CS265 Advanced Programming Techniques

Regex

Pattern Matching

- Regular Expressions
  - `grep`
  - `egrep` – extended grep
  - `fgrep` – “fixed” string grep
  - Editors like vi, sed, emacs, …
  - Other tools
- Wildcard matching
  - Bash shell is doing wildcard matching
  - aka filename expansion
  - aka filename matching

Wildcard Matching

Shell Metacharacters

Symbol	Meaning
>	Output redirection
>>	Output redirection (append)
<	Input redirection
<<	Input redirection (append)
*	File substitution wildcard; zero or more characters
?	File substitution wildcard; one character; any character between brackets
[]	File substitution wildcard; any character between brackets
\`cmd\`	Command substation (back quotes to the left of 1)
$(cmd)	Command substitution

Symbol	Meaning
|	The pipe
;	Command sequence
||	OR conditional
&&	AND conditional
()	Group commands
&	Run command in the background
#	Comment
$	Expand the value of variable
\	Prevent/escape interpretation of next character

special characters for filename expansion

Special characters for double quotes (plus ..)

Quotes

What is the difference between these commands

```
% ls a*t

% ls 'a*t’

% ls “a*t”
```

Quotes

- Single quotes
  - Single quotes preserve the literal value of each character within the quotes
  - The shell does not try to interpret the special characters inside single quotes

- Double quotes
  - Double quotes can also be used to protect special characters
  - The shell will interpret $, \ and `..` inside the double quotes.

Single Quotes vs Double Quotes - Try these

```
a=apple
echo “$a”
echo ‘$a’
echo “’$a’”
echo ‘”$a”’
echo “red$arocks”
echo “red$a”
echo ‘”’
echo “’”
echo ‘\”’
echo ‘\’’ (invalid)
echo “\’”
echo “\””
```

Wildcards (aka File Name Substitution)

- Allow user to refer to several files at once
- How to list all files in the current directory that start with ‘a’?

```
ls a*
```

? – Matches a single character

```
ls a?.txt

a1.txt a2.txt ab.txt


ls lab1.???

lab1.doc lab1.pdf
```

* - Matches several characters

```
ls a*.txt

a1.txt a2.txt abcd.txt

abc.txt a.b.txt ab.txt
```

```
ls lab1.*

lab1. lab1.c lab1.doc

lab1.docx lab1.pdf
```

[…] - Matches any character

Matches any one of the listed characters

```
ls lab[123].pdf

lab1.pdf lab2.pdf lab3.pdf


ls a[ab]*.???

abcd.txt abc.txt ab.txt
```

Regular Expressions

`grep` Command

- Outputs all lines in the input that match the given regular expression

```
grep [options] regex [file …]
```

- e.g.

```
grep hello *.txt
```

- outputs all lines containing `hello` in any file that ends in `.txt` in the current directory

Regular Expressions

- A regular expression is a special string (a sequence of characters)
- Describes a search pattern, i.e. each regular expression matches a set of strings
- `grep` uses regular expressions to search the contents of files
- Looks like wildcards but is quite different!

Regular Expressions - Literals

- Letters and numbers are literal - that is they match themselves:
- The regular expression

```
foobar
```

- matches only the string

```
foobar
```

. Matches exactly one character

- The regular expression

```
fooba.
```

- Matches the following strings

```
foobar

foobat

foobay

etc.
```

. Matches exactly one character

- Each dot must match exactly one character
- The regular expression

```
f..bar
```

- matches

```
foobar or fWRbar
```

- but not

```
fubar or fooobar
```

[…] Matches any listed character

- The regular expression

```
foob[aeiou]r
```

- matches only the 5 strings

```
foobar
foober
foobir
foobor
foobur
```

* zero or more of the last character

- The regular expression

```
fo*
```

- matches

```
f
fo
foo
fooo
foooo
etc.
```

Big difference with how * operates in a regular expression vs as a wildcard

* zero or more of the last character

- The regular expression

```
[0-9][0-9]*
```

- matches all decimal numbers of at least one digit including ones with leading zeros such as

```
0

1

10

00

000042
```

Big difference with how * operates in a regular expression vs as a wildcard

* zero or more of the last character

- The regular expression

```
        *
```

- matches anything
- including an empty string

^$ - Beginning and end of line

- The regular expression

```
^foobar
```

- matches any line that starts with `foobar`
- The regular expression

```
foobar$
```

- matches any line that ends with `foobar`

`grep` Command

grep

- Let’s say you want to search for any word that starts with `b` followed by `0` or more `a`’s in file `a.txt`
- The following will not work

```
grep ba* a.txt
```

- Why not?

grep – another example

- The command

```
grep [a-c]* chap[12]
```

- The shell may transforms this to (depending on the directory files)

```
grep array.c bug.c comp.c chap1 chap2
```

- Which looks for the pattern `array.c` in the files `bug.c comp.c chap1` and `chap2`.

- To bypass the shell and go directly to grep use quotes

```
grep ”[a-c]*” chap[12]

grep ‘[a-c]*’ chap[12]
```

`grep` Options

`-i` case-insensitive search (don’t distinguish between a and A)
`-v` invert search (output lines which don’t match)
`-l` Output only the names of files with matching lines
`-c` Output only the number of lines that match

grep – Interesting Cases

Removes all lines beginning with ‘#’

```
        grep -v '^#'
```

Removes all lines which are either empty or contain only spaces

```
        grep -v '^[ ]*$'
```

`fgrep` –fixed string grep

- Like `grep`, `fgrep` searches for things but does not do regular expressions, just fixed strings

```
fgrep 'hello.*goodbye'
```

- Searches for string `“hello.*goodbye”` but does not match it as a regular expression

`egrep` –Extended grep

- `grep` interprets only basic regular expressions
- Extended regular expressions use additional metacharacters to allow expression of more elaborate search patterns
- Use `egrep` if you require this

? – 0 or 1 of the last character

- The regular expression

```
[1-9][0-9]?
```

- matches all numbers from 1 to 99
- The regular expression

```
colou?r
```

- matches

```
color
colour
```

| - Used as an OR

- The extended regular expression

```
        0|[1-9][0-9]?
```

- matches all numbers from 0 to 99
- Parentheses can be used as well

Regular Expressions

Regular Expressions

- Text patterns that define a set of strings
  - Called a regular language
- Used by many utilities:
  - `vi`, `less`, `emacs`, `egrep`, `sed`, `awk`, `ed`, `tr`, `perl`, etc.
- Used by many programming languages:
  - Javascript, .NET, Java, Perl, Python, Ruby, ..
- Note, syntax varies slightly between utilities
- Very handy

Regular Expressions

Regular expressions are used to

- find text that matches a pattern
- search and replace text that matches a pattern
- validate that input data fit into a given pattern
- rearrange text (split, etc.)
- simplify text processing and programming tasks

- regex, regexp, regexprs

Grep

- *`grep`* is derived from the `g/re/p` command that performed a regular expression search in the Unix text editor *`ed`*
- `g/re/p` meant to globally (`g`) search for `re` and print (`p`)
- `grep` was so popular that all Unix systems now have a dedicated grep utility

Grep

Outputs all lines in the input that match the given regular expression

```
    grep [options] regex [file …]
```

e.g.

```
    grep hello *.txt
```

outputs all lines containing `hello` in any file that ends in `.txt` in the current directory

Grep Options

```
grep [options] regex [file …]
```

Options

`-i` case-insensitive search (don’t distinguish between a and A)

`-v` invert search (output lines which don’t match)

`-l` Output only the names of files with matching lines

`-c` Output only the number of lines that match

grep, fgrep and egrep

- fgrep is faster grep and does not do regular expressions, just fixed strings with some wildcards

```
fgrep 'hello.*goodbye'
```

- egrep (extended grep)
  - grep interprets only basic regular expressions
  - egrep uses additional metacharacters to allow expression of more elaborate search patterns
  - Use egrep if you require this

Finding strings with egrep

- `egrep` is a handy tool for searching text files

```
    egrep regex file(s)
```

- If no files are provided, `egrep` reads `stdin` (behaves asa filter)

```
    $ egrep Waldo *.locations
    ...
    $ who | egrep Waldo
```

- You might also use `awk`, search in `vim` or `emacs`, etc.

Primitive Operations for Regular Expressions

Primitive Operations (define REs)

c Any literal character matches itself

r* Kleene Star – matches 0 or more

r1r2 Concatenation – r1 followed by r2

r1|r2 Choice – r1 or r2

(r) Parentheses are used for grouping, to force

evaluation

\ Escape character (Turns off special meaning of metacharacters)

| - Union (used as an OR)

- To get any line that contains `by` or `waves`:

```
$ egrep ‘by|waves’ input1
pass by in your car
He waves to you
```

- Note, | is a shell metacharacter
- Use quotes to keep the shell’s hands off of it
- Use parentheses to force evaluation

```
$ egrep ‘(Y|y)ou’ input1
You see my cat
pass by in your car
He waves to you
```

Concatenation

- Hopefully, explained already.

  `(a|b)c\.(log|txt)` matches a string that:

  Starts with a or b
  followed by c
  followed by the literal .
  ending with, either, txt or log

- Note, the period was escaped (explanation follows)

* - Closure

`R*` – R, matched 0 or more times.

- * modifies the previous RE
- It does not match anything on its own
- ab*c matches ac abc abbc abbbc abbbbc ...
- (ab)*c matches c abc ababc abababc ...

Common Syntax

. Matches any single character

[.]{n} Matches any single character exactly n times

[.]{n,m} Matches any single character exactly between n and m times

[( )] Allows us to group several characters to behave as one

[R?] Zero or one occurrences of R

[R+] One or more occurrences of R

[...] Character class – matches any single character in brackets

[^...] Character class, inverted (negation)

Anchors

- ^ Beginning of line

```
^a
```

- $ End of line

```
z$
```

- \< \> Word anchors

```
\<word\>
```

. any character

- matches any character except the special ones (metacharacters)
- special characters: `$()*+.?[]\^{}|`

```
egrep ‘.ou’ input1
```

You see my cat
pass by in your car
He waves to you

[] – character classes

- Matches any single character in the brackets:
- [brc]at matches bat rat cat
  - Not Bat
- Careful! [Y,y]ou matches You you ,ou
- [ab]*yz matches yz ayz byz abyz aayz bbyz bayz

abbayz bbbbbbbbbbbbbbabbbbbbbbbbbbbbyz ...

- Very few characters have special meaning inside the brackets:
  - - Range, if it's not the first character
  - ^ Negation of class, if it is the first character

Ranges in character classes

- is used to create ranges inside a character class.

- 0x[0-7] matches 0x0 0x1 … 0x3 … 0x7
  - Not 0x8
- [cl-n]ode matches code lode mode node
- [c,l-n]ode also matches ,ode
- [a-zA-Z] matches any single letter
- [a-Z] doesn’t match anything (if using ASCII)
  - See http://www.asciitable.com/
- [A-z] also matches [\ ] ^ _ `
- To match the - character, place it first:
- [-ln]ode matches -ode lode node

^ - invert character class

- ^ negates the notion of the match, if it appears first.
- [^C] matches any character not in C
- [^rbc]at matches hat zat Cat Bat sat …
  - Not rat bat cat at
- To match the ^ character, put it elsewhere:
- [r^bc]at matches rat bat cat ^at

POSIX bracketed expressions

These are widely implemented.

(Note, they, in turn, need to be in brackets.)

[:alnum:] [:alpha:] [:ascii:]

[:blank:] [:cntrl:]   [:digit:]

[:graph:] [:lower:] [:print:]

[:punct:] [:space:] [:upper:]

[:word:]   [:xdigit:]

Pre-defined character classes (GNU Utilities)

- Some classes are so popular, they have nicknames:
  - \d any numeric digit
  - \w word character (alphanumeric or _) (equivalent to [:alnum:])
  - \s whitespace
- These classes are also inverted:
  - \D any character, not a digit
  - \W not a word character
  - \S not whitespace

Line anchors ‵ and ′

- They provide context for a regex
- They do not match any characters

```
$ egrep ‘[Yy]ou’ input1
You see my cat
pass by in your car
He waves to you
```

- Use the caret (^) to anchor the beginning of a line:

```
$ egrep ‘^[Yy]ou’ input1
You see my cat
```

- Use the dollar sign ($) to anchor the end of a line:

```
$ egrep ‘[Yy]ou$’ input1
He waves to you
```

Word anchors

- Use \< and \> to match the beginning/end of a word

```
$ egrep “\<[Yy]ou\>” input1
You see my cat
He waves to you

$ egrep “our\>” input1
pass by in your car
```

Used Responsibly Regular Expressions are a plus

- Validate Phone Numbers

```
"^\(*\d{3}\)*( |-)*\d{3}( |-)*\d{4}$”
```

- Getting the trailing folder from a path

```
"[^\\]+\\*$"
```

Question to the class..

What does this regular expression match?

```
\S+@\S+
```

Regex for Validating Email Address

- Simplistic match `\S+@\S+`

- Better (99.99% accuracy)– built from RFC5322 Official Standard

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-
]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-
\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-
z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-
9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-
\x09\x0b\x0c\x0e-\x7f])+)\])
```

- Even better?

http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

Regular Expressions are greedy

- REs are, by default, greedy
- Will match the longest string they can

Search and replace using `vi`

```
:%s/search/replacement/g
```

`:` go to the command line

`%` to substitute all lines, could also use

`5,10` substitute in lines 5-10

`.` substitute in current line only (same if . Is missing)

`+10` (substitute the next 10 lines)

`search` a regular expression that describes the string to change

`replacement` what we replace the string with

`g` make the substitution globally in the qualifying lines

default changes only the first occurrence

Search and Replace using `sed`

- `sed` stands for stream editor and it can be used for text manipulations (searching, search and replacing, inserting and deleting)
- Mostly used for searching and replacing, e.g.,

```
sed 's/word1/word2/g' input.file > output.file
```

- replaces every instance of `word1` with `word2` in `input.file` and stores the output in `output.file`

Epilogue

- Various utilities use slightly different flavors
- Some, e.g., treat a particular character as special, while others want them escaped to invoke their special behavior
- `man regex` might be helpful

Lessons

- Lesson 1: Regular expressions are great for pattern matching
- Lesson 2: Different utilities use different regular expressions, so be careful

/regex/

Acknowledgements

These slides are copied or inspired by the work of the following courses and people:

- CS265 & CS571, Drexel University, Kurt Schmidt, Jeremy Johnson, Geoffrey Mainland, Spiros Mancoridis, Vera available at https://www.cs.drexel.edu/~kschmidt/CS265/
- Bil Tzerpos's class