Little Languages

Lecture 24:

sed and Capturing Groups

VM day today!

Pull 590-material from upstream.

sed - Streaming Text Editor

- You've experienced a couple utilities that print *text data* to **`stdout`**
  1. cat - reads lines of input and prints them
  2. egrep - reads lines of input and prints those matching a regular expression

- Today we'll briefly tour another that allows us to *edit* input before printing
  - **sed** - "Streaming **ed**" - You *do* remember **ed** right? The original interactive editor.
  - Developed in 1973-1974 at Bell Labs by Lee McMahon

- **sed**'s little language adds commands around to regular expressions, i.e.:
  - **d**elete lines that match some regular expression
  - **i**nsert a line above a matching line
  - **a**ppend a line after a matching line
  - **s**ubstitute one regular expression with some text (search/replace)

`sed`'s -E Flag is for Extended Regular Expressions

- Throughout this course we're using *extended regular expression* syntax.
  - This is why we used `egrep` instead of grep (reminder: grep *also* has an -E flag)
  - The syntax you're implementing in `thegrep` is *extended regular expression* syntax
  - Most modern programming language's regular expression support follows this style

- Common pattern of a sed command:

```
sed -E <command> file
```

Inserting Lines into a Stream

- If some regular expression pattern is matched, then ***insert*** a new line of text ***before*** the matched line in the output stream.

```
sed -E '/regular expression/i <inserted text>' file
```

- The slashes are *delimiters* in sed. You can use any delimiter character you'd like, but the forward slash is most idiomatic.
  - If your regular expression pattern involves forward slashes, choosing another delimiter allows you to avoid escaping every forward slash in your pattern.

- Let's try inserting lines into Robert Frost's *The Road Not Taken*

```
sed -E '/the/i ~~~THE~~~' poem.txt
```

Appending Lines into a Stream

- If some regular expression pattern is matched, then ***append*** a new line of text ***after*** the matched line in the output stream.

`sed -E '/regular expression/a <inserted text>' file`

- This is just like insert, except the line follows.

- Let's try appending lines into Robert Frost's *The Road Not Taken*

  `sed -E '/^$/a ###\n' poem.txt`

Deleting Lines from a Stream

- If some regular expression pattern is matched, then ***filter*** the line from the output stream.

```
sed -E '/regular expression/d' file
```

- Let's try appending lines into Robert Frost's *The Road Not Taken*

```
sed -E '/^$/d' poem.txt
```

Substituting Values - "Search / Replace"

- It's often useful to search for some pattern and replace it inline.

- The **s**ubstitute command allows you to do so:

```
sed -E 's/regular expression/replacement/g' file
```

- Notice:

1. The **s** for substitute comes *before* the delimited pattern.
2. The delimiter separates the regular expression from its replacement.
3. The **g** at the end signifies "global"
   - Without it, only the first match of the regular expression on each line is replaced.
   - With it, all matches on each line are replaced.

- Let's try playing with substitutions in phone-numbers.txt

Example Substitutions

- The phone-numbers.txt has data in the following format:

```
504-621-8927
810-292-9388
856-636-8749
```

- You can substitute dashes with dots using sed (compare with g flag):

```
$ sed -E 's/-/./g' phone-numbers.txt
504.621.8927
810.292.9388
856.636.8749

$ sed -E 's/-/./' phone-numbers.txt
504.621-8927
810.292-9388
856.636-8749
```

Example Substitutions

- You can use regular expression anchors like
  - `^ start of line`
  - `$ end of line`

- Add prefixes to lines:

```
$ sed -E 's/^/Number: /' phone-numbers.txt
Number: 504-621-8927
Number: 810-292-9388
Number: 856-636-8749
```

- Add suffixes to lines:

```
$ sed -E 's/8$/8 <-- Match/' phone-numbers.txt
504-621-8927
810-292-9388 <-- Match
856-636-8749
```

Advanced Substitutions

- How could you add parenthesis around the first set of area code digits?
  - Or, less valuably but just as challenging, reverse the order of each group of numbers
- Regular expressions give you the ability to match *patterns like*:
  - [0-9]{3} - Any set of 3 characters in the range of 0-9
- What if you could ***use the matched text in the replaced text***?
- With ***capturing groups*** you can!

Capturing Groups in Regular Expression Search/Replace

- Parenthesis serve a dual-purpose in many regular expression libraries:
  1. They allow complete control over order of operations
  2. They denote a ***capturing group***

- A **capturing group** holds onto matched text for you to use in substitution.

- Each capturing group is indexed starting from 1
  - The 0 capturing group refers to the entire matched text

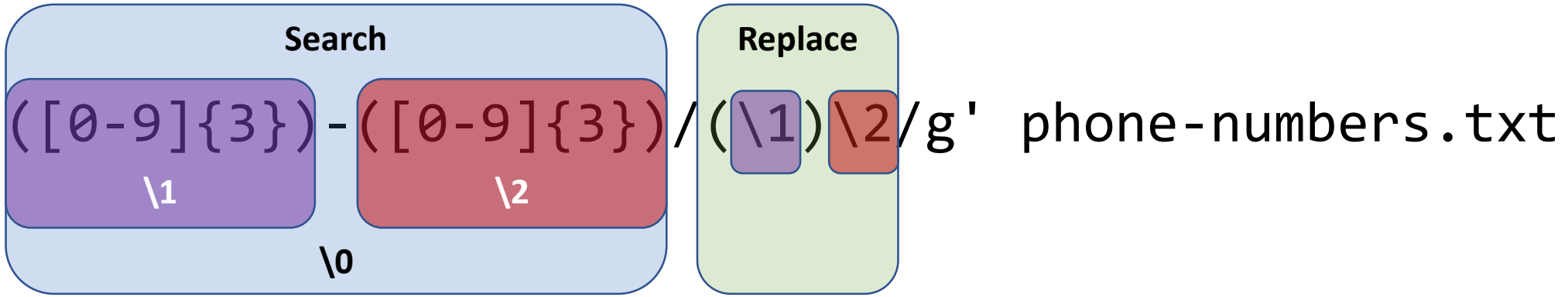
- How capturing groups are referred to in replacement is language specific
  - In sed/vim - a backslash prefixing the group index, for example: \1
  - In JavaScript/Java - a dollar symbol prefixing the group index, for example: $1

Example Substitutions with Capturing Groups

- Surround the area code in parenthesis:

```
sed -E 's/([0-9]{3})-([0-9]{3})/(\1)\2/g' phone-numbers.txt
```

- Like many little languages, these patterns look cryptic. Visual groupings help:

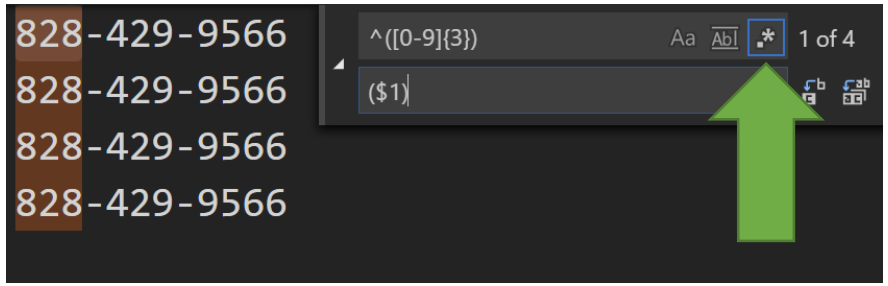


sed - Reference

- Great reference is available online
  - http://www.grymoire.com/Unix/Sed.html#TOC

For Developers,
Regular Expressions Substitution is Everywhere

- It's in VSCode's Search / Replace
  - And vim's, Eclipse's, and *every* programmer's text editor



- It's standard in higher-level programming languages
  - JavaScript, Java, Python, etc.

```
> "828-429-9566".replace(/^([0-9]{3})-/, "($1)")
"(828)429-9566"
```

- It's available in popular libraries in lower-level languages

Regular Expressions are a Power Tool

"Some people, when confronted with a problem, think
'I know, I'll use regular expressions.'
Now they have two problems."
-Jamie Zawinski

Once you're comfortable summoning the powers of regular expressions you'll want to use them *everywhere* you're processing text.

For very simple substitutions (one string literal for another) and problems (basic string manipulation) don't overthink it. Just use the basics.

For complex problems, where your input text has a grammatical structure (HTML/JSON), don't underthink it. Find a library to properly parse the structure or (rarely) write a parser.

Let's combine some utilities through the magic of pipes

```
$ curl https://cs.unc.edu/people-page/faculty/ \
    | pandoc –f html –t markdown \
    | egrep '###' \
    | sed –E 's/.*\[(.*)\].*/\1/g'
```