

CSE 303: Concepts and Tools for Software Development

Hal Perkins

Spring 2008

Lecture 5— Regular Expressions (and more), grep, other utilities

Where are We

- We are done learning this bizarre pseudo-programming language called the shell (pick up more for hw2).
- Today: Specifying string patterns for many utilities, particularly `grep` and `sed` (also needed for hw2).
 - find vs. find-and-replace
- Next: `sed`
- And then: We start learning C.

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, \dots matches the single character
- p_1p_2, \dots if we can write s as s_1s_2 , p_1 matches s_1 , p_2 matches s_2 .
- $p_1|p_2, \dots$ 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 $\underbrace{p_1 \dots p_1}_i$ 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

Conveniences

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

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

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, `\{` 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 $\wedge\backslash([a-zA-Z]*\backslash)\backslash 1\$$
- You *cannot* do this with regular expressions; the program must keep the previous strings.
 - Especially useful with sed because of *substitutions*.

Other Utilities

Some very useful programs you can learn on your own:

`find` (search for files, e.g., `find /usr -name words`)

`diff` (compare two files' contents, output is easy for humans and programs to read (see all `patch`))

- May use `patch` to submit part of hw2

Also:

For many programs the `-r` flag makes them *recursive* (apply to all files, subdirectories, subsubdirectories, ...).

So “delete everything on the computer” is `cd /; rm -rf *`
(be careful!)