Lecture 11A: sed, a programmatic way to find and replace

Introduction

sed stands for stream editor

Think of streams as another way to talk about the STDIN, STDOUT that we've covered in depth. Essentially, `sed` will read the contents of a file, perform a modification on the input, and send the modified text to Standard Output by default. *Keep in mind* that by default, we are **not** changing the file's contents.

Both `sed` and `awk` (which we are covering next lecture) are complicated tools. Books are written about how to use them. From that book's introduction:

> Initially, using sed and awk will seem like the long way to accomplish a task. After several attempts you may conclude that the task would have been easier to do manually. Be patient. You not only have to learn how to use sed and awk but you also need to learn to recognize situations where using them pays off. As you become more proficient, you will solve problems more quickly and solve a broader range of problems.

Keep in mind that we can only scrape the surface of what `sed` can actually do.

Another anecdote: At one of my placements, we needed to create an OTA (over the air) update script. My supervisor Sent over a sample script that would create a hash of *itself* while excluding the line that contained any previous hash, and then compare that result with that previous hash. Essentially the script was checking to see if it had been modified since its last execution. This function was contained in one line, using `sed` . "Oh, just ignore that `sed` command, it works but don't ask how", he told me. So there you go, if you are able to invest the time into these tools, it gives you the power to perform black magic..!

Printing

All `sed` commands use the syntax: `sed '<address> <instructions>'` followed by an optional argument, which is a filename. Recall the `cars` file:

```
cat -n cars
```

```
    1  plym    fury    77      73      2500
   2  chevy   nova    79      60      3000
   3  ford    mustang 65      45      17000
   4  volvo   gl      78      102     9850
   5  ford    ltd     83      15      10500
```

```
     6  Chevy   nova    80      50      3500
     7  fiat    600     65      115     450
     8  honda   accord  81      30      6000
     9  ford    thundbd 84      10      17000
    10  toyota  tercel  82      180     750
    11  chevy   impala  65      85      1550
    12  ford    bronco  83      25      9525
```

Now let's introduce our first command:

```
sed '3,6 p' cars
```

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
ford    mustang 65      45      17000
ford    mustang 65      45      17000
volvo   gl      78      102     9850
volvo   gl      78      102     9850
ford    ltd     83      15      10500
ford    ltd     83      15      10500
Chevy   nova    80      50      3500
Chevy   nova    80      50      3500
fiat    600     65      115     450
honda   accord  81      30      6000
ford    thundbd 84      10      17000
toyota  tercel  82      180     750
chevy   impala  65      85      1550
ford    bronco  83      25      9525
```

Notice which lines get repeated. `sed` by default will print all lines, so here you see the combination of the entire `cars` file with the output from the `3,6 p` instruction.

`-n` will turn off the printing of all lines.

```
sed -n '3,6 p' cars
```

```
 ford    mustang 65      45      17000
volvo   gl      78      102     9850
ford    ltd     83      15      10500
Chevy   nova    80      50      3500
```

Deleting

Again, please note that although we are 'deleting' line 5, we are only doing this to the output on our display.

```
sed '5 d' cars
```

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
ford    mustang 65      45      17000
volvo   gl      78      102     9850
Chevy   nova    80      50      3500
fiat    600     65      115     450
honda   accord  81      30      6000
ford    thundbd 84      10      17000
toyota  tercel  82      180     750
```

```
chevy   impala  65      85      1550
ford    bronco  83      25      9525
```

```
sed '5,8 d' cars
```

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
ford    mustang 65      45      17000
volvo   gl      78      102     9850
ford    thundbd 84      10      17000
toyota  tercel  82      180     750
chevy   impala  65      85      1550
ford    bronco  83      25      9525
```

The first example goes like this: "*for each line, print but delete line 5.*" The second does the same but for a range.

Quitting

```
sed '5 q' cars
```

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
ford    mustang 65      45      17000
volvo   gl      78      102     9850
ford    ltd     83      15      10500
```

Once we reach line 5, we 'quit' `sed` and nothing more is printed to our screen. In other words, this does the same thing as `head -5 cars` .

Regular Expressions

You knew weren't done with these, didn't you?

```
sed -n '/chevy/ p' cars
```

```
 chevy   nova    79      60      3000
chevy   impala  65      85      1550
```

This seems to do the same thing as `grep` . Here's an example where we delete the pattern:

```
sed '/chevy/ d' cars
```

```
 plym    fury    77      73      2500
ford    mustang 65      45      17000
volvo   gl      78      102     9850
ford    ltd     83      15      10500
Chevy   nova    80      50      3500
fiat    600     65      115     450
honda   accord  81      30      6000
ford    thundbd 84      10      17000
toyota  tercel  82      180     750
ford    bronco  83      25      9525
```

Notice that this didn't ignore case. You could modify it with `/chevy/I`.

Now consider this example. What does this do?

```
sed '/chevy/ q' cars
```

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
```

Substituting

The real power of `sed` is in being able to search and replace strings. How does this look?

```
sed 's/[0-9]/*/' cars
```

```
 plym    fury    *7      73      2500
chevy   nova    *9      60      3000
ford    mustang *5      45      17000
volvo   gl      *8      102     9850
ford    ltd     *3      15      10500
Chevy   nova    *0      50      3500
fiat    *00     65      115     450
honda   accord  *1      30      6000
ford    thundbd *4      10      17000
toyota  tercel  *2      180     750
chevy   impala  *5      85      1550
ford    bronco  *3      25      9525
```

Only the first number in each line is being replaced by an asterisk. Using `g` for a 'global' replace will replace *all* of the numbers:

```
sed 's/[0-9]/*/g' cars
```

```
 plym    fury    **      **      ****
chevy   nova    **      **      ****
ford    mustang **      **      *****
volvo   gl      **      ***     ****
ford    ltd     **      **      *****
Chevy   nova    **      **      ****
fiat    ***     **      ***     ***
honda   accord  **      **      ****
ford    thundbd **      **      *****
toyota  tercel  **      ***     ***
chevy   impala  **      **      ****
ford    bronco  **      **      ****
```

Use `cat` to check that the `cars` is still intact. Remember: by default this doesn't affect the original file. If we wanted to, we could redirect this out to a new file, but also using `-i` will allow us to 'save' our changes.

This is essentially doing the same thing that you can do in any text editor, and using syntax that is much more complicated. Why would we ever use this?

Answer: Imagine now that instead of working with just one file named 'cars', you are working with *hundreds* of files called 'cars*'. Even the most efficient office worker will

need to open each file in notepad.exe and run their 'ctrl f'. sed has the ability to save thousands of hours, as long as you understand the syntax. *Always do a dry run to check for errors before replacing important data. "With great power comes great responsibility."* We can teach you the power, but maybe not the responsibility..!

Summary

- `sed 'address instruction' filename`
- checks for address match, one line at a time, and performs instruction if address matched
- prints lines to standard output by default (supressed by -n option)

Addresses

- can use a line number, to select a specific line (for example: 5)
- can specify a range of line numbers (for example: 5,7)
- can specify a regular expression using //
- default address (if none is specified) will match every line

Instructions

- *p* - print line(s) that match the address (usually used with -n option)
- *d* - delete line(s) that match the address
- *q* - quit processing at the first line that matches the address
- *s* - substitute text to replace a matched regular expressions, similar to vi substitution

---

Substitutions in Vim

Now that we've talked about substutions using `sed`, we can use this to perform replacements in Vim as well. Replacements are done in **Command Mode** so press **:** to enter that mode. Once in command mode, the syntax looks like this:

```
s/old/new/g
```

This will replace `old` with `new`, but only on the current line. There's two small differences to keep in mind. Use **%** to change *all* instances of `old`, and add **c** to ask for *confirmation*.

```
%s/old/new/gc
```

There is one more thing to touch on here. Consider the following:

```
<blink>Let's get rid of the ugly blink tags. What is this, Geocities?!</blink>
```

We know that we want to get rid of `<blink>` and `</blink>` but keep everything in between. Consider this:

```
%s/<blink>.*</blink>//gc
```

If you try that, it's going to fail. We need to escape the **/** inside the closing blink tag.

```
%s/<blink>.*<\/blink>//gc
```

That's going to successfully delete our entire line, including the part we want to keep. :

The way we preserve something is using *backreferences*. Create a backreference with quotes **( )**.

```
%s/<blink>(.*)<\/blink>//gc
```

Anything inside the backreference can be 'pasted' into the new field with **₩1**. So now we have this:

```
%s/<blink>(.*)<\/blink>/\1/gc
```

If you try this, it won't work... again. What's wrong? We have to escape both **(** and **)**. So finally:

```
%s/<blink>\(.*\)<\/blink>/\1/gc
```

And there we go.