

---

# CSE 390a

## Lecture 7

Regular expressions,  
egrep, and sed

slides created by Marty Stepp, modified by Jessica Miller and Ruth Anderson

<http://www.cs.washington.edu/390a/>

# Lecture summary

---

- regular expression syntax
- commands that use regular expressions
  - `egrep` (extended grep) - search
  - `sed` (stream editor) - replace
- links
  - <http://www.panix.com/~elflord/unix/grep.html>
  - <http://www.robelle.com/smugbook/regexpr.html>
  - <http://www.grymoire.com/Unix/Sed.html> or
  - <http://www.gnu.org/software/sed/manual/sed.html>

# What is a regular expression?

---

```
"[a-zA-Z_\\- ]+@(( [a-zA-Z_\\- ])+\\. )+[a-zA-Z]{2,4}"
```

- **regular expression** ("regex"): a description of a pattern of text
  - can test whether a string matches the expression's pattern
  - can use a regex to search/replace characters in a string
  - regular expressions are extremely powerful but tough to read
    - (the above regular expression matches basic email addresses)
- regular expressions occur in many places:
  - shell commands (grep)
  - many text editors (TextPad) allow regexes in search/replace
  - Java Scanner, String split (CSE 143 grammar solver)

# egrep and regexes

---

command	description
egrep	<u>extended</u> grep; uses regexes in its search patterns; equivalent to grep -E

```
egrep "[0-9]{3}-[0-9]{3}-[0-9]{4}" faculty.html
```

- grep uses “basic” regular expressions instead of “extended”
  - extended has some minor differences and additional metacharacters
  - we’ll just use extended syntax. See online if you’re interested in the details.
- -i option before regex signifies a case-insensitive match
  - egrep -i "mart" matches "Marty S", "smartie", "WALMART", ...

# Basic regexes

---

"abc"

- the simplest regexes simply match a particular substring
- this is really a pattern, not a string!
- the above regular expression matches any line containing "abc"
  - *YES*: "abc", "abcdef", "defabc", ".=.abc.=.", ...
  - *NO*: "fedcba", "ab c", "AbC", "Bash", ...

# Wildcards and anchors

---

- . (a dot) matches any character except `\n`
  - `".oo.y"` matches "Doocy", "goofy", "LooPy", ...
  - use `\.` to literally match a dot `.` character
- `^` matches the beginning of a line; `$` the end
  - `"^fi$"` matches lines that consist entirely of `fi`
- `\<` demands that pattern is the beginning of a *word*;
- `\>` demands that pattern is the end of a word
  - `"\<for\>"` matches lines that contain the word "for"
- *Exercise* : Find lines in `ideas.txt` that refer to the C language.
- *Exercise* : Find act/scene numbers in `hamlet.txt` .

# Special characters

---

| means OR

- "abc | def | g" matches lines with "abc", "def", or "g"
- precedence of ^(Subject|Date) vs. ^Subject|Date:
- There's no AND symbol. Why not?

() are for grouping

- "(Homer|Marge) Simpson" matches lines containing "Homer Simpson" or "Marge Simpson"

\ starts an escape sequence

- many characters must be escaped to match them: / \ \$ . [ ] ( ) ^ \* + ?
- "\. \\n" matches lines containing ".\n"

# Quantifiers: \* + ?

---

\* means 0 or more occurrences

- "abc\*" matches "ab", "abc", "abcc", "abccc", ...
- "a(bc)\*" matches "a", "abc", "abcbc", "abcbcbc", ...
- "a.\*a" matches "aa", "aba", "a8qa", "a!?!\_a", ...

+ means 1 or more occurrences

- "a(bc)+" matches "abc", "abcbc", "abcbcbc", ...
- "Goo+gle" matches "Google", "Goooogle", "Goooooogle", ...

? means 0 or 1 occurrences

- "Martina?" matches lines with "Martin" or "Martina"
- "Dan(ie~~l~~)?" matches lines with "Dan" or "Daniel"

• *Exercise* : Find all ^^ or ^\_^ type smileys in chat.txt .



# More quantifiers

---

$\{min, max\}$  means between *min* and *max* occurrences

- "a(bc){2,4}" matches "abcbc", "abcbcbc", or "abcbcbcbc"
- *min* or *max* may be omitted to specify any number
  - "{2,}" means 2 or more
  - "{,6}" means up to 6
  - "{3}" means exactly 3

# Character sets

---

[ ] group characters into a character set;  
will match any single character from the set

- "[bcd]art" matches strings containing "bart", "cart", and "dart"
- equivalent to "(b|c|d)art" but shorter
- inside [ ], most modifier keys act as normal characters
  - "what[.!\*?]\*" matches "what", "what.", "what!", "what?\*\*\*!", ...
- *Exercise* : Match letter grades in 143.txt such as A, B+, or D- .

# Character ranges

---

- inside a character set, specify a range of characters with -
  - "[a-z]" matches any lowercase letter
  - "[a-zA-Z0-9]" matches any lower- or uppercase letter or digit
- an initial ^ inside a character set negates it
  - "[^abcd]" matches any character other than a, b, c, or d
- inside a character set, - must be escaped to be matched
  - "[+\-]?[0-9]+" matches optional + or -, followed by  $\geq$  one digit
- *Exercise* : Match phone #s in `faculty.html`, e.g. (206) 685-2181 .

# sed

---

command	description
sed	<u>s</u> tr <u>e</u> am <u>e</u> ditor; performs regex-based replacements and alterations on input

- Usage:
  - `sed -r "s/REGEX/TEXT/g" filename`
    - substitutes (replaces) occurrence(s) of regex with the given text
    - if *filename* is omitted, reads from standard input (console)
    - sed has other uses, but most can be emulated with substitutions
- Example (replaces all occurrences of 143 with 390):
  - `sed -r "s/143/390/g" lecturenotes.txt`

# more about sed

---

- sed is line-oriented; processes input a line at a time
  - -r option makes *regexes* work better
    - recognizes ( ), [ ], \*, + the right way, etc.
  - s for *substitute*
  - g flag after last / asks for a *global match* (replace all)
- special characters must be escaped to match them literally
  - `sed -r "s/http:\\/\\/https:\\/\\/g" urls.txt`
- sed can use other delimiters besides / ... whatever follows s
  - `find /usr | sed -r "s#/usr/bin#/home/billy#g"`

# sed exercises

---

- In movies.txt:
  - Replace “The” with “The Super Awesome”
  - Now do it only when The occurs at the beginning of the line.
  - (Need the next slide for this)
  - Move the year from the end of the line to the beginning of the line.
  - Do this and also sort the movies by year
  - Now do the two items above and then put the year back at the end of the line.

# Back-references

---

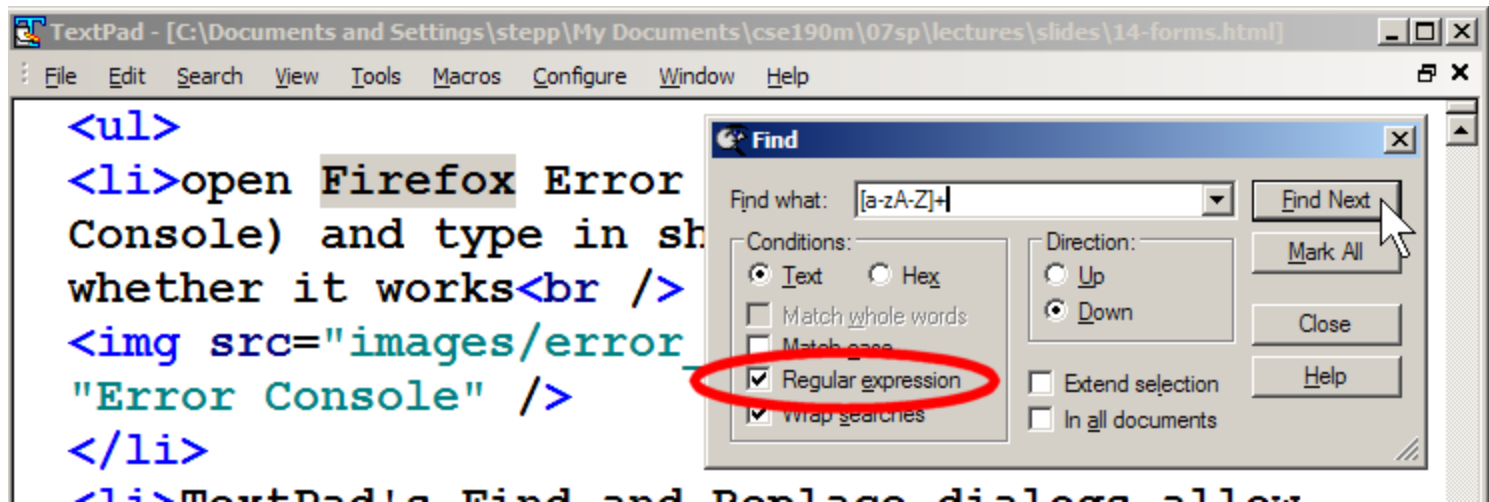
- every span of text captured by ( ) is given an internal number
  - you can use `\number` to use the captured text in the replacement
  - `\0` is the overall pattern
  - `\1` is the first parenthetical capture
  - ...
- Back-references can also be used in egrep pattern matching
  - Match "A" surrounded by the same character: `"(. )A\1"`
- Example: swap last names with first names
  - `sed -r "s/([^ ]+), ([^ ]+)/\2 \1/g" names.txt`
- *Exercise* : Reformat phone numbers with 685-2181 format to (206) 685.2181 format.

# Other tools

- find supports regexes through its -regex argument

```
find . -regex ".*CSE 14[23].*" 
```

- Many editors understand regexes in their Find/Replace feature





# Exercise

---

- Write a shell script that reads a list of file names from `files.txt` and finds any occurrences of MM/DD dates and converts them into MM/DD/YYYY dates.
  - Example:  
04/17
  - would be changed to:  
04/17/2011

# Yay Regular Expressions!

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!



BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

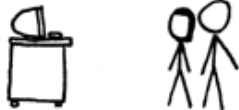


IT'S HOPELESS!

EVERYBODY STAND BACK.



I KNOW REGULAR EXPRESSIONS.



Courtesy XKCD