CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

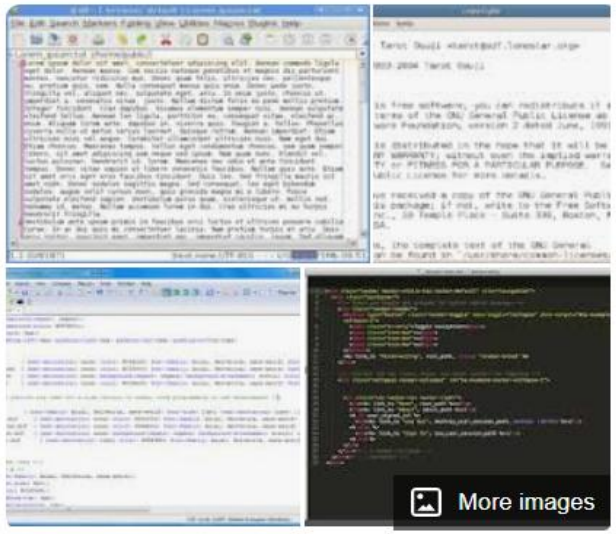
Subject Matter

- Regular Expressions
- SED

Your Goals

- HW2
- Practice practice practice …

Aside: text editors



A text editor is a type of computer program that edits plain text. Such programs are sometimes known as "notepad" software, following the naming of Microsoft Notepad. Wikipedia

Vi (Vim)

- Move around, mark edit using letter keys <hjkl>
- Get to insert mode by typing 'i', 'esc' out
- Get to menu by typing ":"
- Save and quit ":wq", no-save and quit: ":q!"
- Vim is 'improved' vi - more powerful
- Vimtutor at prompt

Emacs

- Endlessly extendable; can use arrows to move
- Menu commands use C (control) or M (meta/alt)
- Quit: C-x C-c
- Tutorial, C-h for help

Emacs

```
 1 ;; use spaces to indent
 2 ;; Note - can be over-ridden for other modes
 3 (setq-default indent-tabs-mode nil)
 4
 5 ;; change the default tab width to 2 spaces
 6 ;; and then use that width.
 7 (setq-default tab-width 2)
 8 (setq indent-line-function 'insert-tab)
 9
10 ;; always display line numbers
11 (global-display-line-numbers-mode 1)
12
13 ;; Don't use delete mode
14 (normal-erase-is-backspace-mode 0)
```

Try
M-x doctor
M-x snake
M-x morse-region

Grep matching

- Looks at one line at a time to match a pattern
- Default is to match .*p.*
  - (just means that anything can come before or after the string)
- But you can anchor a pattern with ^ (beginning) and/or $ (end) or both.

```
[mh75@calgary scripts]$ ls -R ~ | grep demo
demoscript
shiftdemo
[mh75@calgary scripts]$ ls -R ~ | grep '^demo'
demoscript
[mh75@calgary scripts]$ ls -R ~ | grep 'demo$'
shiftdemo
[mh75@calgary scripts]$ ls -R ~ | grep '^demo$'
[mh75@calgary scripts]$
```

Regular Expressions

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

Basic Regex

Characters: 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])

Special characters in Regex

**** : escape following character

() : group patterns for order of operations

. : lea

p₁|

***** : pre

? : matches zero or one of the previous p (***ε***|p)

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

$: anchors to end of line

<> : word boundaries

Some of these characters are escaped, using \

Some are not

Which is which depends on implementation

Character Classes

. → 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

Character Classes (more demo)

. → 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

```
$ cat testfile
Megan teaches 374
kayak
rubber duck
$ grep '[u-z]' testfile
kayak
rubber duck
$ grep 'teaches' testfile
Megan teaches 374
$ grep [A-Z].*[0-9] testfile
Megan teaches 374
```

Invisible characters

^ → start of a line

$ → end of a line

\t → a tab

\s or [[:space:]] → white space

[^abc] → NOT abc

```
$ cat testfile
Megan teaches 374
wow kayak rubber duck
rahrah yeah yeah wow    wow
$ grep '^wow' testfile
wow kayak rubber duck
$ grep 'wow$' testfile
rahrah yeah yeah wow    wow
$ grep -P '([a-z]*)\t\1' testfile
rahrah yeah yeah wow    wow
$ grep '\([a-z]*\)[[:space:]]\1'
testfile
Megan teaches 374
wow kayak rubber duck
rahrah yeah yeah wow    wow
$ grep ‘[^0-9]’ testfile
Megan teaches 374
```

Repetition

* → zero or more

+ → one or more

? → zero or one

{n} → exactly n repetitions

A|b → A or b

```
$ cat testfile
wf wof woof wooooof
$ grep 'w[o]*f' testfile
wf wof woof wooooof
$ grep 'w[o]\+f' testfile
wf wof woof wooooof
$ grep 'w[o]\?f' testfile
wf wof woof wooooof
$ grep 'w[o]\{2\}f' testfile
wf wof woof wooooof
$ grep 'woof\|wof' testfile
wf wof woof wooooof
```

Backreferences

- 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$
- \(p\)\{n\} will match the p n times. \{n,m\} matches at least n, but not more than m times.
- You cannot do this with actual regular expressions; the program must keep the previous strings.

```
$ cat testfile
kayak rubber duck
$ grep '\([a-z]\)\([a-z]\).*\2\1' testfile
kayak rubber duck
$ cat testfile
wf wof woof wooof wooooof
$ grep 'w[o]\{2,4\}f' testfile
wf wof woof wooof wooooof
```

Demo

```
$ ls /usr/share/dict/words
$ ln –s /usr/share/dict/words words
$ wc words
```

```
wget
https://courses.cs.washington.edu/courses/cse374/26sp/assignments/numberslist
```

Gotchyas

- 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 glob 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!)

Try this with Grep

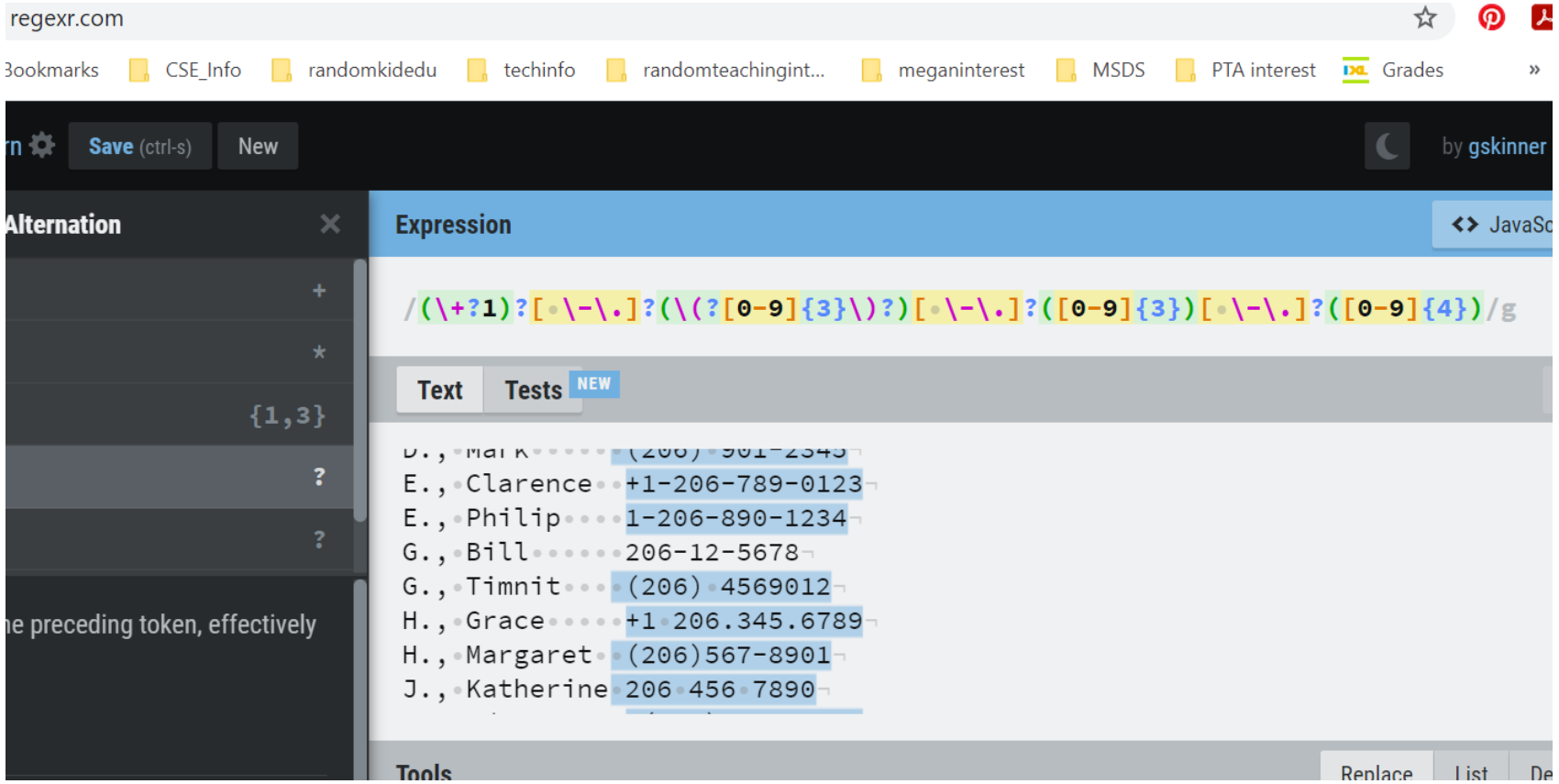
Can you write a regular expression to identify every phone number?

\ : 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
```

(download 'numberslist' from website)

https://regexr.com/



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 sed

```
$sed [OPTIONS] [COMMAND] [FILE]
$input_stream | sed [COMMAND]
$sed –i 's/orig/replace/g’ FILE
```

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`

Basic sed (more)

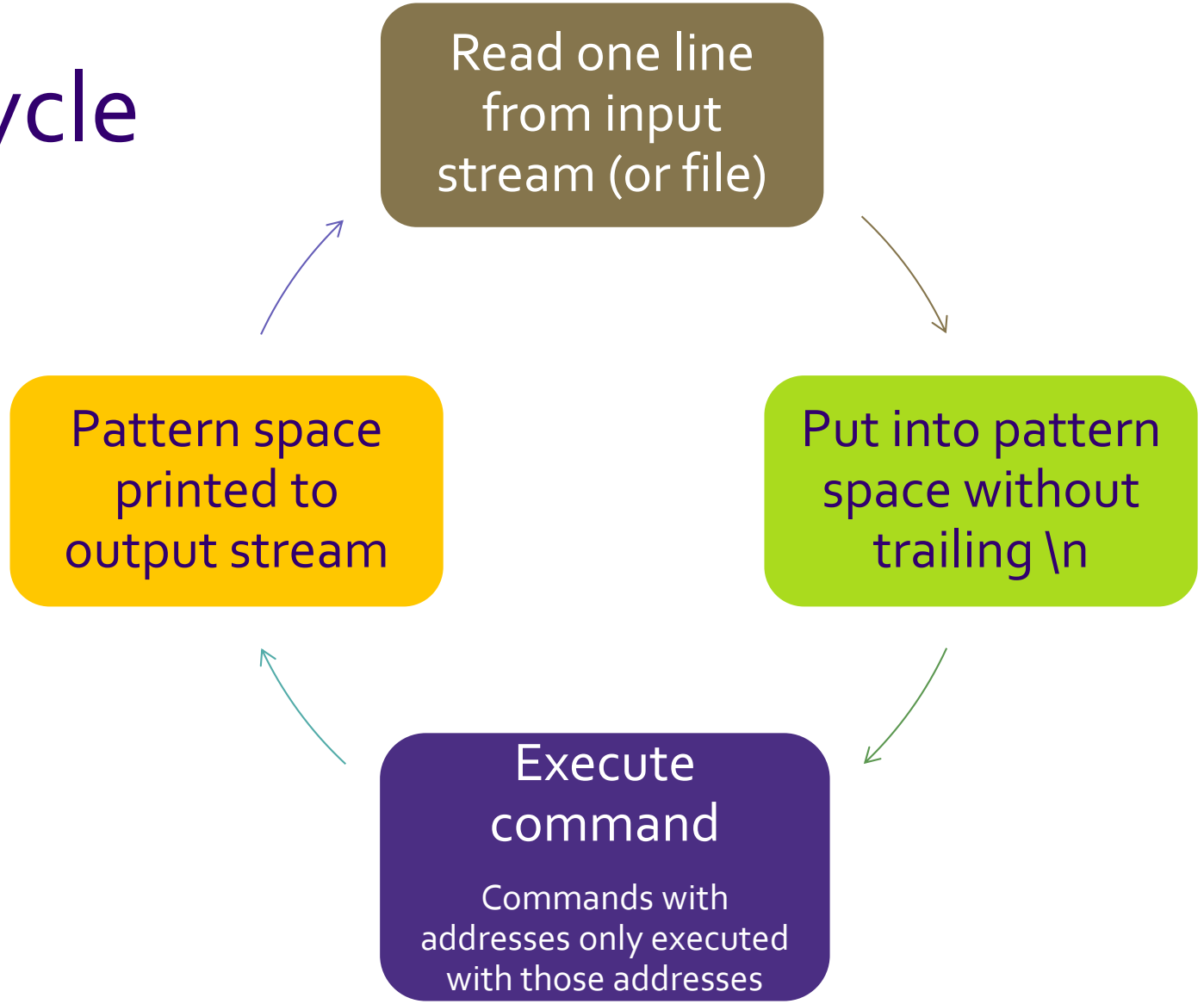
```
$sed [OPTIONS] [COMMAND] [FILE]
$input_stream | sed [COMMAND]
$sed -i 's/orig/replace/g' FILE
```

`FILE:` An input file, could be more than one.

`-i:` an example of an OPTION (flag) – this replaces the original file with an edited one.

`s/orig/replace/g:` the command, enclosed in single quote to avoid globbing. S indicates substitution, orig& replace are regex patterns, g indicates global.

Sed cycle


Read one line from input stream (or file)
Put into pattern space without trailing \n
Execute command
Commands with addresses only executed with those addresses
Pattern space printed to output stream

Sed 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)

Sed 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

```
$ 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 Demo

```
wget https://courses.cs.washington.edu/courses/cse374/26sp/assignments/numberslist
```

Sed additional thoughts

- 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?)

- `awk` allows you to edit an input stream with more power than `sed`
- All of these, though, are scripting languages with their own flavor and their own power. `sed` is one of the most basic.