CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Winter 2007

Guest Lecture

Today

- Quickly finish up a couple bash programming features you need for homework 1.
- Today: Specifying string patterns for many utilities, particularly `grep` and `sed`.
  - Will use only `grep` (and `egrep`) today.
  - Only finding (vs. finding-and-replacing)

Bash command-line arguments

When you run a bash script, you can pass it arguments:

- Third argument in variable 3 (i.e., get it via $3)
- $0 holds the script name (from the caller's perspective)
- $# is the total number of arguments

See `arguments.sh` for examples.

Bash tests

Can *test* arithmetic facts (e.g., “at least 3 arguments”) and file-system facts (e.g., “is blah a directory”)

Many different operators; look them up.

To get a zero or a one, put test in brackets *with spaces around them*. Typically used with a conditional command.

- `[ $# -gt 0 ]`
- `[ -f foo]`

See `example1.sh` for examples.

See Pocket Guide pp. 168–171 (and, or, ...)

Globbing vs. Regular Expressions vs. ...

"Globbing" refers to filename expansion characters.

"Regular expressions" are a different but overlapping set of rules for specifying patterns to programs like `grep`. (Sometimes called "pattern matching".)

More distinctions:

- Regular expressions a la CSE322
- "Regular expressions" in `grep`
- "Regular expressions" in `egrep` (same as `grep -E`)
- More subtle distinctions per program...

Real Regular Expressions

Some of the crispest, elegant, most useful CS theory out there.

What computer scientists know and ill-educated hackers don't (to their detriment).

A regular expression p may "match" a string s. If $p =$

- a, b, ... matches the single character
- $p_1 p_2$, ... if we can write s as $s_1 s_2$, p_1 matches s_1, p_2 matches s_2.
- $p_1 | p_2$, ... if p_1 matches s or p_2 matches s (in `egrep`, for `grep` use \|)
- $p_1 *$, if there is an $i \geq 0$ such that $\underbrace{p_1 \ldots p_1}_{i}$ matches s.

  (for $i = 0$, matchines the zero-character string).

Lots of examples with `egrep`.

Why this language?

Amazing facts (see 322):

- Exactly the patterns that can be found by a program that can say *before* it sees its input how much space it needs. (Decide if a 1GB string has a substring that matches...)
- You can write a program that takes two regular expressions and decides if one matches every string the other does.
- ... see CSE322

Conveniences

Lots of "conveniences" do not make the language any more powerful:

- p_1+ is just $p_1 p_1*$
- $p_1?$ is just $(|p_1)$
- `[zd-h]` is just `z | d | e | f | g | h`
- `[^A-Z]` and `.` are long but technically just conveniences.
- $p_1\{n\}$ is just $\underbrace{p_1 \ldots p_1}_{n}$
- $p_1\{n,\}$ is just $\underbrace{p_1 \ldots p_1}_{n} p_1*$
- $p_1\{n,m\}$ is just $\underbrace{p_1 \ldots p_1}_{n} \underbrace{p_1? \ldots p_1?}_{m}$

Beginning and end

Really `grep` is matching each line against `.*`p`.*`.

You can say that is not what you want with ^ (beginning) and $ (end) or both (match whole line exactly).

I can't think of a good reason to put these characters in the middle of a pattern, but you can.

Fundamentally, we are still in the realm of "real" regular expressions.

Nasty gotchas

- Special characters for one program not special for another.
- For example, I found `\{` for `grep` but `{` for `egrep`.
- Must quote your patterns so the shell does not muck with them – and use single quotes if they contain $.
- Must escape special characters with \ if you need them literally: `\.` and `.` are very different.
  - But inside `[]` less quoting (so backslash becomes literal)!

Previous matches

- 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} one with $\backslash n$.
- Simple example: double-words `^\([a-zA-Z]*\)\1$`
- You *cannot* do this with regular expressions; the program must keep the previous strings.
  - Especially useful with `sed` because of *substitutions*.