What do you think?

Work with a partner(s):

Write a command that finds the phrase `#!/bin/bash` in all the files in your directory tree.

Write a command that finds all the files containing the phrase `bash` in your home directory.

How are you using grep or globbing in one or both of these commands?

CSE 374 Lecture 6

Regular expressions and grep

Globbing and Regex

Globbing: the shell filename expansion; matches some patterns

Regular expressions (regex): a set of rules for matching patterns in text

(We see regular expressions in math, as formal grammars in cs, and other variations as well. Different applications (egrep) my have slightly different rules.)

Regex theory

1. A set of rules for matching a pattern (P) to a string (S)
2. All strings are made of a combination of the null (empty) set, the empty string ε, and a single character.
3. Regular expressions match a string if
   3.1. P is a literal character (a, b, …) that matches the string S
   3.2. P_1P_2 matches S if $S = S_1S_2$ such that P_1 matches S_1 and P_2 matches S_2
   3.3. $P_1|P_2$ matches S if P_1 matches S OR P_2 matches S
   3.4. P* matches S if there is an *i* such that P...P (*i* times) matches S. Includes *i=0* which matches ε.

Regex rules

Regular expressions have

Characters: the literal characters [a b 9] (S is an exact duplicate of P)

Anchors: sets the position in the line where P may be found (^ or $)

Modifiers: modify the range of text P may match (* or [set_of_chars])

Note: Regex details & implementation may vary between application, but general rules apply.

Regex special characters *(some are \escaped)*

\ : escape following character

‘.’ : matches any single character at least once

$p_1|p_2$: matches p_1 OR p_2

‘*’ : matches zero or more of the previous p

‘?’ : matches zero or one of the previous p ($\varepsilon|p$)

‘+’ : matches one or more of previous p (pp*)

() : group patterns for order of operations

{} : repeat n times

[] : contain literals to be matched (single or range)

^ : Anchors to beginning of line

$: anchors to end of line

<> : word boundaries

Classes of characters

```
.      == any character
[a-z]  == a, b, c, d … z
[A-Z]  == A, B, C, D … Z
[0-9]  == 0,1,2,3,4,5,6,7,8,9
abc    == literally abc
```

```
c.t          → cat, cut, cot
[Hh]ello!    → Hello!, hello!
[BLERG] -> B, L, E, R or G
[0-5][5-9]   → 15,16,17,18,19
               25,26,27,28,29
               35,36,37,38,39
               45,46,47,48,49
               55,56,57,58,59
```

Repipipitititition

* == *zero or more*, a* → {, a, aa, aaa, aaaa, …}

+ == *one or more*, a+ → {a, aa, aaa, …}

? == *zero or one* of the preceding, a? → {, a}

{n} == exactly n repetitions of the preceding, a{3} → aaa

a|b == a or b, this|that|when|how → this, that, when, how

All but | are **POSTFIX OPERATORS** (they come after the pattern)

Invisible characters

^ == the start of a line

$ == the end of a line

\t == a tab

? == *zero or one* of the preceding

Extras

[^abc] : matches everything NOT abc

‘*’ : is greedy; matches as much as possible

Grep Regex

By default, grep matches each line against .*p.*

You can anchor the pattern with ^ (beginning) and/or $ (end) or both (match whole line exactly)

These are still “real” regular expressions

Backreference & repeated matches

Up to 9 times in a pattern, you can group with (p) and refer to the matched text later!

You can refer to the text (most recently) matched by the nth group with \n.

Simple example: double-words ^\([a-zA-Z]*\)\1$

You cannot do this with actual regular expressions; the program must keep the previous strings.

\(p\)\{n\} will match the p n times. \{n,m\} matches at least n, but not more than m times.

Bash Regex Gotchya's

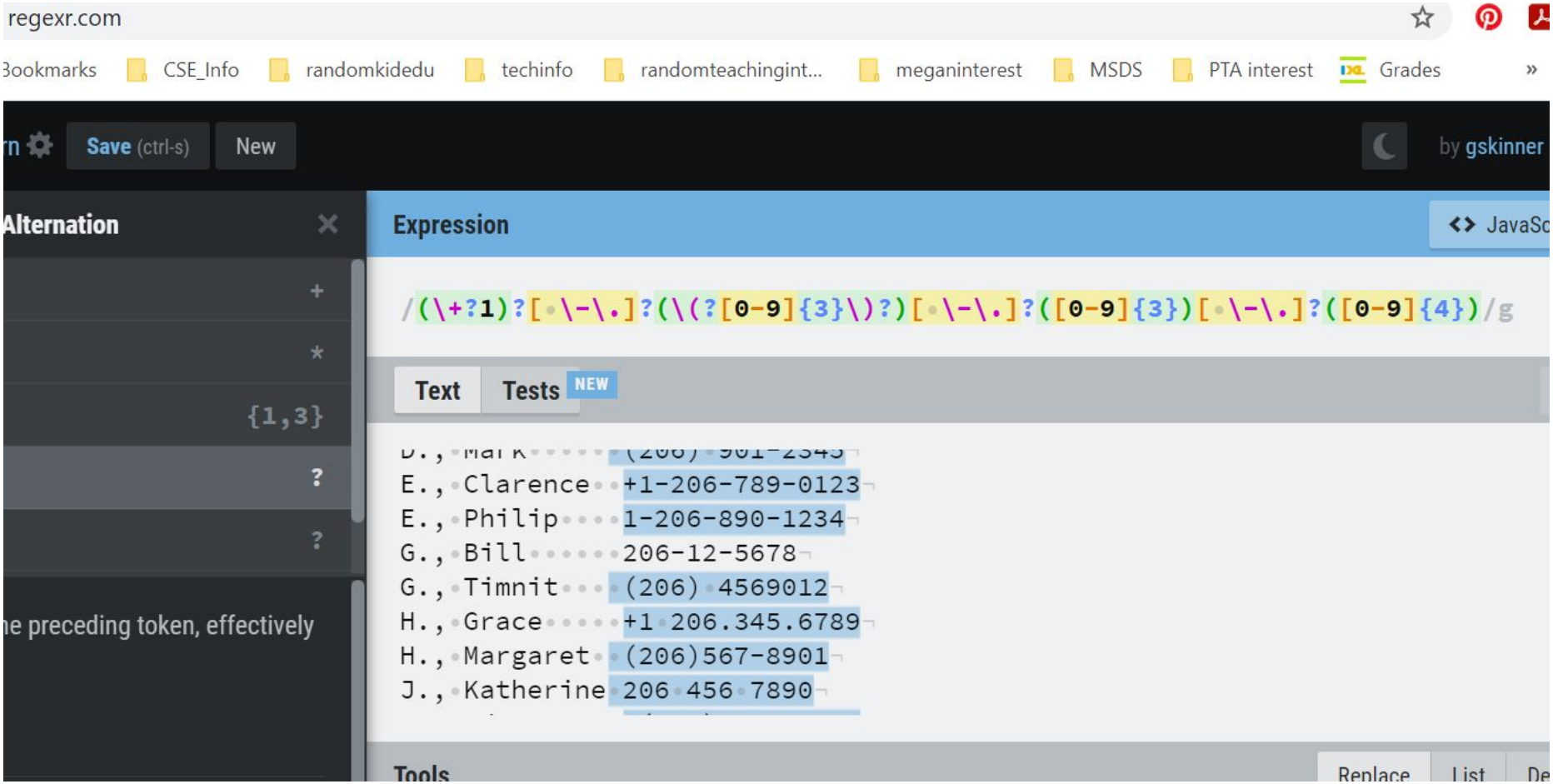
- Modern (i.e., gnu) versions of grep and egrep use the same regular expression engine for matching, but the input syntax is different for historical reasons
    - For instance, \{ for grep vs { for egrep – See grep manual sec. 3.6
- Must quote patterns so the shell does not muck with them – and use single quotes if they contain $ (why?)
- Must escape special characters with \ if you need them literally: \. and . are very different
    - But inside [ ] many more characters are treated literally, needing less quoting (\ becomes a literal!)

Regular expressions and Grep

Can you write a regular expression to identify every phone number?

\ : escape following character
‘.’ : matches any single character at least once
$p_1|p_2$: matches p_1 OR p_2
‘*’ : matches zero or more of the previous p
‘?’ : matches zero or one of the previous p ($\varepsilon|p$)
‘+’ : matches one or more of previous p (pp*)
() : group patterns for order of operations
{} : repeat n times
[] : contain literals to be matched (single or range)
^ : Anchors to beginning of line
$: anchors to end of line
<> : word boundaries

```
D., Mark       (206) 901-2345
E., Clarence  +1-206-789-0123
E., Philip    1-206-890-1234
G., Timnit    (206) 4569012
H., Grace     +1 206.345.6789
H., Margaret  (206)567-8901
J., Katherine 206 456 7890
L., Ada       (206) 123-4567
L.,Jerry      2061235678
O., Ellen     206 2346789
T., Alan      206-234-5678
W., Jeannette 206 678.9012
```


regexr.com
Bookmarks
CSE_Info
randomkidedu
techinfo
randomteachingint...
meganinterest
MSDS
PTA interest
Grades
Save (ctrl-s)
New
by gskinner
Alternation
+
*
{1,3}
?
?
he preceding token, effectively
Expression
JavaSc
/(\+?1)?[.\-\.]?(\(?[0-9]{3}\)?)[.\-\.]?([0-9]{3})[.\-\.]?([0-9]{4})/g
Text
Tests
NEW
D.,Mark (206) 901-2345
E.,Clarence +1-206-789-0123
E.,Philip 1-206-890-1234
G.,Bill 206-12-5678
G.,Timnit (206) 4569012
H.,Grace +1 206.345.6789
H.,Margaret (206)567-8901
J.,Katherine 206 456 7890
Tools
Replace
List

What is ‘sed’?

Run ‘man sed’ now!

Stream editor: makes basic text transformations on an input stream

Use ‘sed *command* file[s]’

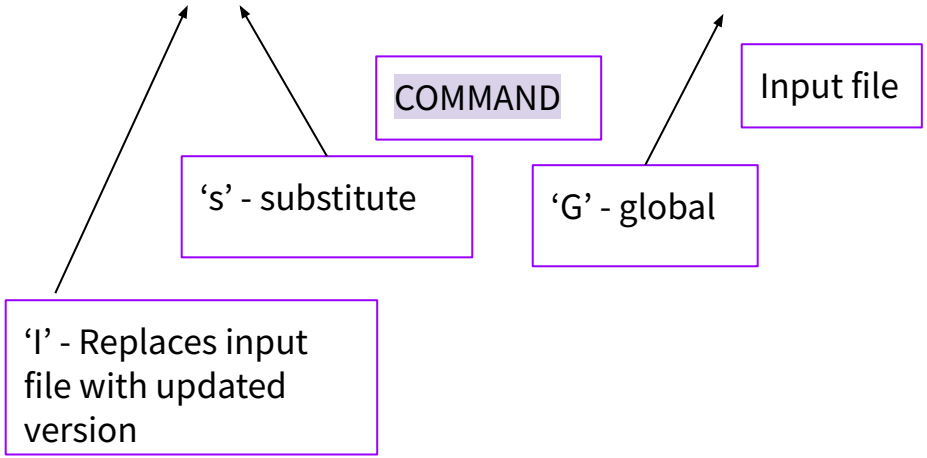
Changes line by line, one pass through

Basic usage: sed

$ sed [OPTIONS] [COMMAND] [FILE]

$ input_stream | sed [COMMAND]

$ sed -i 's/original/replacement/g' test.txt



Useful options:

-i : replace input file with edited version

-e : allows for multiple commands - applies each left to right (`sed -e 's/a/A/' -e 's/b/B/' <old >new`)

-f : reads command from a file

-n : suppresses output except when told otherwise

Omitting file applies [COMMAND] to stdin

Sed cycle

1. Read one line from input stream
2. Put in pattern space without trailing /n
3. Execute command
    a. commands with address are only executed if address is verified
4. Pattern space is printed to the output stream

Addresses

Addresses apply sed only to specific lines. Address comes before command.

Number : only that line number

$: last line of input

First~step : every ‘step’ lines starting with ‘first’

/regexp/ : only lines matching the regular expression

l1,l2: range - between line that matches l1, and line that matches l2 (l1&l2 can be numbers or regex)

Other types of commands

‘P’ : print this line (often used with ‘-n’ to suppress printing of non-marked lines)

‘d’ : delete this pattern space and continue

‘y’ : transliterate characters

‘a’: append text

‘i’ : insert text

‘c’ : replace text

```
sed -n 's/pattern/&/p' <file
```

```
$ echo hello world | sed
'y/abcdefghij/0123456789/'
74llo worl3$
```

```
$ seq 3 | sed '2i hello'
1
hello
2
3
```

```
$ seq 10 | sed '2,9c hello'
1
hello
10
```

sed - more ideas

- Sed encounters one line at a time, and does one pass of the input.
- Delimiter ‘/’ can be changed to anything, like ‘_’ or ‘:’ - may help if COMMAND contains many ‘/’
- Multi-line editing is possible, but painful, with sed (with ‘hold buffer’). Use another scripting program (like ‘awk’).
- Branches are also possibly (‘b’ and ‘t’ commands)
- Use backreferences (\1, \2 etc) to refer back to regex gathered with \(to \)

What about ‘awk’

Or perl? Or ed? Or ruby?

Special purpose language for text editing on an input stream. More programming concepts, used for bigger commands.

Many scripting choices, often with more functionality. Sed stands as the quickest, easiest, and standard on *nix systems for simple commands.