sed

sed is a standard Unix utility which is a "stream editor", allowing transformations of its input. It reads the stream input line by line, performs command/script actions (if allowed on that line), and outputs the *modified* line.

Typical uses of sed:

- Execute repetitive edits to one or more files.
- Convert data
- Select lines containing certain data

The actions to be performed can be in either the command line directly or can be a script inside a *scriptFile*:

```
sed options 'command' file1 ...
sed options -f scriptFile file1 ...
```

sed processing steps for each input line:

1. Place the line in a buffer (pattern space)
2. Execute the command or commands on the buffer
3. Output the buffer to stdout

Create a SedExamples folder. When logged into a fox server, please cd to the **/usr/local/courses/ssilvestro/cs3423/sed** directory and copy all the files to your sed directory.

cs1713p0.c
cs1713p0v2.c
file1
file3
file4
file5
file6
Linux
fileSSN
inventory.txt

Three types of addressing modes in sed:

0 – no address → operates on all lines of input
1 – one address → operates on all matching lines
2 – address range → operates on all lines within the range

Example 1: Only output to stdout all lines in cs1713p0.c that contain "printf"

```
$ sed -n '/printf/p' cs1713p0.c
 // about the safety of scanf and printf
    printf("%-10s %-20s %10s %10s %10s %10s\n"
            printf("invalid input when reading student data, only %d valid
values. \n"
            printf("\tdata is %s\n", szInputBuffer);
         printf("%-10s %-20s %10.2f %10.2f %10.2f %10.2f\n"
```

- '/printf/p' means to print lines containing "printf"
- The -n means to **not** output lines normally. When used with the "p", only the results of the "p" are returned.

Example 2: Create a new file2 from file1 replacing "Linux" with "Unix".

```
$ sed -i.bak 's/Unix/Linux/g' *.tex
```

- **s** means **substitute**. It is followed by a match pattern and a replacement value.
- **g** means **globally** change each occurrence on the line (this doesn't mean globally in the file)
- every line is sent to file2 including both modified and unmodified lines.

Example 3: Modify file3 removing carriage returns. For safety, make a backup of file3 in file3.sav.

```
$ sed -i.sav 's/\r/\n/g' file3
```

- `-i` means to edit the file "in place" which means to modify the file. It creates a backup copy of "file3" using the specified suffix ".sav". This option is very useful if you need to modify multiple files.
- replaces each carriage return with an empty string (i.e., deletes the carriage return)

Example 4: Modify file4 and file5 replacing only one occurrence of "cat" with "dog" per line for lines 1 thru 3. Save the old files with a ".cat" suffix

```
$ sed -i.cat ‘1,3 s/cat/dog/' file4 file5
```

- 1,3 is the range of lines to apply the substitution
- The backup copies are named file4.cat and file5.cat

Example 5: Delete lines in file6 that begin with "#", producing file6.new

```
$ sed '/^#/d' file6 > file6.new
```

- The `'/^#/d'` is a pattern for deleting lines.
- "^" specifies that the pattern must be at the beginning of the line.

	• The "d" means to delete. **Note:** the characters after the ending slash/delimiter are called **flags**.
Using a Script File As stated above, the **-f** switch is used to specify a script file which allows for more complex capabilities. Notice that example #1 printed lines which had "printf" in "//" comments. We can remove those lines. Our script will: • delete lines that begin with any number of spaces (including zero) followed by "//". Any number of spaces is specified by `" *"`. • print lines that contain "printf"	**Example 6:** Only output to stdout the lines in cs1713p0.c that contain "printf", but don't include lines that begin (not necessarily in column 1) with "//". # Create a script file named "example6" `$ cat >example6` `/^ *\/\//d` `/printf/p` CTRL-D • The first action gave a pattern that begins at the beginning of the line, followed by any number of spaces, and then followed by two slashes which had to be escaped using backslashes since slashes are our delimiters for patterns. • If the pattern in the first action is matched, the line is deleted. • The second action is only executed after the first action. If the line is deleted, the second action is not executed. • The second action prints lines containing "printf" `$ sed -n -f example6 cs1713p0.c` `printf("%-10s %-20s %10s %10s %10s %10s\n"` `printf("invalid input when reading student data, only %d valid values. \n"` `printf("\tdata is %s\n", szInputBuffer);` `printf("%-10s %-20s %10.2f %10.2f %10.2f %10.2f\n"` `# Also try it on cs1713p0v2.c` `$ sed -n -f example6 cs1713p0v2.c` `// about the safety of scanf and printf` `printf("%-10s %-20s %10s %10s %10s %10s\n"` `printf("invalid input when reading student data, only %d valid values. \n"` `printf("\tdata is %s\n", szInputBuffer);` `printf("%-10s %-20s %10.2f %10.2f %10.2f %10.2f\n"` What problems do you see with that output? ??.
Example 6 only handled leading spaces. What if the "//" was preceded by tabs and/or spaces? We need a pattern of any number of spaces or tabs in any order. This is done by `"[ \t]*"` . Note that in non-GNU versions of sed, you may have to enter a tab character instead of "\t".	**Example 7:** Only output to stdout the lines in cs1713p0.c that contain "printf", but don't include lines that begin (not necessarily in column 1) with "//". The "//" could be preceded by a combination of spaces and tabs. # Create a script file named "example7" `$ cat >example7` `/^[ \t]*\/\//d` `/printf/p` CTRL-D • The first action gave a pattern that begins at the beginning of the line, followed by any number of spaces or tabs, and then followed by two

	slashes which had to be escaped using backslashes since slashes are our delimiters for patterns. • If the pattern in the first action is matched, the line is deleted. • The second action is only executed after the first action. If the line is deleted, the second action is not executed. • The second action prints lines containing "printf" `$ sed -n -f example7 cs1713p0v2.c` `printf("%-10s %-20s %10s %10s %10s %10s\n"` `printf("invalid input when reading student data, only %d valid values. \n"` `printf("\tdata is %s\n", szInputBuffer);` `printf("%-10s %-20s %10.2f %10.2f %10.2f %10.2f\n"`
Example 6 and example 7 didn't print the entire **printf** statements; instead, those examples just printed the line that contained "printf". How can we print the entire printf statement? We need a range of lines that begin with a "printf(" and end with its ");"	**Example 8:** Only output to stdout the lines in cs1713p0.c that contain actual "printf" statements including all lines until the ");". # Create a script file named "example8" `$ cat >example8` `/^[ \t]*\/\//d` `/printf(/,/);/p` • The second action lists a range of lines that begin with a line containing "printf(" and ends with a line containing ");". Those are two different patterns which are in a range since there is a comma between them. `$ sed -n -f example8 cs1713p0.c` `printf("%-10s %-20s %10s %10s %10s %10s\n"` `, "ID", "Name", "Exam 1", "Exam 2", "Final", "Average");` `printf("invalid input when reading student data, only %d valid values. \n"` `, iScanfCnt);` `printf("\tdata is %s\n", szInputBuffer);` `return ERR_BAD_INPUT;` `}` `dAverage = (student.dExam1 + student.dExam2 + student.dFinalExam) / 3;` `printf("%-10s %-20s %10.2f %10.2f %10.2f %10.2f\n"` `, student.szStudentIdNr` `, student.szStudentFullNm` `, student.dExam1` `, student.dExam2` `, student.dFinalExam` `, dAverage);` Notice that the result is incorrect. Why? ?? Answer: Two-address mode cannot start and stop on the same line; the second address is not checked until at least the next line has been read, and thus cannot apply to a single line.
We can fix the problem with example 8 as shown to the right.	**Example 9:** Only output to stdout the lines in cs1713p0.c that contain actual "printf" statements including all lines until the ");". This fixes the problem in #8 # Create a script file named "example9" `$ cat >example9` `/^[ \t]*\/\//d`

```
/printf(.*);/p
/printf(.*);/d
/printf(/,/);/p
```

- The second action matches a statement that begins with "printf(" followed by any characters and then has a ");" all on the same line. It then prints it.
- The third action does the same pattern match as the second action, but it deletes the line; therefore, the fourth action is only executed when it doesn't match.

```
$ sed -n -f example9 cs1713p0.c
    printf("%-10s %-20s %10s %10s %10s %10s\n"
        , "ID", "Name", "Exam 1", "Exam 2", "Final", "Average");
            printf("invalid input when reading student data, only %d valid values. \n"
                , iScanfCnt);
            printf("\tdata is %s\n", szInputBuffer);
        printf("%-10s %-20s %10.2f %10.2f %10.2f %10.2f\n"
            , student.szStudentIdNr
            , student.szStudentFullNm
            , student.dExam1
            , student.dExam2
            , student.dFinalExam
            , dAverage);
```

Some important pattern symbols for sed **regular expressions**:

Basic regular expressions (requires "-r" in sed or "-E" in grep to use natively):

.	matches any character including newline
^	the pattern that follows must begin at the first character
$	the pattern that precedes the $ must match at the end of the line
*	matches zero or more of the preceding character, group or bracketed list
[*list*]	matches one character to any of the characters **listed** within the brackets. For convenience, range abbreviations can be used (e.g., [a-f], [0-9])
[^*list*]	matches one character if it is **not listed** within the brackets.

Extended regular expressions (requires "-r" in sed or "-E" in grep to use natively, without needing escape characters):

\(*regexp*\)	groups the inner *regexp* and allows it to be referenced later with \1, \2, etc
\+	similar to *, but matches one or more
\{*i*\}	matches *i* occurrences of the preceding character, group or bracketed list
\{*i*,\}	matches *i* or more occurrences of the preceding character, group or bracketed list
\{*i*,*j*\}	matches between *i* and *j* occurrences of the preceding character, group or bracketed list
\?	matches zero or one instance of the preceding;

Example patterns for sed regular expressions:

/^The/	Matches any line that begins with the word "The"
/[a-z]\{3\}[0-9]\{3\}/	Matches a lowercase abc123 ID.
/[0-9]*$/	Matches any line that ends with zero or more digits. Why is that pattern not very useful? What would make it have to end in digits? /[0-9]\+$/
/\(bark\)\+/	Matches one or more occurrences of "bark"
/(.*)/	Matches a left parenthesis followed by any number of any characters followed by a right parenthesis.

What does this pattern mean?

/[^abcd]\.$/ ??

What would be the pattern for matching lines that are empty or contain any number of spaces and/or tabs?

??

Sam's note: demonstrate different between these
Use --color flag to illustrate matches / nature of match.

```
\grep "[^abcd].$" ~/lines
\grep "[^abcd]\.$" ~/lines ← is now a literal, not a quantifier!

\grep "[a-z]?$" ../lines
```

	equivalent to `\{0,1\}`
pat1`|`*pat2*	alternation: matches either *pat1* or *pat2*

Symbols representing characters to match (e.g., ., [list], etc) are called **classes.** Symbols which dictate the amount of a preceding class (e.g., *, +, etc) are called **quantifiers**.

Groups can be referenced later with "\1", "\2", etc.

To reference the characters ".", "^". "$", "*", "[", "]", in a regular expression, it must be escaped with a backslash.

To reference the characters "+", "{", "}", "(", ")", just use each normally. Their special (extended) meaning requires the use of a backslash when referenced in a basic regular expression.

```
\grep "[a-z]\?$" ../lines   ← is now a quantifier, not
                                    a literal!
```

Note that sed also supports **extended regular expressions** if the **-E** or **-r** switch is specified (**BSD vs GNU**). This changes some of the above symbols and adds some additional symbols.

.	matches any character including newline
^	the pattern that follows must begin at the first character
$	the pattern that precedes the $ must match at the end of the line
*	matches zero or more of the preceding character, group or bracketed list
[*list*]	matches one character to any of the characters **listed** within the brackets. For convenience, range abbreviations can be used (e.g., [a-f], [0-9])
[^*list*]	matches one character if it is **not listed** within the brackets.
`(`*regexp*`)`	groups the inner *regexp* and allows it to be referenced later with \1, \2, etc
`+`	similar to *, but matches one or more
`{`*i*`}`	matches *i* occurrences of the preceding character, group or bracketed list (see other variants above (e.g. {i,}, {1,j})
pat1`|`*pat2*	matches either *pat1* or *pat2*
`?`	matches zero or one instance of the preceding; equivalent to `{0,1}`

Example patterns for sed extended regular expressions:

`/[a-z]{3}[0-9]{3}/`	Matches a lowercase abc123 ID.
`/(bark)+/`	Matches one or more occurrences of "bark"
`/\(.*\)/`	Matches a left parenthesis followed by any number of any characters followed by a right parenthesis.
`/(abc)|(xyz)/`	`Matches exactly "abc" or "xyz".`

Sam's note: `Look at ~/sed/samexp`

Using sed extended regular expressions, what would be the pattern to match a phone any phone number like (210)456-1234, but with any digits?

`Purely basic regular expressions:`
([0-9][0-9][0-9])[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]

Extended regular expressions, but in *basic mode*:
([0-9]\{3\})[0-9]\{3\}-[0-9]\{4\}

Purely extended regular expressions:
\([0-9]{3}\)[0-9]{3}-[0-9]{4}

To reference the characters ".", "^". "$", "*", "[", "]", "+", "{", "}", "(", ")", "|" in an extended regular expression, it must be escaped with a backslash.	**Run: ~/sed/phonepats.bash phonenum**
Exercise #1: For privacy reasons, we want to replace Social Security Number references in emails from customers with XXX-XX-XXXX when we archive the email. Assume the input file is fileSSN, produce an output file with SSNs replaced as described above.	Sed in extended mode: `sed -r -e 's/[0-9]{3}-[0-9]{2}-[0-9]{4}/XXX-XX-XXXX/g' ~/sed/ssnnum` Sed in regular mode, but using extended regex: `sed -e 's/[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}/XXX-XX-XXXX/g' ~/sed/ssnnum` Sed in regular mode, using purely basic regex: `sed -e 's/[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]/XXX-XX-XXXX/g' ~/sed/ssnnum`
Negation We can negate an address (including a regex pattern) by following the pattern with a bang.	**Example 10**: Delete all lines in the file that **don't** contain "linux" or "Linux" `$ sed '/[Ll]inux/! d' Linux` `Although Linux is well known today, Linux was not known` `was a high school English teacher, if she knew Linux.`
Append and Insert **a** Append inserts one or more lines of text **after** the matched lines. **i** Inserts one or more lines of text **before** the matched lines.	**Example 11:** We will append the Line "hello there" to each of the first three lines of the data from file5. # create a script file named "app11" `$ cat > app1` `1,3a hello there` `CTRL-D` `$ sed -f app11 file5` `The weather was getting very bad.  It was raining cats and dogs.` `hello there` `It was so bad that I stepped in a poodle.` `hello there` `what a catastrophe.` `hello there` `I was so tired that I needed a cat nap.` **Example 12:** Insert the two lines shown below before any lines containing "cat" or "dog" in file5. # create a script file name "ins12" `$ cat > ins12` `/(cat)|(dog)/i\` `Who is watching\` `a cat or dog video?` `CTRL-D` `$ sed -E -f ins12 file5` `Who is watching` `a cat or dog video?` `The weather was getting very bad.  It was raining cats and dogs.` `It was so bad that I stepped in a poodle.` `Who is watching`

	a cat or dog video? what a catastrophe. Who is watching a cat or dog video? I was so tired that I needed a cat nap.
Change **c** changes one or more **lines** replacing them with the specified text	**Example 13**: Replace lines 2-6 with "eunuchs" for the linux file $ sed '2,6c eunuchs' Linux Although Linux is well known today, Linux was not known eunuchs a word that was a homonym, she was unnerved. **Example 14**: Completely replace lines containing "Linux" or "Unix" with simply "eunuchs" $ sed -E '/(Linux)|(Unix)/c eunuchs' Linux eunuchs outside of technical circles in the 1980s. I worked for a company that was hiring technical writers in the mid 1980s. An interviewer asked a candidate, who eunuchs Having never heard of the operating system, but knowing a word that was a homonym, she was unnerved.
Substitute We have already seen several examples for substitute (examples 2, 3, and 4 and exercise #1). s/*matchPattern*/*replaceValue*/*flags* • *matchPattern* is the pattern to find in a line • *replaceValue* is the value used as a replacement for the matched value • *flags:* g - globally replacement all occurrences in the line p - print the contents of the pattern space if successful w - write pattern space to a file if successful *i* - this is a number; replace the *i*th occurrence only Although our examples have shown slash ("/") as the delimiter between the *matchPattern, replaceValue, and flags,* you can specify a different delimiter after the "s". This can make it easier when you need slash in your pattern or replacement value. Substitute simply uses the first character after the "s" as the delimiter Instead of s/\/\/.*$// you could use s\|//.*$||	**Example 15**: Examine the file inventory.txt. It contains many inventory records. The values are separated by spaces except the last data value is a product name which can contain multiple spaces and follows the phrase "Name:". We want to keep the product ID and unit price. This means that we want to remove the second, third, and fifth logical columns. Note that product IDs end in three numbers. $ cat >example15 ?? CTRL-D $ sed -f example15 inventory.txt PPF001 9.95 SBB001 14.95 SBG002 14.95 BOM001 29.95 MCW001 12.45 TTP001 9.95 NHC001 9.95 SSX001 29.95 **Sam's note:** my answer in ~/sed/example15.bash sed -r -e 's/ [0-9]+ [0-9]+ ([0-9.]+) .*$/ \1/' inventory.txt
Alternative to Scripts when needing multiple commands	**Example 15-2:** use -e command arguments to solve example 15. $ sed '-e ??' '-e ??' \

The **-e** (lowercase) command argument can be used to provide multiple edits to a file without requiring a script file.

This is very useful when you need variable values for different sed script steps since sed doesn't provide an easy way to pass variables as parameters.

```
? '-e ??' inventory.txt
Answer: sed -r -e 's/ [0-9]+//' -e 's/ [0-9]+//' -e 's/ ([0-9.]+)
.*$/ \1/' inventory.txt
PPF001 9.95
SBB001 14.95
SBG002 14.95
BOM001 29.95
MCW001 12.45
TTP001 9.95
NHC001 9.95
SSX001 29.95
```

Next

The next command can be used to skip lines without printing them.

- n – Skips to the next line of input and overwrites the current line. Therefore, the current line and all changes made to it will not be printed.
- N – Reads the next line of input and appends it (with a newline character preceding) to the end of the current line.

sed uses newline characters (\n) to separate one line from another and when a line is copied into the pattern space the newline character is removed. This means that using the commands we've seen so far, we would be unable to remove newline characters from a file. One solution to this is the N command.

Note that when printing a line of output sed prints a newline character provided there is at least one character in the buffer. Because of these restrictions sed is not the recommended tool for removing newline characters. Instead use 'tr'

Example 16: Remove newline characters from any line containing 'Unix' and replace them with a tab character

```
$ cat >example16
/(Linux)|(Unix)/ s/\n/\t/
CTRL-D
$ sed -E -f example16 < file1

Although Unix is well known today, Unix was not known
outside of technical circles in the 1980s.  I worked
for a company that was hiring technical writers in
the mid 1980s.  An interviewer asked a candidate, who
was a high school English teacher, if she knew Unix.
Having never heard of the operating system, but knowing
a word that was a homonym, she was unnerved.
```

The above script fails to find and replace the newline characters because they have already been removed by sed before pattern matching happens.

```
$ cat >example16
/(Linux)|(Unix)/N
s/\n/\t/
CTRL-D

$ sed -E -f example16 < file1
Although Unix is well known today, Unix was not known   outside of technical circles in the 1980s.  I worked
for a company that was hiring technical writers in
the mid 1980s.  An interviewer asked a candidate, who
was a high school English teacher, if she knew Unix.    Having never heard of the operating system, but knowing
a word that was a homonym, she was unnerved.
```

Input Buffers

The sed editor provides 2 buffers to use while editing a file. These are referred to as the **pattern space** and the **hold space.**

The **pattern space** and **hold space** buffers have several commands that can be used to manipulate them.

All commands we have used so far operate on the pattern space.

Immediately after finishing all commands in the file and before looping back, sed outputs the contents of the pattern space followed by a newline unless the -n flag is set. **This is why all lines print by default.**

The hold space is an additional buffer that can be used to hold a value across multiple lines. It remains unchanged until operated on by one of the commands on the right.

- **g** – Copies contents of hold space into pattern space
- **G** – Appends contents of hold space onto pattern space (Adding a newline between)
- **h** – Copies contents of pattern space into hold space
- **H** – Appends contents of pattern space into hold space (Adding a newline between)
- **x** – Swaps values stored in pattern space and hold space

Move a line

Example 16 first uses the **h** command to store the contents of line containing the pattern 'last'. Next the **d** command is used to remove it from the pattern space so that it is not printed. Finally, on the last line of input the **G** command is used to append the line stored in the hold space onto the end of the pattern space before printing.

What would happen if there were multiple lines in the file containing the pattern 'last'?

??

Warning: some implementations of sed restrict the max size stored in both the pattern space and hold space to 4000 bytes. GNU sed has no max buffer size provided it can malloc more memory.

Example 17: Examine the file hold_example.txt. It has a problem; the last line is in the wrong positon and needs to be moved to the end. This can only be done if we are somehow able to save off the first line and only print it when we reach the last line.

```
$ cat >example17
/last/h
/last/d
$G
CTRL-D
$ sed -f example17 hold_example.txt
First line
Second line here
Here is the third line
Almost at the final line.
This should be the last line.
```