# The Stream Editor *sed*

One of the more useful Unix commands is the *sed* editor, which allows you to systematically alter a file in many useful ways; but it is also somewhat obscure to use The *man* page for it is hopelessly complex, but there is a very useful book devoted to it.[1]

These notes are only a brief introduction. Full use of `sed` requires that you gain some skill with what Unix calls *regular expressions*; I discuss these separately.

## Overview

*Sed* basic function is to read a line from standard input[2] For each line read in, `sed` first looks to see if the line matches some criterion; if so, `sed` performs some action on the line, and sends the result to standard output. Then `sed` reads the next line, and does the same thing, and so on until the end of the file.

The default action for *sed* is to print every line, so typing `cat file1 | sed` is the same as typing `cat file1`. However, if we use a `-n` flag, this default printing is supressed; typing `cat file1 | sed -n` will produce no output. Here is an example, for a short file we will process with `sed` in various ways:

```
%cat file1
The quikc brown fox
jumped over the lazy dog.
The dog ignured the gnu.
%cat file1 | sed -n
%
```

[1] Dale Dougherty and Arnold Robbins (1997) *sed & awk* (O'Reilly Media)

[2] Remember that this can be the output of another program piped in via `|`, or the contents of a file, sent using `cat ... |`.

## Printing Part of a File

*Sed* can choose lines by number, so we can use *sed* to print out particular lines by giving a line number, or range of numbers, and specifying that the action is `p` (print). The lines in a file are considered to be numbered from 1 onwards; the last line is referred to by a `$`. For example, to print the first line we would have:

```
%cat file1 | sed -n 1,1p
The quikc brown fox
%
```

and to print the last two lines

```
%cat file1 | sed -n '2,$p'
jumped over the lazy dog.
The dog ignured the gnu.
%
```

Note that if we did not use the `-n` flag, we would get

```
%cat file1 | sed '2,$p'
The quikc brown fox
jumped over the lazy dog.
jumped over the lazy dog.
The dog ignured the gnu.
The dog ignured the gnu.
%
```

because *sed* prints every line (the default) and does so again for the lines we asked for. This mode of using `sed` is a rapid way to extract particular parts of a file – though if you just want to get some number of lines from the start or end, you should use *head* or *tail*.

*Sed* will also match against strings of characters, as for example

```
%cat file1 | sed -n '/dog/p'
jumped over the lazy dog.
The dog ignured the gnu.
%
```

You will have noticed that in the later examples, the expression telling *sed* what to do is surrounded by single quotes (apostrophes), `'`. The reason for this is that the dollar sign `$` is also interpreted (by the shell), unless "protected" by the quote marks. If we did not use them we would get

```
%cat file1 | sed -n 2,$p
tcsh: p: Undefined variable.
%
```

because the shell thinks that `$p` means "the variable *p*". It is prudent to always enclose `sed` commands in quote marks, even if you might not need to, just to avoid having to remember what the shell will and will not let you use.

## Editing Lines

We have seen one action: `p`, which causes printing of a line. We now conside the "substitute" or `s` action, whicb has to be follwed by a pair of character strings in delimiters; this causes the second string to replace the first. An example will make this clearer:

```
%cat file1 | sed 's/kc/ck/'
The quick brown fox
jumped over the lazy dog.
The dog ignured the gnu.
%
```

which, by changing `kc` to `ck` fixes a spelling error. Each line is printed (since there is no `-n` flag) only after it has been acted on, in this case by a substitution. Another example to fix a different error is:

```
%cat file1 | sed 's/nu/no/'
The quikc brown fox
jumped over the lazy dog.
The dog ignored the gnu.
%
```

In the last line, the action is done, but only once (changing `ignured` to `ignored`), which is the default. To have the substitution done for every occurrence, we would add a `g` (for "global") and get:

```
%cat file1 | sed 's/nu/no/g'
The quikc brown fox
jumped over the lazy dog.
The dog ignored the gno.
%
```

which in this case is not what we wanted.

Another *sed* action is `d`, which means to delete (not print) the line. So, for example, we could combine this with matching strings to get

```
%cat file1 | sed '/dog/d'
The quikc brown fox
%
```

which has deleted all the lines with `dog`.

We can also match to lines that do not contain a string, by preceding the action part with a `!`; for example[3]

```
%cat file1 | sed '/dog/!d'
jumped over the lazy dog.
The dog ignured the gnu.
%
```

which has kept all the lines with `dog`; though as we saw above, there is a more direct way of doing this.

Our sample file has two spelling errors; to fix both, we write

```
%cat file1 | sed -e 's^kc^ck^' -e 's/nu/no/'
The quick brown fox
jumped over the lazy dog.
The dog ignored the gnu.
%
```

which illustrates two usages. First, we can use other delimiters than `/` to divide the strings; whatever character follows the `s` is taken to be the delimiter. Second, we can use the `-e` flag to mean "the next stuff is an instruction"; if this flag is not used, *sed* can take only one command on a line.[4]

All versions of *sed* allow the edits to be put in a file; for example, if we had a file `tmps` containing

```
s^kc^ck^
s/nu/no/
```

then we could make the edits using:

[3] A warning: if you are running the C shell or something derived from it this will fail because the quotes do not protext the `!`.

[4] Some versions of *sed* would allow the example given above to be compressed to `sed 's/kc/ck/;s/nu/no/'` However, this is not a good idea; *sed* expressions can easily become obscure, and compacting them in this way guarantees their incomprehensibility.

```
%cat file1 | sed -f tmps
The quick brown fox
jumped over the lazy dog.
The dog ignored the gnu.
%
```

which is simpler whenever you have many edits to consider. You might embed this all in a script as

```
%!/bin/bash
cat << XXX > tmps
s^kc^ck^
s/nu/no/
XXX
cat file1 | sed -f tmps > file2
rm tmps
```

This would be the preferred way to run `sed` to clean up a file called `file1` and put the results into `file2`. To add or modify `sed` commands, you just change what is in the here document part (between the `cat` and the `XXX`) and rerun the script, which also does cleanup at the end. The commands are are performed in the order given, which can sometimes lead to unexpected modifications.

And just to remind you that *sed* does the same thing on all lines

```
%cat file1 | sed 's/dog/cat/'
The quikc brown fox
jumped over the lazy cat.
The cat ignured the gnu.
%
```

## Editing and Matching

The previous two sections showed how *sed* could match to material on a line and perform a simple action (`p`); or, alternatively, preform a more complicated action (`s`) on all lines. But we can combine matching and more complicated actions; for example:

```
%cat file1 | sed '/T/s/e/is/g'
This quikc brown fox
jumped over the lazy dog.
```

```
This dog ignurisd this gnu.
%
```

Here the `sed` command means means “On any line with a `T`” (as indicated by the `/T/`), “change any `e` to `is`” (as indicated by the `s/e/is/g`; remember that `g` makes the substitution global on each line), This instruction made one change on the first line, ignored the second, and made three changes (one nonsensical) on the third; remember that the matching and substitution are independent, so once a `T` has been found, the substitution of `is` for `e` will proceed for the whole line. All the `/`’s in the command may seem confusing; many versions of sed would allow the clearer version `sed ’/T/ s/e/is/g’`.

## Examples Using Regular Expressions

We now turn to the use of regular expressions in `sed`; before reading this you should read the notes that describe these. Regular expressions are heavily used in *sed*, and, conversely, `sed` examples are often used to illustrate regular-expression usage.

Some examples of regular expressions in either the matching or the choice of what to edit are shown in the following table;[5] you should work through these to be sure you understand them.

| | |
|---|---|
| `sed ’s/^[ TAB]*//’` | delete whitespace (spaces or tabs) from front of line |
| `sed ’s/[ TAB]*$//’` | delete whitespace (spaces or tabs) from end of line |
| `sed ’s/^/ /’` | insert 5 blank spaces at beginning of each line |
| `sed ’!/"/s/,/TAB’` | replace commas with tabs on all lines not containing double quotes (useful for reformatting .csv files from spreadsheets. Lines with double quotes are skipped because in many such files these protect embedded commas–but not against *sed* |
| `sed ’/^$/d` | remove all lines with no characters |
| `sed ’/^[ TAB]*$/d` | remove all blank lines |

In this table `TAB` refers not to these letters, but to the tab key.

One regular expression that we did not discuss before, because it is useful only in `sed`, is the character `&`, which appears in the “right-hand field” of a substitute (`s`) command. This character, used this way, means “whatever was

[5] Many of these are taken from a list produced by Eric Perment, available at `http://www.pement.org/sed/sed1line.txt`; thanks to Andy Barbour for telling me about this.

matched in the first part of the command". It may at first seem that this is a pointless expression: after all, the purpose of the substitute command is to replace what we have found. If we were to change our command `s/kc/ck/` to `s/kc/&/` it would simply replace what we found with itself.

However, consider the command

```
sed 's/Section [1-9][̇0-9]*/(&)/'
```

The left side would match, for example, `Section 1.` or `Section 1.0` or `Section 1.01` or `Section 9.62`; and all of these would be replaced by themselves, *but* surrounded by parentheses, e.g., `(Section 9.62)`. This is just the kind of change over multiple occurrences that we could do by hand, rather tediously, but that `sed` can do much faster, and with much less work on our part.

## On Developing *sed* Scripts

The following suggestions on developing sets of *sed* commands come mostly from the book by Dougherty and Robbins; I can attest to their soundness:

A. Use *grep* to examine what a particular expression will find: probably more occurrences than you expected. Make sure you understand most of the structure of the input file before starting with *sed*.

B. Develop the script a few commands at a time; if you write a lot of commands at once, and the output doesn't come out the way you expect, you'll have a harder time finding out what went wrong.[6]

C. Start with the commands that make the obvious changes, and the most frequently-occurring ones. You can then deal with the less frequenct, and more peculiar, cases later, using commands designed for them, placed later in the command list. It may be quicker to do the last few changes by hand.

D. Keep checking the output, especially for odd cases. The larger the volume of text, the more likely there will be some unusual piece of text that your command will act on in some unexpected way (like the change from "ignured" to "ignurisd").

[6] This is a basic rule for any kind of programming: it is better to make small steps, each one easily corrected, than to try for everything at once.

E. Don't be too clever. This too is a good general rule in programming: a clever, terse solution will probably take more of your own time to figure out later. `Sed` lends itself to this, with commands like

```
sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta'
```

which, complicated as it looks, just puts commas into a numeric string every three places. To break it down: