CSE 303: Concepts and Tools for Software Development

Dan Grossman
Spring 2005
Lecture 5— Regular Expressions (and more), grep, find
Where are We

• We are done learning this bizarre pseudo-programming language called the shell.

• Today: Specifying string patterns for many utilities, particularly `grep` and `sed`.

• Friday: Ben teaches `sed` (needed for homework 2).

• Monday: We start learning C.

Note: Homework 2 is hard and due next Friday. Already posted.
“Globbing” refers to filename expansion characters.

“Regular expressions” are a different but overlapping set of rules for specifying patterns to programs like `grep`. (Called “pattern matching” in the Nutshell book.)

More distinctions:

- Regular expressions a la CSE322
- “Regular expressions” in `grep`
- “Regular expressions” in `egrep`
- More subtle distinctions per program...
Real Regular Expressions

Some of the crispet, 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 $p =$

- $a, b, \ldots$ matches the single character
- $p_1p_2, \ldots$ if we can write $s$ as $s_1s_2$, $p_1$ matches $s_1$, $p_2$ matches $s_2$.
- $p_1|p_2, \ldots$ if $p_1$ matches $s$ or $p_2$ matches $s$ (in egrep, not grep or sed)
- $p_1^*$, if there is an $i \geq 0$ such that $p_1 \ldots p_1$ matches $s$.

(for $i = 0$, matches 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
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 $z \mid d \mid e \mid f \mid g \mid h$
- $[^A-Z]$ and . are long but technically just conveniences.
- $p_1\{n\}$ is just $p_1 \ldots p_1$
- $p_1\{n,\}$ is just $p_1 \ldots p_1 p_1^*$
- $p_1\{n, m\}$ is just $p_1 \ldots p_1 p_1 ? \ldots p_1 ?$
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)!

Dan Grossman CSE303 Spring 2005, Lecture 5
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 \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.