CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Spring 2008

Lecture 6— sed, command-line-tools wrapup

Where are we

- learned how to use the shell to run, combine, and write programs
- learned regular-expressions (plus more) and `grep` for *finding* guided by regexps.
  - To finish up: reusing previous parts of matched string
- Now: `sed` for *find-and-replace* guided by regexps
- Then: Short plug for `awk` (not tested or taught)
- Then: Introduction to C

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; see the man page)

`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 terrible language that processes one line at a time. Multi-line manipulations possible but painful.

Simple most-common use (and `-e` optional here):
`sed -e s/pattern/replacement/g file`

"For each line of file, replace every (longest) substring that matches pattern with replacement and then print it to standard out."

Simple variations:

- omit file: read from stdin
- omit `g`: replace only first match
- `sed -n` and add `p` where `g` is: print only lines with ≥ 1 match
- multiple `-e s/.../.../...`: apply each left-to-right
- `-f file2`: read script from file; apply each line top-to-bottom

More sed

The replacement text can use \1 ... \9 – very common.

Hint: 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.

Aside: "Line-ending madness" on 3 common operating systems.

Toward full `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
- a label such as `:foo` before address or command

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

The 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`, `G`, `H`
- 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.

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 2, 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, ...)
- 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 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.