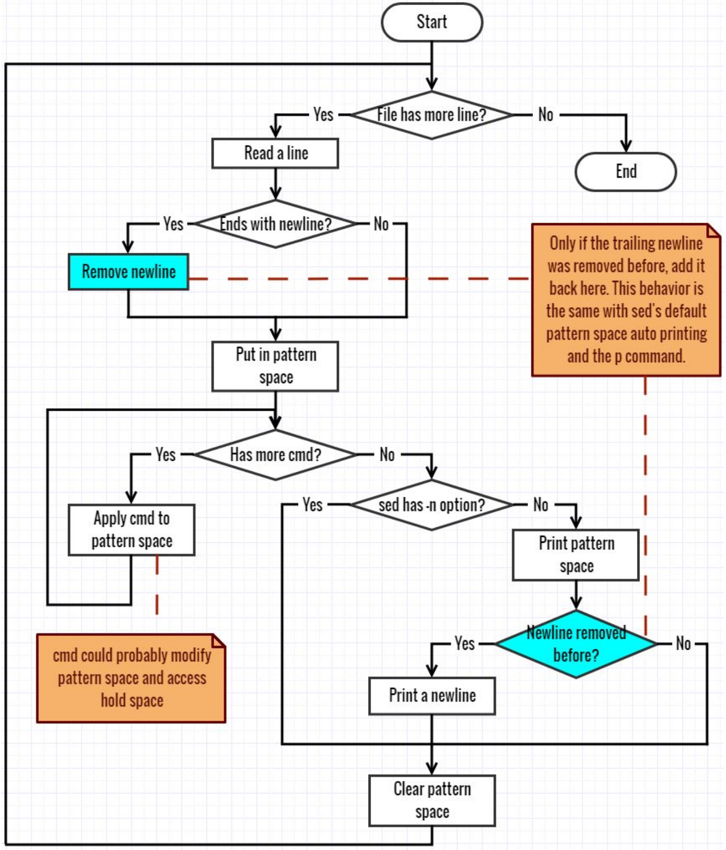
⇒ if modifiers is left away, i.e. sed 's/pattern/string' is used only the first occurence is replaced.

# 11.08 sed cycle

⇒ sed goes line by line (removing trailing newline), then puts .

⇒ underlying it is actually a more complicated structure using two buffers called pattern and hold space which can be accessed & modified in multiple ways:

→ if you're interested `man sed`.



Start
File has more line?
Yes
No
Read a line
End
Ends with newline?
Yes
No
Remove newline
Only if the trailing newline was removed before, add it back here. This behavior is the same with sed's default pattern space auto printing and the p command.
Put in pattern space
Has more cmd?
Yes
No
sed has -n option?
Yes
No
Apply cmd to pattern space
Print pattern space
Newline removed before?
Yes
No
cmd could probably modify pattern space and access hold space
Print a newline
Clear pattern space

# 11.08 substitution - examples

```
sed 's/unix/linux/1' <(echo "unix is a great os. unix is open source."
linux is a great os. unix is open source
```

```
sed 's+unix+linux+g' <(echo "unix is a great os. unix is open source.")
linux is a great os. linux is open source.
```

use regex in BRE

```
sed 's/.nix/linux/g' <(echo "unix is a great os. unix is open source.")
linux is a great os. linux is open source.
```

with `-E`, ERE enabled. Note the whitespace in the pattern!

```
sed -E 's/[^ ]+ix /linux /g' <(echo "unix is a great os. unix is open
source.")
linux is a great os. linux is open source.
```

# 11.08 more practical sed examples

1. convert tabs to 4 spaces using `sed "s/$(printf '\t')/    /"`
2. Prefix lines with string `sed 's/^/prefix/'`
3. suffix lines with string `sed 's/$/suffix/'`
4. Parenthesize first character of each (capitalized) word using

   `sed 's/\(\b[A-Z]\)/\(\1\)/g'`

   → `\b` is anchor for word start, only works under GNU/Linux b.c. GNU extension

   → under Mac OS X/BSD the same can be achieved using

   `sed -E 's/([[:<:]][A-Z])([a-z]*)[[:>:]]/\(\1\)\2/g'`

trick to produce a tab character

special classes to match word start/word end

# 11.08 deletion using sed

⇒ can use sed `s/pattern//` to delete a pattern.
E.g. `sed/^The.*//` deletes the content of lines starting with The

⇒ how to delete full lines?

→ `sed /pattern/d` to delete lines matching pattern

Examples:

1. remove empty lines: `sed /^$/d`
2. remove lines starting with the via `sed /^the/d`

# 11.08 using sed address spaces

⇒ we can restrict commands to be applied to certain lines only!

⇒ prefix the command with the address space

Examples:

1. delete first line `sed '1d'`
2. delete last line `sed '$d'`
3. delete lines 3-5 sed `'3,5d'`
4. delete first 3 lines `sed '1,3d'` (counts from 1)
5. delete lines from lines 3 till end `sed '3,$d'`

# 11.08 sed - print

⇒ instead of using `d` to delete lines, one can also use `p` to print explicitly lines together with the -n option on sed which suppresses echoing each line to stdout (i.e. only output of actions is written to stdout!)

Examples:

1. print first 2 lines via `sed -n 1,2p`
2. print all lines which start with # via `sed -n /^#/p`

without -n, sed would duplicate first two lines!

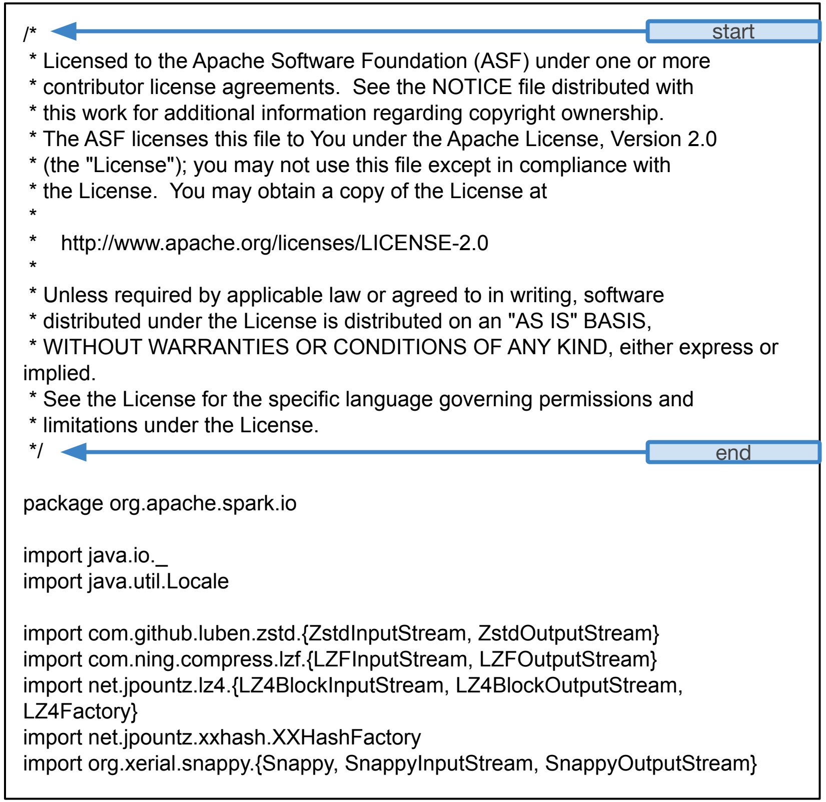
# 11.08 advanced sed

⇒ one can also mix line numbers and patterns to address lines!

⇒ Example: We want to remove license

```
sed '1,/\*\/$/d' file.scala
```

delete all lines starting from first till line where match for */ is found at end of line



```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.spark.io

import java.io._
import java.util.Locale

import com.github.luben.zstd.{ZstdInputStream, ZstdOutputStream}
import com.ning.compress.lzf.{LZFInputStream, LZFOutputStream}
import net.jpountz.lz4.{LZ4BlockInputStream, LZ4BlockOutputStream,
LZ4Factory}
import net.jpountz.xxhash.XXHashFactory
import org.xerial.snappy.{Snappy, SnappyInputStream, SnappyOutputStream}
```

# 11.08 One more practical example

⇒ strip file of comments and empty lines via sed!

```
sed '1p;/^#/d;/^$/d' script.sh
```

`2,$ /^#/d` does not work, can either use addresses OR pattern to specify where to use command

⇒ How does it work?

1. print first line (want to keep shebang line!)
2. remove all lines starting with #
3. delete all empty lines

# 11.08 Adding / inserting lines

⇒ with a line can be appended after addr space/pattern via

e.g. `sed '1a\subheader` file.txt

⇒ insert line before addr space/pattern via i

e.g. adding bash shebang line to a script via

```
sed '1i\#!/bin.bash/' script.sh
```

→ one can leave away the \, i.e. `sed '1i test'`

# 11.08 More about sed

→ Useful links to learn more sed commands:

1. http://www.theunixschool.com/2014/08/sed-examples-remove-delete-chars-from-line-file.html
2. https://www.geeksforgeeks.org/sed-command-in-linux-unix-with-examples/
3. https://www.gnu.org/software/sed/manual/sed.html
4. http://www.grymoire.com/Unix/Sed.html

Another **awk**ward tool…

# 11.09 awk

⇒ pattern-matching programming language with variables, arithmetic operations, string operations, loops, conditionals, …

⇒ works similar to sed line-by-line and applies a small program to each line

⇒ in sed basically commands, patterns, modifiers vs.
awk's philosophy: supply a tiny program!

# 11.09 Basic awk

General syntax:

can leave condition or action parts out if necessary.

```
awk 'condition {action; action; }' file
```

⇒ condition can be e.g. a pattern, i.e. `awk '/pattern/ { ... }'\`

⇒ simplest action: `print` → prints line for which condition is true

Example:

```
awk '/^tux/ {print}' file.txt  # print all lines
                               # which start with tux.
```

`awk /pattern/` does basically the same as grep. If no action is specified, print is used per default.

# 11.09 More on actions & conditions

⇒ `awk` automatically splits each line into fields, which can referenced using `$1, $2, …`

⇒ `$0` holds the complete line

⇒ use `-F delimiter` to specify delimiter for awk

⇒ special variables exist which hold e.g.

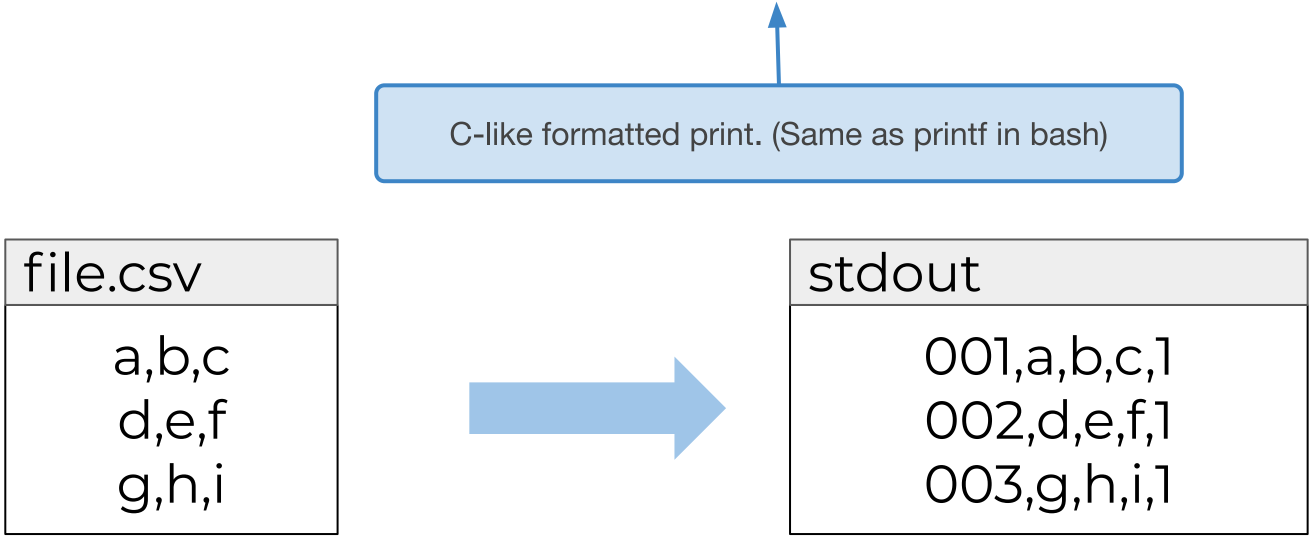
`NF` (number of fields)

`NR` (number of current record, i.e. line number)

`FS` (field separator)

# 11.09 Awk example, adding line numbers

```
awk '{printf("%03d,%s,%d\n",NR,$0,NF);}' file.csv
```



C-like formatted print. (Same as printf in bash)

file.csv

```
a,b,c
d,e,f
g,h,i
```

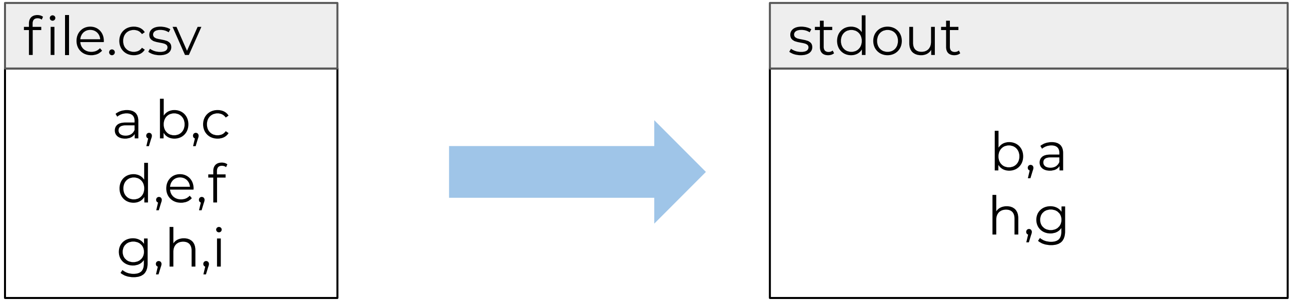
stdout

```
001,a,b,c,1
002,d,e,f,1
003,g,h,i,1
```

# 11.09 Awk example, swapping columns

```
awk -F, '/^[ag]/ {print $2 FS $1;}' test.csv
```

you can also write `print($2 FS $1)`. Same as `printf("%s%s%s\n",$2,FS,$1)`



file.csv

```
a,b,c
d,e,f
g,h,i
```

stdout

```
b,a
h,g
```

How does it work?

1. `-F,` splits on comma. I.e. for the first line `$0` holds `a,b,c` `$1` holds `a`
2. `/^[ag]/` is only true for lines which start with either a or g
3. `print $2 FS $1` prints NEWLINE delimited value of second field, then the field separator, then the first field. Note: `print $2,$1` would print fields separated by whitespace

# 11.09 More on awk

⇒ awk **IS** complex. It has a very powerful language.

Resources:

1. http://www.grymoire.com/Unix/Awk.html
2. https://www.gnu.org/software/gawk/manual/gawk.pdf (570 pages !!!)
3. Effective awk Programming (3rd Edition) by Arnold Robbins

# 11.10 Next lecture

There is **no lecture** next Tuesday, 8th October

⇒ Next lecture is on Thursday 10th October

⇒ Lab on 10th October, 8pm-10pm.

⇒ Intro to HTML, HTTP requests & CSS

# End of lecture.

Next class: Thu, 4pm-5:20pm @ CIT 477

# Appendix

Fetching lecture slides via grep/sed/curl:

```
curl -s https://cs.brown.edu/courses/csci0060/lectures.html | grep -Eo
'href=".*\.pdf' | sed 's/href="\.//' | sed
's|^|https://cs.brown.edu/courses/csci0060|' | xargs -n1 curl -O
```