CSE 374
Programming Concepts & Tools

Brandon Myers

Winter 2015

Lecture 6 – sed, command-line tools wrapup

(Thanks to Hal Perkins)

Friday: Hacker tool of the week

- I want to see the beginning of a file
  - **head** salaries.csv
  - head -n 2 salaries.csv # first 2 lines
  - egrep -o '^["]C[a-zA-Z]' salaries.csv | head -n 1
- I want to see the end of a file
  - **tail** words
  - tail -n 10 salaries.csv # last 3 lines
  - tail -n +4 salaries.csv # line 4 to end
  - tail -f mynotes.txt
    - "follow" mode: (don't exit immediately) print lines as they are appended to the file

Where we are

- Learned how to use the shell to run, combine, and write programs
- Learned regular-expressions (plus more) and grep for *finding* guided by regexps
- Now: finish grep then learn sed for *find-and-replace* guided by regexps
- Then: Short plug for awk (not tested or taught)

Previous matches – back references

- Up to 9 times in a pattern, you can group with (*p*) and refer to the matched text later!
  - (Need backslashes in sed.)
- You can refer to the text (most recently) matched by the n^{th} group with \n.
- Simple example: double-words ^([a-zA-Z]*)\1$
- You cannot do this with actual regular expressions; the program must keep the previous strings.
  - Especially useful with sed because of substitutions.
- Last time: four of the same vowel?

Review

- grep takes a pattern and a file (or stdin)
- The pattern describes a regexp:
  - Example: a[bc]*.?.?d*e
  - Special characters: . ? ^ $ * ( ) [ ] + { } \ | (Some need escaping in grep but not in egrep)
- grep prints any line that has one or more substrings that match.
  - Or invert with -v
  - Or count with -c
- So the output is basically a subset of the input. What if we want to *change* or *add* some output? Enter sed…

sed

- A *stream editor*; a little language that processes one line at a time
- Simple most-common use (and -e optional here):

  sed -e s/*pattern*/*replacement*/g *file*
- "For each line in file, replace every (longest) substring that matches *pattern* with *replacement* and then print it to stdout." (as with grep, often want to quote 's/…/…/g' to avoid shell substitutions)
- Simple variations:
  - omit file: read from stdin
  - omit g: replace only first match
  - sed -n 's/…/…/p' : print only lines with a match
  - multiple -e s/.../.../...: apply each left-to-right
  - -f file2: read sed script from file; apply each line top-to-bottom

More sed

- The replacement text can use \1 . . . \9 – very common.
- Example usage: To avoid printing the whole line, match the whole line and then have the replacement print only the part you want.
- Newline note: The \n is not in the text matched against and is (re)-added when printed.
  - i.e., lines are read into an “edit buffer” and processed there without the (local system’s) newline.

Even more sed

- “sed lines” can have more:
  - different commands (so far, s for substitution)
    - A couple others: p, d, N
    - Other useful ones use the hold space (next slide)
  - different addresses (before the command)
    - number for exactly that line number
    - first~step (GNU only) (lines are first + n*step)
    - $ last line
    - /regexp/ lines containing a match of regexp

Fancy stuff

- Usually (but not always) when you get to this stuff, your script is unreadable and easier to write in another language.
  - • The “hold” space. One other string that is held across lines. Also the “pattern” space (where the “current line” starts).
    - x – swap hold space and pattern space
    - G – append hold space to pattern space
    - H – append pattern space to hold space
  - a label such as :foo before address or command

[:label] [address] [command-letter][more-stuff-for-command]

  - Branches to labels (b and t)
    - Enough to code up conditionals and loops like in assembly language.
- Your instructor never remembers the details, but knows roughly what is possible.

sed summary

- The simplest way to do simple find-and-replace using regexps.
- Standard on all Linux/Unix systems, even in limited recovery boot modes
- Programs longer than a few lines are possible, but probably the wrong tool.
- But a line-oriented stream editor is a very common need, and learning how to use one can help you use a better one.
- In homework 3, a “one-liner” is plenty.
- For the rest, see the manual.

awk

We will skip awk, another useful line-oriented editor.

Compared to sed:

+ Much saner programming constructs (math, variables, for-loops, if statements, functions)
+ Easier to print “fields” of lines, where fields are separated by a chosen “delimiter”
+ Easier to process multiple lines at a time (change the end-of-line delimiter)
- Less regexp support; one-liners not as short

String-processing summary

- Many modern scripting languages (perl, python, ruby, et al) support grep, sed, and awk features directly in the language, perhaps with better syntax.
  - Better: combine features
  - Worse: one big program that "hopefully has everything" instead of useful small ones
- When *all* you need to do is simple text manipulation, these tools let you "hack something up" quicker than, say, Java.
- But if you need "real" data structures, performance, libraries, etc., you reach their practical limits quickly.
- Control flow is easier in "real" languages than in sed