Advanced text processing: regex sed awk

CIS 2230 Linux System Administration

Lecture 9

Steve Ruegsegger

Review

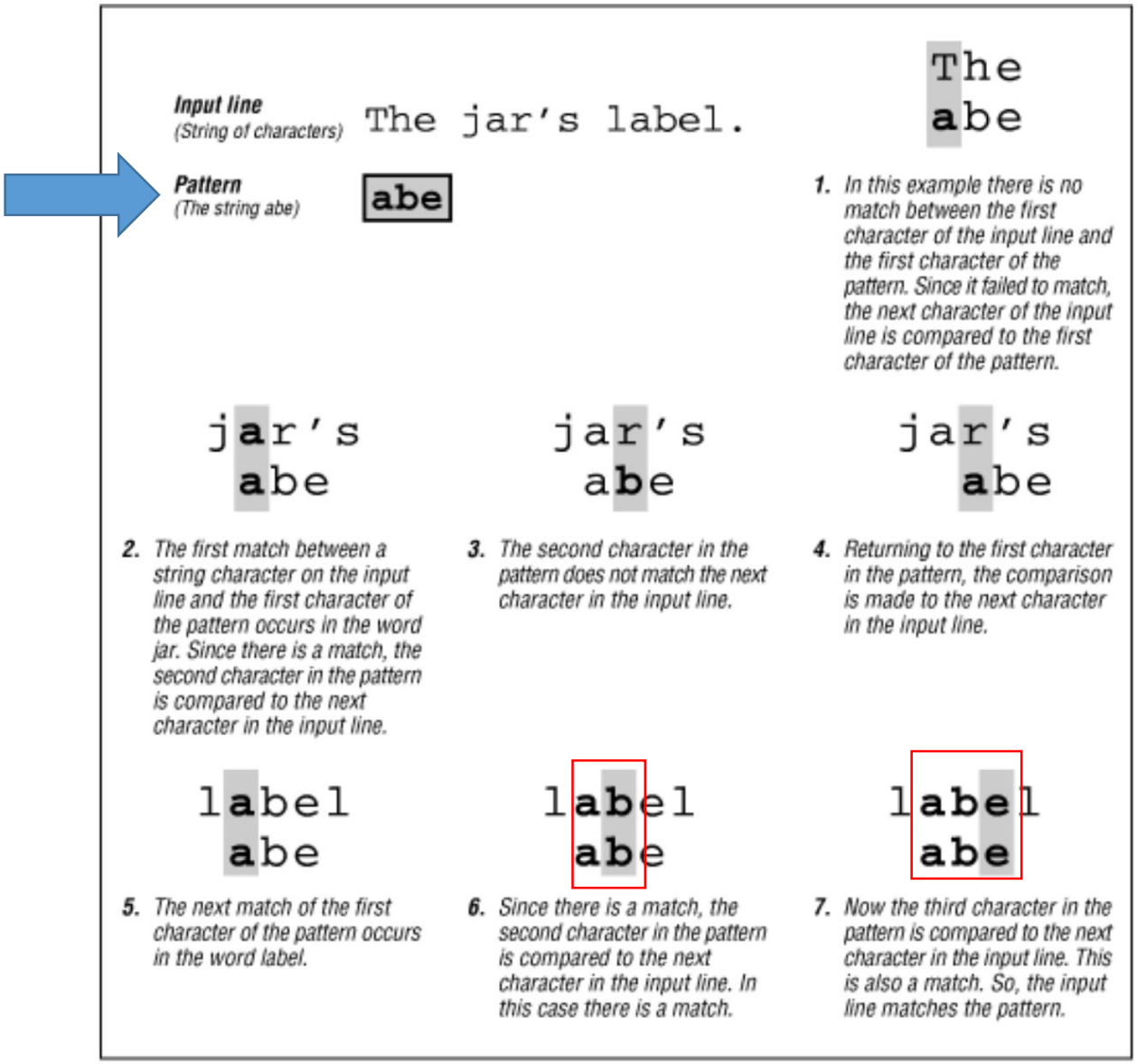
- What are the 3 standard streams?
- What are their numbers?
- Explain what this command does?
  `$ echo 'whoami' 'hostname' 'date' >> run.log`
- What is `/dev/null`?
- Explain `head` & `tail`?
- Explain `more` & `less`?
- What does `$ tail -f <file>` do?
- What must precede `uniq`? Why?
- What are the 4 types of pattern matching for `grep`?
- What do `zcat` and `zmore` do?

What is a Regular Expression (regex)?

- A regular expression is a set of characters that specify a pattern.
- Regular expressions are used when you want to search lines of text containing a particular pattern.
- regex looks character-by-character for the pattern anywhere in the line of text
- grep comes from g/re/p
  - `g/re/p` is the "ed" command: "global regular expression print"
  - It runs regex pattern search for each line in a text file or stream
- Examples:

```
$ grep PATT files...
$ cat file | grep PATT
```

How regex pattern matching works



http://docstore.mik.ua/orelly/unix/sedawk/ch03_01.htm

Simple regex example

same

```
$ grep OH foobar.txt
$ cat foobar.txt | grep OH
```

foobar.txt

```
Centerville, OH 45459
I'm driving to Ohio
Oh, that puppy is cute
The OHIO STATE BUCKEYES lost to Michigan again.
John is a friend of mine
```

Regex /OH/ match?

Now, try the -i option:

```
$ grep -i OH foobar.txt
```

Run grep “interactively” by simply using stdin

```
$ grep “ing”
```

Rules for matching

- There are three (3) important "parts" to a regular expression
  1. **Anchors** – specify the *position* of the pattern in relation to a line of text.
  2. **Character Sets** – match one or more characters in a single position. (wildcards)
  3. **Modifiers** – specify how many times the previous character set is repeated.

Rules for matching

- Let's start with a simple *example* regex:

```
^# *[Ss]teve
```

  - ^ is an anchor that indicates the beginning of the line.
  - # is a simple character (shell comment)
  - " *" is a modifier, specifying that the previous character (space) can appear any number of times, including zero.
  - [Ss] is a character set of next characters

- These lines all return T Why?

```
#Steve wrote this
#   steve ruegsegger
# steve@yahoo.com
```

These do NOT.

```
Steve was here
Steve #1
# written by steve
# - steve rueg
```

The Anchor Characters: ^ and $

- ^ at the very beginning of the regex means the pattern must match at the *beginning* of the line
- $ at the end of the regex means the pattern must match at the *end* of the line
- Ex:

```
$ ps -ef | grep '^bob'        # will not match jimbob
$ ps -ef | grep 'firefox$'    # only ending in firefox
```

Character sets

1. Specific string

```
$ who | grep "steve"
```

2. Match "any character" with "." (dot), i.e. single character wildcard = .

```
$ ps | grep "sas.."
```

- TRUE for these strings: sas80, sas81, sas91, sas92

3. Specifying a Set or Range of Characters with [ ]

Common ranges:

```
^[0123456789]$
^[0-9]$
[A-Za-z0-9_]
```

Note: these are 'character' ranges, not numeric

- Complex example:

Line starts with “VTC” then has at least one space
followed by a C and 2 more cap. letters then 1 number

```
^VTC *C[A-Z][A-Z][0-9]
```

Numeric ranges in regex (not)

- Regex is character pattern matching
- Regex is character pattern matching
- What if I want all the 400's (400 – 499)

```
/4[0-9][0-9]/
```

- There is no [400-499] *numeric* range

Exceptions in a character set

- If the “carat” or “hat” (^) is the first character in a [ ] range, then it means "everything except this set"!
- Example
- Match upper-class courses (3000 & 4000 level)

```
$ grep 'CIS[^12][0-9][0-9] '
```

Repeating character sets with *

- The 3rd part of a regular expression is the modifier.
- It is used to specify how may times you expect to see the previous character set.
- The special character "*" matches zero or more copies.
- Example:
  “who logged into “pts/0” on M-W this week (9/2 – 9/4)”

```
$ last | grep "pts/0 .*Sep [234]"
```

```
steve-r@srueg:/var/log$ last | grep "pts/0 .*Sep [234]"
steve-r  pts/0        :0.0             Wed Sep  4 19:47 - 23:26  (03:38)
steve-r  pts/0        :0.0             Wed Sep  4 15:30 - crash  (01:41)
steve-r  pts/0        :0.0             Tue Sep  3 14:30 - 15:27 (1+00:57)
```

Matching a specific number of sets with \{ and \}

- There is a special pattern you can use to specify the minimum and maximum number of *repeats*.
- This is done by putting those two numbers between "\{" and "\}".
  - For grep, a { } in the regex means you are searching for those characters.
  - But placing a backslash *before* {}'s or ()'s turns on a *special meaning*.
  - So, the escaped curley brackets mean that the character *before* is to be *repeated*
- Examples:
  - Find words 4 to 8 chars in length: `[a-z]\{4,8\}`
  - Find numbers in pairs: `[0-9]\{2\}`
  - Who's been online for > 9 days?

```
$ w | grep '[1-9]\{2\}days'
steve    tty1     -        28Aug11 19days  0.28s  0.18s -bash
steve    tty7     :0       26Aug11 21days 18:31m 38.51s gnome-session
```

OR

- Use pipe | to mean a logical “or”
  - The pipe must be *escaped* to mean “or”
- Use parens () to group the OR options
  - The parens must be *escaped* too
- Ex:

```
grep '^\(From\|Subject\): ' /spool/mail/$USER
ps -ef | grep '\(root\|steve\)'
ls -l | grep '\(aug\|sep\).*foo.*txt$'
```

Sets of Versions of Regular Expression

- Regexp also has more powerful rules
- There are several 'sets' or 'types' or 'versions' of the regexp definition
- Why do you think?
- All the previous regexp discussion has been for the Basic set.

"command"

Utility	Regular Expression Type
vi	Basic
sed	Basic
grep	Basic
csplit	Basic
dbx	Basic
dbxtool	Basic
more	Basic
ed	Basic
expr	Basic
lex	Basic
pg	Basic
nl	Basic
rdist	Basic
awk	Extended
nawk	Extended
egrep	Extended
EMACS	EMACS Regular Expressions
PERL	PERL Regular Expressions

Extended Regular Expressions

- There is an *extended* regular expression definition
- It tried to make regex 'better' – but it couldn't just replace the basic, established regex.
- So, this definition is called teh 'extended' definition
- Used by `egrep` and `awk`.
  - Note: `egrep` same as `$ grep -E`

What's new in extended regex

1. egrep "switched" the meaning of *escaping* { }'s and ( )' .
   Therefore, { }'s and ( )'s have the 'special meaning'.
   If you want to actually search for a { } character, now you escape it.
2. OR also doesn't have to be escaped
3. Special escaped sets
   - \w = [a-zA-Z0-9_] "word characters: numbers, letters and underscore"
   - \W = [^a-zA-Z0-9_] "the opposite of \w"
4. The special character "+" matches one or more copies.

```
$ ps -ef | egrep 'tty[0-9]+'
```

5. The special character "?" matches zero or one.

```
$ cat apache.log | egrep
'https?:\/\/(www\.)?cnn\.com'
$ w | egrep 'days?'
```

Regex Summary

Modifiers

Modifiers are used to perform case-insensitive and global searches:

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

Syntax: /regex/modifiers

Ex:

```
/# +steve/i      # will match Steve and steve and STEVE
```

http://www.w3schools.com/jsref/jsref_obj_regexp.asp

Regex Summary

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
.	Find a single character, except newline or line terminator
\w	Find a word character
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character
\s	Find a whitespace character
\S	Find a non-whitespace character

Regex Summary

Brackets

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character NOT between the brackets
[0-9]	Find any digit between the brackets
[^0-9]	Find any digit NOT between the brackets
(x|y)	Find any of the alternatives specified

Regex Summary

Quantifiers

Quantifier	Description
n+	Matches any string that contains at least one *n*
n*	Matches any string that contains zero or more occurrences of *n*
n?	Matches any string that contains zero or one occurrences of *n*
n{X}	Matches any string that contains a sequence of *X* *n*'s
n{X,Y}	Matches any string that contains a sequence of X to Y *n*'s
n{X,}	Matches any string that contains a sequence of at least X *n*'s
n$	Matches any string with *n* at the end of it
^n	Matches any string with *n* at the beginning of it

Basic vs Extended Summary

task	Basic	Extended
Look for 2 numbers	`[0-9]\{2\}`	`[0-9]{2}`
Look for the string “{foobar}”	`{foobar}`	`\{foobar\}`
Look for zipcodes 05465 or 05111	`05\(465|111\)`	`05(465|111)`
Look for “CIS” and 1 or more numbers	`CIS[0-9][0-9]*`	`CIS[0-9]+`
Look for “http:” or “https:”	`https\{0,1\}:`	`https?:`

Regex strategies

- *e.g.* look for email addresses

1. Usually, start pattern with an “anchor”
   - Example below: @, \.com
2. Use wildcards where the pattern needs them
3. Use “head” and “more” to (spot) check work

```
$ grep -E 'Univ\w*.?( of)? Mich'
```

Text anchors

Text file

```
I'm vacationing in Michigan
She's at University of Michigan
He graduated from Univ. Mich.
After University of Dayton, she moved to Michigan
```

Regex match?

Numeric ranges in regex

- Regex is "pattern matching"
- Regex does not know what a "number" is
- Regex does not understand place value (1's, 10's, 100's)
- How to match 400-559?

```
/(4[0-9][0-9]|5[0-5][0-9])/
```

- What about 400-560?

```
/4[0-9][0-9]|5[0-5][0-9]|560/
```

Examples

- Do all these make sense?
  - Typical City, State Zip

```
^(\w )+, [A-Z]{2} \d{5}(-\d{4})?
```

  - match a date between 1900-01-01 and 2099-12-31:

```
^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$
```

  - valid email:

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
```

  - Valid MC number:
    All MasterCard numbers start with the numbers 51 through 55. All have 16 digits

```
^5[1-5][0-9]{14}$
```

  - Valid Visa cc number: All Visa card numbers start with a 4. New cards have 16 digits.
    Old cards have 13.

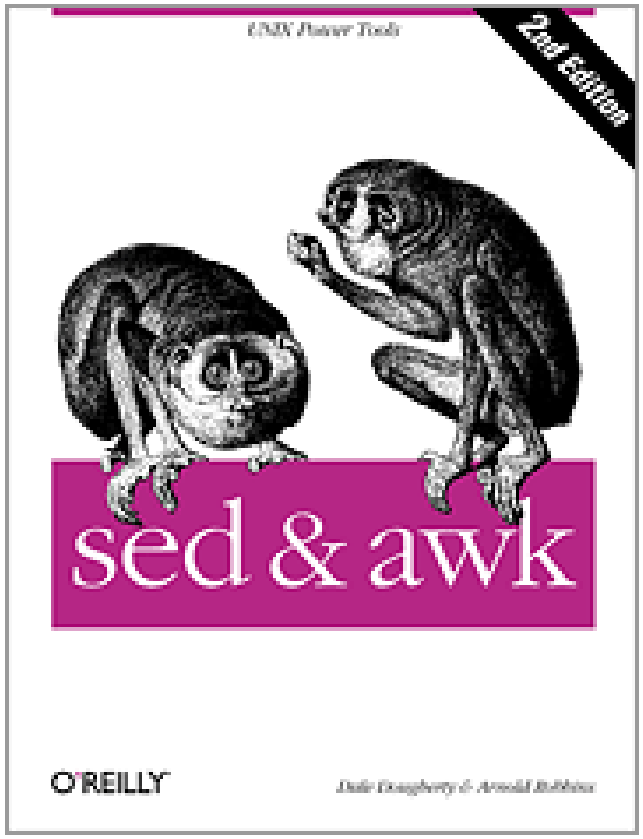
```
^4[0-9]{12}([0-9]{3})?$
```

- Write a regex to match VTC CIS course numbers. For example, “CIS2230” or “cis 2235”.

sed

The Awful Truth about sed

- Sed is the ultimate stream editor.
- Most people never learn its real power.
- The language is very simple, but the documentation is *terrible*.
- `man` won't really help
- I have a sed and awk book which I often have to reference when using sed or awk



The one and only – essential command

- format: `sed <commands> [FILE]`
- sed has several commands, but most people only learn one -- the substitute command: s
- The substitute command changes all occurrences of the regular expression into a new value.
- A simple example is changing "day" in the "old" file to "night" in the "new" file:

```
sed 's/Fall 2015/Spring 2016/' old > new
```

Ex

```
$ echo "Good day." | sed 's/day/night/'
$ sed 's/ MA/, Massachusetts/' list.txt
```

The slash as a delimiter

- The character *immediately* after the `s` is the delimiter.
  - It is conventionally a slash : /
  - But, alas, it can be *anything* you want, however.
- If you want to change a pathname that contains a slash - say /usr/local/bin to /common/bin :

```
sed 's/\/usr\/local\/bin/\/common\/bin/' < old >new
```

- *Gulp*. Some call this a 'Picket Fence' and it's ugly. It is easier to read if you use an underline instead of a slash as a delimiter:

```
sed 's_/usr/local/bin_/common/bin_' < old >new
```

- *Some* people use colons:

```
sed 's:/usr/local/bin:/common/bin:' < old >new
```

- *Others* use the "|" character.

```
sed 's|/usr/local/bin|/common/bin|' < old >new
```

Replace Patterns

- The 'search term' can be a regex pattern!
- Eg

```
sed 's/[aA]uthor/Steve/' book.txt
sed 's/<H[23]>/<H4>/' index.html
sed '/[uU]niv.*[mM]ich[^ ]*/U of M (Go Blue!) /' foo.txt
```

Remember the match

- Put parens around patterns to match and 'remember'
- During substitution, "`\n`" is the remembered pattern
- sed has up to \9 remembered patterns.
- Examples:
  - If you wanted to keep the first word of a line, and delete the rest of the line:

```
sed 's/\([a-z]*\).*/\1/'
```

- What will these do?

```
$ echo cis2230 | sed 's/[a-z]*\([0-9]*\)/CIS\1/'
$ echo 'Steve Ruegsegger' | \
  sed 's/\([a-zA-Z]*\) \([a-zA-Z]*\)/\2, \1/'
```

-r option for 'regex-extended'

- Now, the parens are not escaped! (Whew)
- What will these do?

```
$ echo cis2230 | sed -r 's/[a-z]*([0-9]*)/CIS\1/'
$ sed -r 's/file([0-9]*)\.doc/myfile\1.bak/'
$ sed -r 's/var([0-9])/variable0\1/'  *.c
```

sed -f scriptname

- If you have a large number of sed commands, you can put them into a file and use

```
sed -f sedscript <old >new
```

where sedscript could look like this (one sed command per line):

```
# comment
s/Disk A/Disk B/
s/2230/2235/g
s/EECS\([0-9\)/ECE\1/
```

AWK

awk

- There are three variations of AWK:
  1. AWK - the original from AT&T
  2. NAWK - A newer, improved version from AT&T
  3. GAWK - The Free Software foundation's version
  - In Ubuntu, check out `$ man awk`
- Purposes
  - It is an excellent filter and report writer.
  - AWK is an excellent tool for processing rows and columns for text files
  - Within each column, you can do a numeric search, a string match, or a regex search. (cool!)
  - As you know, many UNIX utilities generates rows and columns of information

Basic Structure

- The essential organization of an AWK program follows the form:

```
$ awk '(pattern) { action }' [files...]
```

- The quotes define the first argument to awk
- The second arg is a list of files
  - Optional, of course
  - We often STREAM into an awk:

```
$ cat <file> | grep <something> | sort <this> | \
  head <that> | awk '(pattern) { action }'
```

awk pattern

- Recall

```
$ awk '(pattern) { action }'
```

- The `pattern` specifies on which lines the action is performed
  - *i.e.* a grep pre-filter 
- Like most UNIX utilities, AWK is line oriented.
  - That is, the pattern specifies a test that is performed with *each line* read as input.
  - If the condition is true, then the action is taken.
  - The default pattern is something that matches every line.

awk pattern types

- There are 3 main pattern types:
  1. Numeric: `$1 > 3`
  2. String: `$4 eq 'steve'`
  3. Regex: `$6 ~ /^foo.*[6-8]/`
- They can be combined in any combination!

```
$ ps -ef | awk '($2 > 100 && $1 eq 'steve' \
  && $8 ~ /firefox/)'
```

awk action

- The most simple action is to *print*
- Each column is a numbered variable
  - col 1 = `$1`
  - col 2 = `$2`
  - etc.
  - `$0` is all cols in the lines
- Build a string to print out with quotes and column numers ($1, $2, etc).
- Separate into output columns with a "," if you want
- Often we use `awk` to simply print select cols.

```
$ ls -l | awk '{print "fn="$8, "owner=$3"}'
```

FS - The Input Field Separator Variable

- To specify a specific field separator
  - `-F` option from the command line
  - `FS=` in a shell program
- Examples

```
$ awk -F: '{if ($2 == "") print $1 ": no password!"}' </etc/passwd
$ awk -F: '($3 >= 1000) {print "user: " $1}' /etc/passwd
```

awk example

```
$ ls -l | awk '($1!~/^d/ &&  $5 > 2000){print $5,$3,$9}' | sort -n

2576 xorg.conf.new root
10817 getpw* steve
18390 wiki.txt.old steve
18390 wiki.txt steve
671350041 rao0.mp4 steve
```

Special awk Actions

- Two other important patterns are specified by the keywords "BEGIN" and "END."
  - As you might expect, these two words specify actions to be taken before any lines are read, and after the last line is read.
- format for "action":

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

- Example:

```
ls -l | awk \
'BEGIN { print "File\tOwner" }\
 { print $8, "\t", $3} \
END { print " - DONE -" }'
```