Sed - An Introduction

Last update Sun Sep 24 19:44:00 EDT 2006
Thanks to Keelan Evans for spotting some typos.
Thanks to Wim Stolker as well.

Table of Contents

Introduction to Sed

How to use sed, a special editor for modifying files automatically. If you want to write a program to make changes in a file, sed is the tool to use.

There are a few programs that are the real workhorse in the Unix toolbox. These programs are simple to use for simple applications, yet have a rich set of commands for performing complex actions. Don't let the complex potential of a program keep you from making use of the simpler aspects. This chapter, like all of the rest, start with the simple concepts and introduces the advanced topics later on.
A note on comments. When I first wrote this, most versions of sed did not allow you to place comments inside the script. Lines starting with the '#' characters are comments. Newer versions of sed may support comments at the end of the line as well.

The Awful Truth about sed

Sed is the ultimate **s**tream **ed**itor. If that sounds strange, picture a stream flowing through a pipe. Okay, you can't see a stream if it's inside a pipe. That's what I get for attempting a flowing analogy. You want literature, read James Joyce.

Anyhow, *sed* is a marvelous utility. Unfortunately, most people never learn its real power. The language is very simple, but the documentation is terrible. The Solaris on-line manual pages for *sed* are five pages long, and two of those pages describe the 34 different errors you can get. A program that spends as much space documenting the errors than it does documenting the language has a serious learning curve.

Do not fret! It is not your fault you don't understand *sed*. I will cover *sed* completely. But I will describe the features in the order that I learned them. I didn't learn everything at once. You don't need to either.

The essential command: s for substitution

Sed has several commands, but most people only learn the substitute command: *s*. The substitute command changes all occurrences of the regular expression into a new value. A simple example is changing "day" to "night:"

```
sed s/day/night/ <old >new
```

I didn't put quotes around the argument because this example didn't need them. If you read my earlier tutorial, you would understand why it doesn't need quotes. If you have meta-characters in the command, quotes are necessary. In any case, quoting is a good habit, and I will henceforth quote future examples. That is:

```
sed 's/day/night/' <old >new
```

There are four parts to this substitute command:

```
s         Substitute command
/../../   Delimiter
day       Regular Expression Pattern String
night     Replacement string
```

We've covered quoting and regular expressions.. That's 90% of the effort needed to learn the substitute command. To put it another way, you already know how to handle 90% of the most frequent uses of *sed*. There are a few fine points that must be covered.

The slash as a delimiter

The character after the *s* is the delimiter. It is conventionally a slash, because this is what *ed*, *more*, and *vi* use. It can be anything you want, however. If you want to change a pathname that contains a slash - say /usr/local/bin to /common/bin - you could use the backslash to quote the slash:

```
sed 's/\/usr\/local\/bin/\/common\/bin/' <old >new
```

Gulp. It is easier to read if you use an underline instead of a slash as a delimiter:

```
sed 's_/usr/local/bin_/common/bin_' <old >new
```

Some people use commas, others use the "|" character. Pick one you like. As long as it's not in the string you are looking for, anything goes.

Using & as the matched string

Sometimes you want to search for a pattern and add some characters, like parenthesis, around or near the pattern you found. It is easy to do this if you are looking for a particular string:

```
sed 's/abc/(abc)/' <old >new
```

This won't work if you don't know exactly what you will find. How can you put the string you found in the replacement string if you don't know what it is?

The solution requires the special character "&." It corresponds to the pattern found.

```
sed 's/[a-z]*/(&)/' <old >new
```

You can have any number of "&" in the replacement string. You could also double a pattern, e.g. the first number of a line:

```
% echo "123 abc" | sed 's/[0-9]*/& &/'
123 123 abc
```

Let me slightly amend this example. Sed will match the first string, and make it as greedy as possible. The first match for '[0-9]*' is the first character on the line, as this matches zero of more numbers. So if the input was "abc 123" the output would be unchanged. A better way to duplicate the number is to make sure it matches a number:

```
% echo "123 abc" | sed 's/[0-9][0-9]*/& &/'
123 123 abc
```

Using 1 to keep part of the pattern

I have already described the use of "(" ")" and "1" in my tutorial on regular expressions. To review, the escaped parenthesis remember portions of the regular expression. The "1" is the first remembered pattern, and the "2" is the second remembered pattern. If you wanted to keep the first word of a line, and delete the rest of the line, mark the important part with the parenthesis:

```
sed 's/\([a-z]*\)*/\1/'
```

If you want to switch two words around, you can remember two patterns and change the order around:

```
sed 's/\([a-z]*\) \([a-z]*\)/\2 \1/'
```

The "\1" doesn't have to be in the replacement string. It can be in the pattern you are searching for. If you want to eliminate duplicated words, you can try:

```
sed 's/\([a-z]*\)\ \1/\1/'
```

You can have up to nine values: "\1" thru "\9."

Substitute Flags

You can add additional flags after the last delimiter. These flags can specify what happens when there is more than one occurrence of a pattern on a single line, and what to do if a substitution is found.

/g - Global replacement

Most Unix utilties work on files, reading a line at a time. *Sed*, by default, is the same way. If you tell it to change a word, it will only change the first occurrence of the word on a line. You may want to make the change on every word on the line instead of the first. For an example, let's place parentheses around words on a line. Instead of using a pattern like "[A-Za-z]*" which won't match words like "won't," we will use a pattern, "[^]*," that matches everything except a space. Well, this will also match anything because "*" means **zero or more**. The current version of *sed* can get unhappy with patterns like this, and generate errors like "Output line too long" or even run forever. I consider this a bug, and have reported this to Sun. As a work-around, you must avoid matching the null string when using the "g" flag to *sed*. A work-around example is: "[^][^]*." The following will put parenthesis around the first word:

```
sed 's/[^ ]*/(&)/' <old >new
```

If you want it to make changes for every word, add a "g" after the last delimiter and use the work-around:

```
sed 's/[^ ][^ ]*/(&)/g' <old >new
```

Is sed recursive?

Sed only operates on patterns found in the in-coming data. That is, the input line is read, and when a pattern is matched, the modified output is generated, and the **rest** of

the input line is scanned. The "s" command will not scan the newly created output. That is, you don't have to worry about expressions like:

```
sed 's/loop/loop the loop/g' <old >new
```

This will not cause an infinite loop. If a second "s" command is executed, it could modify the results of a previous command. I will show you how to execute multiple commands leter.

/1, /2, etc. Specifying which occurrence

With no flags, the first pattern is changed. With the "g" option, all patterns are changed. If you want to modify a particular pattern that is not the first one on the line, you could use "\(" and "\)" to mark each pattern, and use "\1" to put the first pattern back unchanged. This next example keeps the first word on the line but deletes the second:

```
sed 's/\([a-zA-Z]*\) \([a-zA-Z]*\) /\1 /' <old >new
```

Yuck. There is an easier way to do this. You can add a number after the substitution command to indicate you only want to match that particular pattern. Using this, an

```
sed 's/[a-zA-Z]* //2' <old >new
```

Note the space after the "*." Without the space, *sed* will run a long, long time. (Note: this bug is probably fixed by now.) This is because the number flag and the "g" flag have the same bug. You should also be able to use the pattern

```
sed 's/[^ ]*//2' <old >new
```

but this also eats CPU. If this worked, and it does on some Unix systems, you could remove the encrypted password from the password file:

```
sed 's/[^:]*//2' </etc/passwd >/etc/password.new
```

But this doesn't work. Using "[^:][^:]*" as a work-around doesn't help because it won't match an non-existant password, and instead delete the third field, which is the user ID! Instead you have to use the ugly parenthesis:

```
sed 's/^\([^:]*\):[^:]:/\1::/' </etc/passwd >/etc/password.new
```

You could also add a character to the first pattern so that it no longer matches the null pattern:

```
sed 's/[^:]*:/:/2' </etc/passwd >/etc/password.new
```

The number flag is not restricted to a single digit. It can be any number from 1 to 512. If you wanted to add a colon after the 80th character in each line, you could type:

```
sed 's/./&:/80' <file >new
```

/p - print

By default, *sed* prints every line. If it makes a substitution, the new text is printed instead of the old one. If you use an optional argument to sed, "sed -n," it will not, by default, print any new lines. I'll cover this and other options later. When the "-n" option is used, the "p" flag will cause the modified line to be printed. Here is one way to duplicate the function of *grep* with *sed*:

```
sed -n 's/pattern/&/p' <file
```

Write to a file with /w filename

There is one more flags that can follow the third delimiter. With it, you can specify a file that will receive the modified data. An example is the following, which will write all lines that start with an even number to the file *even*:

```
sed -n 's/^[0-9]*[02468] /&/w even' <file
```

In this example, the output file isn't needed, as the input was not modified. You must have exactly one space between the *w* and the filename. You can also have ten files open with one instance of *sed*. This allows you to split up a stream of data into separate files. Using the previous example combined with multiple substitution commands described later, you could split a file into ten pieces depending on the last digit of the first number. You could also use this method to log error or debugging information to a special file.

Combining substitution flags

You can combine flags when it makes sense. Some things that don't work are using a number and a "g" flag, because this is inconsistent. Also "w" has to be the last flag. For example the following command works:

```
sed -n 's/a/A/2pw /tmp/file' <old >new
```

Next I will discussion the options to *sed*, and different ways to invoke *sed*.

Arguments and invocation of sed

previously, I have only used one substitute command. If you need to make two changes, and you didn't want to read the manual, you could pipe together multiple *sed* commands:

```
sed 's/BEGIN/begin/' <old | sed 's/END/end/' >new
```

This used two processes instead of one. A *sed* guru never uses two processes when one can do.

Multiple commands with -e command

One method of combining multiple commands is to use a *-e* before each command:

```
sed -e 's/a/A/' -e 's/b/B/' <old >new
```

A "-e" isn't needed in the earlier examples because *sed* knows that there must always be one command. If you give *sed* one argument, it must be a command, and *sed* will edit the data read from standard input.

Also see Quoting multiple sed lines in the Bourne shell

Filenames on the command line

You can specify files on the command line if you wish. If there is more than one argument to *sed* that does not start with an option, it must be a filename. This next example will count the number of lines in three files that don't begin with a "#:"

```
sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

The *sed* substitute command changes every line that starts with a "#" into a blank line. *Grep* was used to filter out empty lines. *Wc* counts the number of lines left. *Sed* has more commands that make *grep* unnecessary. But I will cover that later.

Of course you could write the last example using the "-e" option:

```
sed -e 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

There are two other options to *sed.*

sed -n: no printing

The "-n" option will not print anything unless a explicit request to print is found. I mentioned the "/p" flag to the substitute command as one way to turn printing back on. Let me clarify this. The command

```
sed 's/PATTERN/&/p' file
```

acts like the *cat* program: e.g. nothing is changed. Add the "-n" option and the example acts like grep:

```
sed -n 's/PATTERN/&/p' file
```

sed -f scriptname

If you have a large number of *sed* commands, you can put then into a file and use

```
sed -f sedscript <old >new
```

where *sedscript* could look like this:

```
# sed comment - This script changes lower case vowels to upper case
s/a/A/g
s/e/E/g
s/i/I/g
s/o/O/g
s/u/U/g
```

When there are several commands in one file, each command must be on separate line.

Also see here

sed in shell script

If you have many commands and they won't fit neatly on one line, you can break up the line using a backslash:

```
sed -e 's/a/A/g'
-e 's/e/E/g' \
-e 's/i/I/g' \
-e 's/o/O/g' \
-e 's/u/U/g' <old >new
```

Quoting multiple sed lines in the C shell

You can have a large, multi-line *sed* script in the C shell, but you must tell the C shell that the quote is continued across several lines. This is done by placing a back slash at the end of each line:

```
#!/bin/csh -f
sed 's/a/A/g \
s/e/E/g \
s/i/I/g \
s/o/O/g \
s/u/U/g' <old >new
```

Quoting multiple sed lines in the Bourne shell

The Bourne shell makes this easier as a quote can cover several lines:

```
#!/bin/sh
sed '
s/a/A/g
s/e/E/g
s/i/I/g
s/o/O/g
s/u/U/g' <old >new
```

A sed interpreter script

Another way of executing *sed* is to use an interpreter script. Create a file that contains:

```
#!/bin/sed -f
s/a/A/g
s/e/E/g
s/i/I/g
s/o/O/g
s/u/U/g
```

Click here to get file: CapVowel.sed
If this script was stored in a file with the name "CapVowel" and was executable, you could use it with the simple command:

```
CapVowel <old >new
```

Comments

Sed comments are lines where the first non-white character is a "#." On many systems, *sed* can have only one comment, and it must be the first line of the script. On the Sun (1988 when I wrote this), you can have several comment lines anywhere in the script. Modern versions of Sed support this. If the first line contains exactly "#n" then this does the same thing as the "-n" option: turning off printing by default. This cannot be done with a *sed* interpreter script, because the first line must start with "#!/bin/sed -f."

Passing arguments into a sed script

Passing a word into a shell script that calls *sed* is easy if you remembered my tutorial on the Unix quoting mechanism. To review, you use the single quotes to turn quoting on and off. A simple shell script that uses *sed* to emulate grep is:

```
#!/bin/sh
sed -n 's/'$1'/&/p'
```

Click here to get file: sedgrep.sed
If this was stored in a file called *sedgrep*, you could type

```
sedgrep '[A-Z][A-Z]' <file
```

Using sed in a shell here-is document

You can use *sed* to prompt the user for some parameters and then create a file with those parameters filled in. You could create a file with dummy values placed inside it, and use *sed* to change those dummy values. A simpler way is to use the "here is" document, which uses part of the shell script as if it were standard input:

```
#!/bin/sh
echo -n 'what is the value? '
read value
sed  's/XXX/'$value'/' <<EOF
The value is XXX
EOF
```

Click here to get file: sed_hereis.sed
When executed, the script says:

```
what is the value?
```

If you type in "123," the next line will be:

```
The value is 123
```

I admit this is a contrived example. "Here is" documents can have values evaluated without the use of sed. This example does the same thing:

```
#!/bin/sh
echo -n 'what is the value? '
read value
cat <<EOF
The value is $value
EOF
```

However, combining "here is" documents with *sed* can be useful for some complex cases.
Note that

```
sed 's/XXX/'$value'/' <<EOF
```

will give a syntax error if the user types a space. Better form would be to use

sed 's/XXX/"'$value"'/' <<EOF

Multiple commands and order of execution

As we explore more of the commands of *sed*, the commands will become complex, and the actual sequence can be confusing. It's really quite simple. Each line is read in. The various commands, in order specified by the user, has a chance to operate on the input line. After the substitutions are made, the next command has a chance to operate on the same line, which may have been modified by earlier commands. If you ever have a question, the best way to learn what will happen is to create a small example. If a complex command doesn't work, make it simpler. If you are having problems getting a complex script working, break it up into two smaller scripts and pipe the two scripts together.

Addresses and Ranges of Text

You have only learned one command, and you can see how powerful *sed* is. However, all it is doing is a *grep* and substitute. That is, the substitute command is treating each line by itself, without caring about nearby lines. What would be useful is the ability to restrict the operation to certain lines. Some useful restrictions might be:

- Specifying a line by its number.
- Specifying a range of lines by number.
- All lines containing a pattern.
- All lines from the beginning of a file to a regular expression
- All lines from a regular expression to the end of the file.
- All lines between two regular expressions.

Sed can do all that and more. Every command in *sed* can be proceeded by an address, range or restriction like the above examples. The restriction or address immediately precedes the command:

restriction command

Restricting to a line number

The simplest restriction is a line number. If you wanted to delete the first number on line 3, just add a "3" before the command:

sed '3 s/[0-9][0-9]*//' <file >new

Patterns

Many Unix utilities like *vi* and *more* use a slash to search for a regular expression. *Sed* uses the same convention, provided you terminate the expression with a slash. To delete the first number on all lines that start with a "#," use:

```
sed '/^#/ s/[0-9][0-9]*//'
```

I placed a space after the "/*expression*/" so it is easier to read. It isn't necessary, but without it the command is harder to fathom. *Sed* does provide a few extra options when specifying regular expressions. But I'll discuss those later. If the expression starts with a backslash, the next character is the delimiter. To use a comma instead of a slash, use:

```
sed '\,^#, s/[0-9][0-9]*//'
```

The main advantage of this feature is searching for slashes. Suppose you wanted to search for the string "/usr/local/bin" and you wanted to change it for "/common/all/bin." You could use the backslash to escape the slash:

```
sed '/\/usr\/local\/bin/ s/\/usr\/local/\/common\/all/'
```

It would be easier to follow if you used an underline instead of a slash as a search. This example uses the underline in both the search command and the substitute command:

```
sed '\_/usr/local/bin_ s_/usr/local_/common/all_'
```

This illustrates why *sed* scripts get the reputation for obscurity. I could be perverse and show you the example that will search for all lines that start with a "g," and change each "g" on that line to an "s:"

```
sed '/^g/s/g/s/g'
```

Adding a space and using an underscore after the substitute command makes this **much** easier to read:

```
sed '/^g/ s_g_s_g'
```

Er, I take that back. It's hopeless. There is a lesson here: Use comments liberally in a *sed* script under SunOS. You may have to remove the comments to run the script under a different operating system, but you now know how to write a *sed* script to to that very easily! Comments are a Good Thing. You may have understood the script perfectly when you wrote it. But six months from now it could look like modem noise.

Ranges by line number

You can specify a range on line numbers by inserting a comma between the numbers. To restrict a substitution to the first 100 lines, you can use:

```
sed '1,100 s/A/a/'
```

If you know exactly how many lines are in a file, you can explicitly state that number to perform the substitution on the rest of the file. In this case, assume you used *wc* to find out there are 532 lines in the file:

```
sed '101,532 s/A/a/'
```

An easier way is to use the special character "$," which means the last line in the file.

```
sed '101,$ s/A/a/'
```

The "$" is one of those conventions that mean "last" in utilities like *cat*, *vi*, and *ed*. Line numbers are cumulative if several files are edited. That is,

```
sed '200,300 s/A/a/' f1 f2 f3 >new
```

is the same as

```
cat f1 f2 f3 | sed '200,300 s/A/a/' >new
```

Ranges by patterns

You can specify two regular expressions as the range. Assuming a "#" starts a comment, you can search for a keyword, remove all comments until you see the second keyword. In this case the two keywords are "start" and "stop:"

```
sed '/start/,/stop/ s/#.*//'
```

The first pattern turns on a flag that tells *sed* to perform the substitute command on every line. The second pattern turns off the flag. If the "start" and "stop" pattern occurs twice, the substitution is done both times. If the "stop" pattern is missing, the flag is never turned off, and the substitution will be performed on every line until the end of the file.

You should know that if the "start" pattern is found, the substitution occurs on the same line that contains "start." This turns on a switch, which is line oriented. That is, the next line is read. If it contains "stop" the switch is turned off. Otherwise, the substitution command is tried. Switches are line oriented, and not word oriented.

You can combine line numbers and regular expressions. This example will remove comments from the beginning of the file until it finds the keyword "start:"

```
sed -e '1,/start/ s/#.*//'
```

This example will remove comments everywhere except the lines **between** the two keywords:

```
sed -e '1,/start/ s/#.*//' -e '/stop/,$ s/#.*//'
```

The last example has a range that overlaps the "/start/,/stop/" range, as both ranges operate on the lines that contain the keywords. I will show you later how to restrict a command up to, **but not including** the line containing the specified pattern.

Before I start discussing the various commands, should show know that some commands cannot operate on a range of lines. I will let you know when I mention the commands. I will describe three commands in this secton. One of the three cannot operate on a range.

Delete with d

Using ranges can be confusing, so you should expect to do some experimentation when you are trying out a new script. A useful command deletes every line that matches the restriction: "d." If you want to look at the first 10 lines of a file, you can use:

```
sed '11,$ d' <file
```

which is similar in function to the *head* command. If you want to chop off the header of a mail message, which is everything up to the first blank line, use:

```
sed '1,/^$/ d' <file
```

You can duplicate the function of the *tail* command, assuming you know the length of a file. *Wc* can count the lines, and *expr* can subtract 10 from the number of lines. A Bourne shell script to look at the last 10 lines of a file might look like this:

```
#!/bin/sh
#print last 10 lines of file
# First argument is the filename
lines=`wc -l $1 | awk '{print $1}' `
start=`expr $lines - 10`
sed "1,$start d" $1
```

Click here to get file: sed_tail.sh
The range for deletions can be regular expressions pairs to mark the begin and end of the operation. Or it can be a single regular expression. Deleting all lines that start with a "#" is easy:

```
sed '/^#/ d'
```

Removing comments and blank lines takes two commands. The first removes every character from the "#" to the end of the line, and the second deletes all blank lines:

```
sed -e 's/#.*//' -e '/^$/ d'
```

A third one should be added to remove all blanks and tabs immediately before the end of line:

```
sed -e 's/#.*//' -e 's/[ ^I]*$//' -e '/^$/ d'
```

The character "^I" is a *CRTL-I* or tab character. You would have to explicitly type in the tab. Note the order of operations above, which is in that order for a very good reason. Comments might start in the middle of a line, with white space characters before them. Therefore comments are first removed from a line, potentially leaving white space characters that were before the comment. The second command removes all trailing blanks, so that lines that are now blank are converted to empty lines. The last command deletes empty lines. Together, the three commands removes all lines containing only comments, tabs or spaces.

This demonstrates the pattern space *sed* uses to operate on a line. The actual operation *sed* uses is:

- Copy the input line into the pattern space.
- Apply the first *sed* command on the pattern space, if the address restriction is true.
- Repeat with the next sed expression, again operating on the pattern space.
- When the last operation is performed, write out the pattern space and read in the next line from the input file.

Printing with p

Another useful command is the print command: "p." If *sed* wasn't started with an "-n" option, the "p" command will duplicate the input. The command

```
sed 'p'
```

will duplicate every line. If you wanted to double every empty line, use:

```
sed '/^$/ p'
```

Adding the "-n" option turns off printing unless you request it. Another way of duplicating *head*'s functionality is to print only the lines you want. This example prints the first 10 lines:

```
sed -n '1,10 p' <file
```

Sed can act like *grep* by combining the print operator to function on all lines that match a regular expression:

sed -n '/match/ p'

which is the same as:

grep match

Reversing the restriction with !

Sometimes you need to perform an action on every line except those that match a regular expression, or those outside of a range of addresses. The "!" character, which often means *not* in Unix utilities, inverts the address restriction. You remember that

sed -n '/match/ p'

acts like the *grep* command. The "-v" option to *grep* prints all lines that don't contain the pattern. *Sed* can do this with

sed -n '/match/ !p' </tmp/b

Relationships between d, p, and !

As you may have noticed, there are often several ways to solve the same problem with *sed*. This is because *print* and *delete* are opposite functions, and it appears that "!p" is similar to "d," while "!d" is similar to "p." I wanted to test this, so I created a 20 line file, and tried every different combination. The following table, which shows the results, demonstrates the difference:

```
Relations between d, p, and !
  Sed      Range   Command   Results
  ----------------------------------------------------------
  sed -n    1,10    p         Print first 10 lines
  sed -n    11,$    !p        Print first 10 lines
  sed       1,10    !d        Print first 10 lines
  sed       11,$    d         Print first 10 lines
  ----------------------------------------------------------
  sed -n    1,10    !p        Print last 10 lines
  sed -n    11,$    p         Print last 10 lines
  sed       1,10    d         Print last 10 lines
  sed       11,$    !d        Print last 10 lines
  ----------------------------------------------------------
  sed -n    1,10    d         Nothing printed
  sed -n    1,10    !d        Nothing printed
  sed -n    11,$    d         Nothing printed
  sed -n    11,$    !d        Nothing printed
  ----------------------------------------------------------
  sed       1,10    p         Print first 10 lines twice,
                              Then next 10 lines once
  sed       11,$    !p        Print first 10 lines twice,
                              Then last 10 lines once
  ----------------------------------------------------------
  sed       1,10    !p        Print first 10 lines once,
                              Then last 10 lines twice
  sed       11,$    p         Print first 10 lines once,
```

```
                                then last 10 lines twice
```

This table shows that the following commands are identical:

```
sed -n '1,10 p'
sed -n '11,$ !p'
sed '1,10 !d'
sed '11,$ d'
```

It also shows that the "!" command "inverts" the address range, operating on the other lines.

The q or quit command

There is one more simple command that can restrict the changes to a set of lines. It is the "q" command: quit. the third way to duplicate the head command is:

```
sed '11 q'
```

which quits when the eleventh line is reached. This command is most useful when you wish to abort the editing after some condition is reached.

The "q" command is the one command that does not take a range of addresses. Obviously the command

```
sed '1,10 q'
```

cannot quit 10 times. Instead

```
sed '1 q'
```

or

```
sed '10 q'
```

is correct.

Grouping with { and }

The curly braces, "{" and "}," are used to group the commands.

Hardly worth the build up. All that prose and the solution is just matching squigqles. Well, there is one complication. Since each *sed* command must start on its own line, the curly braces and the nested *sed* commands must be on separate lines.

Previously, I showed you how to remove comments starting with a "#." If you wanted to restrict the removal to lines between special "begin" and "end" key words, you could use:

```
#!/bin/sh
# This is a Borne shell script that removes #-type comments
# between 'begin' and 'end' words.
sed -n '
/begin/,/end/ {
s/#.*//
s/[ ^I]*$//
/^$/ d
p
}
'
```

Click here to get file: sed_begin_end.sh
These braces can be nested, which allow you to combine address ranges. You could perform the same action as before, but limit the change to the first 100 lines:

```
#!/bin/sh
# This is a Borne shell script that removes #-type comments
# between 'begin' and 'end' words.
sed -n '
1,100 {
/begin/,/end/ {
s/#.*//
s/[ ^I]*$//
/^$/ d
p
}
}
'
```

Click here to get file: sed_begin_end1.sh
You can place a "!" before a set of curly braces. This inverts the address, which removes comments from all lines **except** those between the two reserved words:

```
#!/bin/sh
sed '
/begin/,/end/ !{
s/#.*//
s/[ ^I]*$//
/^$/ d
p
}
'
```

Click here to get file: sed_begin_end2.sh

Writing a file with the 'w' command

You may remember that the substitute command can write to a file:

```
sed -n 's/^[0-9]*[02468] /&/w even' <file
```

I used the "&" in the replacement part of the substitution command so that the line would not be changed. A simpler example is to use the "w" command, which has the same syntax as the "w" flag in the substitute command:

```
sed -n '/^[0-9]*[02468]/ w even' <file
```

Remember - only one space must follow the command. Anything else will be considered part of the file name. The "w" command also has the same limitation as the "w" flag: only 10 files can be opened in *sed*.

Reading in a file with the 'r' command

There is also a command for reading files. The command

```
sed '$r end' <in>out
```

will append the file "end" at the end of the file (address "$)." The following will insert a file after the line with the word "INCLUDE:"

```
sed '/INCLUDE/ r file' <in >out
```

You can use the curly braces to delete the line having the "INCLUDE" command on it:

```
#!/bin/sh
sed '/INCLUDE/ {
r file
d
}'
```

Click here to get file: sed_include.sh

The order of the delete command "d" and the read file command "r" is important. Change the order and it will not work. There are two subtle actions that prevent this from working. The first is the "r" command writes the file to the output stream. The file is not inserted into the pattern space, and therefore cannot be modified by any command. Therefore the delete command does not affect the data read from the file.

The other subtlty is the "d" command deletes the current data in the pattern space. Once all of the data is deleted, it does make sense that no other action will be attempted. Therefore a "d" command executed in a curly brace also aborts all further actions. As an example, the substitute command below is never executed:

```
#!/bin/sh
# this example is WRONG
```

```
sed -e '1 {
d
s/.*//
}'
```

Click here to get file: sed_bad_example.sh

The earlier example is a crude version of the C preprocessor program. The file that is included has a predetermined name. It would be nice if *sed* allowed a variable (e.g "\1)" instead of a fixed file name. Alas, *sed* doesn't have this ability. You could work around this limitation by creating *sed* commands on the fly, or by using shell quotes to pass variables into the *sed* script. Suppose you wanted to create a command that would include a file like *cpp*, but the filename is an argument to the script. An example of this script is:

```
% include 'sys/param.h' <file.c >file.c.new
```

A shell script to do this would be:

```
#!/bin/sh
# watch out for a '/' in the parameter
# use alternate search delimiter
sed -e '\_#INCLUDE <'$1'>_{
r '$1'
d
}'
```

Click here to get file: sed_include1.sh

SunOS and the # Comment Command

As we dig deeper into *sed*, comments will make the commands easier to follow. Most versions of *sed* only allow one line as a comment, and it must be the first line. SunOS allows more than one comment, and these comments don't have to be first. The last example could be:

```
#!/bin/sh
# watch out for a '/' in the parameter
# use alternate search delimiter
sed -e '\_#INCLUDE <'$1'>_{

# read the file
r '$1'

# delete any characters in the pattern space
# and read the next line in
d
}'
```

Click here to get file: sed_include2.sh

Adding, Changing, Inserting new lines

Sed has three commands used to add new lines to the output stream. Because an entire line is added, the new line is on a line by itself to emphasize this. There is no option, an entire line is used, and it must be on its own line. If you are familiar with many unix utilities, you would expect *sed* to use a similar convention: lines are continued by ending the previous line with a "\". The syntax to these commands are finicky, like the "r" and "w" commands.

Append a line with 'a'

The "a" command appends a line after the range or pattern. This example will add a line after every line with "WORD:"

```
#!/bin/sh
sed '
/WORD/ a\
Add this line after every line with WORD
'
```

Click here to get file: sed_add_line_after_word.sh

You could eliminate two lines in the shell script if you wish:

```
#!/bin/sh
sed '/WORD/ a\
Add this line after every line with WORD'
```

Click here to get file: sed_add_line_after_word1.sh

I prefer the first form because it's easier to add a new command by adding a new line and because the intent is clearer. There must not be a space after the "\".

Insert a line with 'i'

You can insert a new line before the pattern with the "i" command:

```
#!/bin/sh
sed '
/WORD/ i\
Add this line before every line with WORD
'
```

Click here to get file: sed_add_line_before_word.sh

Change a line with 'c'

You can change the current line with a new line.

```
#!/bin/sh
sed '
/WORD/ c\
Replace the current line with the line
'
```

Click here to get file: sed_change_line.sh

A "d" command followed by a "a" command won't work, as I discussed earlier. The "d" command would terminate the current actions. You can combine all three actions using curly braces:

```
#!/bin/sh
sed '
/WORD/ {
i\
Add this line before
c\
Change the line to this one
a\
Add this line after
'
```

Click here to get file: sed_insert_append_change.sh

Leading tabs and spaces in a sed script

Sed ignores leading tabs and spaces in all commands. However these white space characters may or may not be ignored if they start the text following a "a," "c" or "i" command. In SunOS, both "features" are available. The **/usr/bin/sed** command retains white space, while the **/usr/5bin/sed** strings off leading spaces. If you want to keep leading spaces, and not care about which version of *sed* you are using, put a "\" as the first character of the file:

```
#!/bin/sh
sed '
a\
This line starts with a tab
'
```

Adding more than one line

All three commands will allow you to add more than one line. Just end each line with a "\:"

```
#!/bin/sh
sed '
/WORD/ a\
Add this line\
This line\
And this line
'
```

Adding lines and the pattern space

I have mentioned the pattern space before. Most commands operate on the pattern space, and subsequent commands may act on the results of the last modification. The three previous commands, like the read file command, add the new lines to the output stream, bypassing the pattern space.

Address ranges and the above commands

You may remember in my last tutorial I warned you that some commands can take a range of lines, and others cannot. To be precise, the commands "a," "i," "r," and "q" will not take a range like "1,100" or "/begin/,/end/." The documentation states that the read command can take a range, but I get an error when I try this. The "c" or change command allows this, and it will let you change several lines into one:

```
#!/bin/sh
sed '
/begin/,/end/ c\
***DELETED***
'
```

If you need to do this, you can use the curly braces, as that will let you perform the operation on every line:

```
#!/bin/sh
# add a blank line after every line
sed '1,$ {
a\


}'
```

Multi-Line Patterns

Most UNIX utilities are line oriented. Regular expressions are line oriented. Searching for patterns that covers more than one line is not an easy task. (Hint: It will be very shortly.)

Sed reads in a line of text, performs commands which may modify the line, and outputs modification if desired. The main loop of a *sed* script looks like this:

1. The next line is read from the input file and places in the pattern space. If the end of file is found, and if there are additional files to read, the current file is closed, the next file is opened, and the first line of the new file is placed into the pattern space.
2. The line count is incremented by one. Opening a new file does not reset this number.
3. Each *sed* command is examined. If there is a restriction places on the command, and the current line in the pattern space meets that restriction, the command is executed. Some commands, like "n" or "d" cause *sed* to go to the top of the loop. The "q" command causes *sed* to stop. Otherwise the next command is examined.
4. After all of the commands are examined, the pattern space is output unless *sed* has the optional "-n" argument.

The restriction before the command determines if the command is executed. If the restriction is a pattern, and the operation is the delete command, then the following will delete all lines that have the pattern:

```
/PATTERN/ d
```

If the restriction is a pair of numbers, then the deletion will happen if the line number is equal to the first number or greater than the first number and less than or equal to the last number:

```
10,20 d
```

If the restriction is a pair of patterns, there is a variable that is kept for each of these pairs. If the variable is false and the first pattern is found, the variable is made true. If the variable is true, the command is executed. If the variable is true, and the last pattern is on the line, after the command is executed the variable is turned off:

```
/begin/,/end/ d
```

Whew! That was a mouthful. Hey, I said it was a review. If you have read the previous lessons, you should have breezed through this. You may want to refer back to this review, because I covered several subtle points. My choice of words was deliberate. It covers some unusual cases, like:

```
# what happens if the second number
# is less than the first number?
sed -n '20,1 p' file
```

and

```
# generate a 10 line file with line numbers
# and see what happens when two patterns overlap
yes | head -10 | cat -n | \
sed -n -e '/1/,/7/ p' -e '/5/,/9/ p'
```

Enough mental punishment. Here is another review, this time in a table format. Assume the input file contains the following lines:

```
AB
CD
EF
GH
IJ
```

When *sed* starts up, the first line is placed in the pattern space. The next line line is "CD." The operations of the "n," "d," and "p" commands can be summarized as:

```
+----------------+---------+-----------------------------------------
-+
|Pattern   Next  | Command | Output           New Pattern   New Next |
|Space     Input |         |                  Space         Input    |
+----------------+---------+-----------------------------------------
-+
|AB        CD    | n       | <default>         CD           EF       |
|AB        CD    | d       | -                CD           EF       |
|AB        CD    | p       | AB               CD           EF       |
+----------------+---------+-----------------------------------------
-+
```

The "n" command may or may not generate output depending upon the existence of the "-n" flag.

That review is a little easier to follow, isn't it? Before I jump into multi-line patterns, I wanted to cover three more commands:

Print line number with =

The "=" command prints the current line number to standard output. One way to find out the line numbers that contain a pattern is to use:

```
# add line numbers first,
# then use grep,
# then just print the number
cat -n file | grep 'PATTERN' | awk '{print $1}'
```

The *sed* solution is:

```
sed -n '/PATTERN/ =' file
```

Earlier I used the following to find the number of lines in a file

```
#!/bin/sh
lines=`wc -l file | awk '{print $1}' `
```

Using the "=" command can simplify this:

```
#!/bin/sh
lines=`sed -n '$=' file `
```

The "=" command only accepts one address, so if you want to print the number for a range of lines, you must use the curly braces:

```
#!/bin/sh
# Just print the line numbers
sed -n '/begin/,/end/ {
=
d
}' file
```

Since the "=" command only prints to standard output, you cannot print the line number on the same line as the pattern. You need to edit multi-line patterns to do this.

Transform with y

If you wanted to change a word from lower case to upper case, you could write 26 character substitutions, converting "a" to "A," etc. *Sed* has a command that operates like the *tr* program. It is called the "y" command. For instance, to change the letters "a" through "f" into their upper case form, use:

```
sed 'y/abcdef/ABCDEF/' file
```

I could have used an example that converted all 26 letters into upper case, and while this column covers a broad range of topics, the "column" prefers a narrower format.

If you wanted to convert a line that contained a hexadecimal number (e.g. 0x1aff) to upper case (0x1AFF), you could use:

```
sed '/0x[0-9a-zA-Z]*/ y/abcdef/ABCDEF' file
```

This works fine if there are only numbers in the file. If you wanted to change the second word in a line to upper case, you are out of luck - unless you use multi-line editing. (Hey - I think there is some sort of theme here!)

Displaying control characters with a l

The "l" command prints the current pattern space. It is therefore useful in debugging *sed* scripts. It also converts unprintable characters into printing characters by outputting the value in octal preceded by a "\" character. I found it useful to print out the current pattern space, while probing the subtleties of *sed*.

Working with Multiple Lines

There are three new commands used in multiple-line patterns: "N," "D," and "P." I will explain their relation to the matching "n," "d," and "p" single-line commands.

The "n" command will print out the current pattern space (unless the "-n" flag is used), empty the current pattern space, and read in the next line of input. The "N" command does **not** print out the current pattern space and does **not** empty the pattern space. It reads in the next line, but appends a new line character along with the input line itself to the pattern space.

The "d" command deleted the current pattern space, reads in the next line, puts the new line into the pattern space, and aborts the current command, and starts execution at the first *sed* command. This is called starting a new "cycle." The "D" command deletes the first portion of the pattern space, up to the new line character, leaving the rest of the pattern alone. Like "d," it stops the current command and starts the command cycle over again. However, it will not print the current pattern space. You must print it yourself, as step 4 mentioned in the beginning of the article, is not executed. If the "D" command is executed with a group of other commands in a curly brace, commands after the "D" command are ignored. The next group of *sed* commands is executed, unless the pattern space is emptied. If this happens, the cycle is started from the top and a new line is read.

The "p" command prints the entire pattern space. The "P" command only prints the first part of the pattern space, up to the NEWLINE character.

Some examples might demonstrate "N" by itself isn't very useful. the filter

```
sed -e 'N'
```

doesn't modify the input stream. Instead, it combines the first and second line, then prints them, combines the third and fourth line, and prints them, etc. It does allow you to use a new "anchor" character: "\n." This matches the new line character that separates multiple lines in the pattern space. If you wanted to search for a line that ended with the character "#," and append the next line to it, you could use

```
#!/bin/sh
sed '
# look for a "#" at the end of the line
/#$/ {
# Found one - now read in the next line
N
# delete the "#" and the new line character,
s/#\n//
}' file
```

You could search for two lines containing "ONE" and "TWO" and only print out the two consecutive lines:

```
#!/bin/sh
sed -n '
/ONE/ {
# found "ONE" - read in next line
N
# look for "TWO" on the second line
# and print if there.
/\n.*TWO/ p
}' file
```

The next example would delete everything between "ONE" and "TWO:"

```
#!/bin/sh
sed '
/ONE/ {
# append a line
N
# search for TWO on the second line
/\n.*TWO/ {
# found it - now edit making one line
s/ONE.*\n.*TWO/ONE TWO/
}
}' file
```

You can either search for a particular pattern on two consecutive lines, or you can search for two consecutive words that may be split on a line boundary. The next example will look for two words which are either on the same line or one is on the end of a line and the second is on the beginning of the next line. If found, the first word is deleted:

```
#!/bin/sh
sed '
/ONE/ {
# append a line
N
# "ONE TWO" on same line
s/ONE TWO/TWO/
# "ONE
# TWO" on two consecutive lines
s/ONE\nTWO/TWO/
}
}' file
```

Let's use the
"D" command, and if we find a line containing
"TWO" immediately after a line containing
"ONE," then delete the first line:

```
#!/bin/sh
sed '
/ONE/ {
# append a line
N
# if TWO found, delete the first line
/\n.*TWO/ D
}' file
```

Click here to get file: sed_delete_line_after_word.sh

If we wanted to print the first line instead of deleting it, and not print every other line, change the "D" to a "P" and add a "-n" as an argument to *sed*:

```
#!/bin/sh
sed -n '
# by default - do not print anything
/ONE/ {
# append a line
N
# if TWO found, print the first line
/\n.*TWO/ P
}' file
```

Click here to get file: sed_print_line_after_word.sh

It is very common to combine all three multi-line commands. The typical order is "N," "P" and lastly "D." This one will delete everything between "ONE" and "TWO" if they are on one or two consecutive lines:

```
#!/bin/sh
sed '
/ONE/ {
# append the next line
N
# look for "ONE" followed by "TWO"
/ONE.*TWO/ {
# delete everything between
s/ONE.*TWO/ONE TWO/
# print
P
# then delete the first line
D
}
}' file
```

Click here to get file: sed_delete_between_two_words.sh

Earlier I talked about the "=" command, and using it to add line numbers to a file. You can use two invocations of *sed* to do this (although it is possible to do it with one, but that must wait until next section. The first *sed* command will output a line number on one line, and then print the line on the next line. The second invocation of *sed* will merge the two lines together:

```
#!/bin/sh
sed '=' file | \
sed '{
N
s/\n/ /
}'
```

Click here to get file: sed_merge_two_lines.sh

If you find it necessary, you can break one line into two lines, edit them, and merge them together again. As an example, if you had a file that had a hexadecimal number followed by a word, and you wanted to convert the first word to all upper case, you can use the "y" command, but you must first split the line into two lines, change one of the two, and merge them together. That is, a line containing

```
0x1fff table2
```

will be changed into two lines:

```
0x1fff
table2
```

and the first line will be converted into upper case. I will use *tr* to convert the space into a new line:

```
#!/bin/sh
tr ' ' '\012' file|
sed ' {
y/abcdef/ABCDEF/
N
s/\n/ /
}'
```

Click here to get file: sed_split.sh

It isn't obvious, but *sed* could be used instead of *tr*. You can embed a new line in a substitute command, but you must escape it with a backslash. It is unfortunate that you must use "\n" in the left side of a substitute command, and an embedded new line in the right hand side. Heavy sigh. Here is the example:

```
#!/bin/sh
sed '
s/ /\
/' | \
```

```
sed ' {
y/abcdef/ABCDEF/
N
s/\n/ /
}'
```

Click here to get file: sed_split_merge.sh

Sometimes I add a special character as a marker, and look for that character in the input stream. When found, it indicates the place a blank used to be. A back slash is a good character, except it must be escaped with a backslash, and makes the *sed* script obscure. Save it for that guy who keeps asking dumb questions. The *sed* script to change a blank into a "\" following by a new line would be:

```
#!/bin/sh
sed 's/ /\\\
/' file
```

Click here to get file: sed_addslash_before_blank.sh

Yeah. That's the ticket. Or use the C shell and really confuse him!

```
#!/bin/csh -f
sed '\
s/ /\\\\
/' file
```

Click here to get file: sed_addslash_before_blank.csh

A few more examples of that, and he'll never ask you a question again! I think I'm getting carried away. I'll summarize with a chart that covers the features we've talked about:

```
+----------------+---------+----------------------------------------
-+
|Pattern   Next  | Command | Output           New Pattern   New Next |
|Space     Input |         |                  Space         Input    |
+----------------+---------+----------------------------------------
-+
|AB        CD    | n       | <default>          CD          EF       |
|AB        CD    | N       | -                AB\nCD        EF       |
|AB        CD    | d       | -                CD            EF       |
|AB        CD    | D       | -                CD            EF       |
|AB        CD    | p       | AB               CD            EF       |
|AB        CD    | P       | AB               CD            EF       |
+----------------+---------+----------------------------------------
-+
|AB\nCD    EF    | n       | <default>          EF          GH       |
|AB\nCD    EF    | N       | -                AB\nCD\nEF    GH
|
|AB\nCD    EF    | d       | -                EF            GH       |
|AB\nCD    EF    | D       | -                CD            EF       |
|AB\nCD    EF    | p       | AB\nCD           AB\nCD        EF       |
|AB\nCD    EF    | P       | AB               AB\nCD        EF       |
```

```
+---------------+---------+-----------------------------------------
-+
```

Using newlines in sed scripts

Occasionally one wishes to use a new line character in a sed script. Well, this has some subtle issues here. If one wants to search for a new line, one has to use "\n." Here is an example where you search for a phrase, and delete the new line character after that phrase - joining two lines together.

```
(echo a;echo x;echo y) | sed '/x$/ {
N
s:x\n:x:
}'
```

which generates

```
a
xy
```

However, if you are inserting a new line, don't use "\n" - instead insert a literal new line character:

```
(echo a;echo x;echo y) | sed 's:x:X\
:'
```

generates

```
a
X

y
```

The Hold Buffer

So far we have talked about three concepts of *sed*: (1) The input stream or data before it is modified, (2) the output stream or data after it has been modified, and (3) the pattern space, or buffer containing characters that can be modified and send to the output stream.

There is one more "location" to be covered: the *hold buffer* or *hold space*. Think of it as a spare pattern buffer. It can be used to "copy" or "remember" the data in the pattern space for later. There are five commands that use the hold buffer.

Exchange with x

The "x" command eXchanges the pattern space with the hold buffer. By itself, the command isn't useful. Executing the *sed* command

```
sed 'x'
```

as a filter adds a blank line in the front, and deletes the last line. It looks like it didn't change the input stream significantly, but the *sed* command is modifying every line.

The hold buffer starts out containing a blank line. When the "x" command modifies the first line, line 1 is saved in the hold buffer, and the blank line takes the place of the first line. The second "x" command exchanges the second line with the hold buffer, which contains the first line. Each subsequent line is exchanged with the preceding line. The last line is placed in the hold buffer, and is not exchanged a second time, so it remains in the hold buffer when the program terminates, and never gets printed. This illustrates that care must be taken when storing data in the hold buffer, because it won't be output unless you explicitly request it.

Example of Context Grep

One use of the hold buffer is to remember previous lines. An example of this is a utility that acts like *grep* as it shows you the lines that match a pattern. In addition, it shows you the line before and after the pattern. That is, if line 8 contains the pattern, this utility would print lines 7, 8 and 9.

One way to do this is to see if the line has the pattern. If it does not have the pattern, put the current line in the hold buffer. If it does, print the line in the hold buffer, then the current line, and then the next line. After each set, three dashes are printed. The script checks for the existence of an argument, and if missing, prints an error. Passing the argument into the *sed* script is done by turning off the single quote mechanism, inserting the "$1" into the script, and starting up the single quote again:

```
#!/bin/sh
# grep3 - prints out three lines around pattern
# if there is only one argument, exit

case $# in
1);;
*) echo "Usage: $0 pattern";exit;;
esac;
# I hope the argument doesn't contain a /
# if it does, sed will complain

# use sed -n to disable printing
# unless we ask for it
sed -n '
'/$1/' !{
#no match - put the current line in the hold buffer
x
# delete the old one, which is
# now in the pattern buffer
```

```
d
}
'/$1/' {
# a match - get last line
x
# print it
p
# get the original line back
x
# print it
p
# get the next line
n
# print it
p
# now add three dashes as a marker
a\
---
# now put this line into the hold buffer
x
}'
```

Click here to get file: grep3.sh

You could use this to show the three lines around a keyword, i.e.:

```
grep3 vt100 </etc/termcap
```

Hold with h or H

The "x" command exchanges the hold buffer and the pattern buffer. Both are changed. The "h" command copies the pattern buffer into the hold buffer. The pattern buffer is unchanged. An identical script to the above uses the hold commands:

```
#!/bin/sh
# grep3 version b - another version using the hold commands
# if there is only one argument, exit

case $# in
1);;
*) echo "Usage: $0 pattern";exit;;
esac;

# again - I hope the argument doesn't contain a /

# use sed -n to disable printing
```

```
sed -n '
'/$1/' !{
# put the non-matching line in the hold buffer
h
}
'/$1/' {
# found a line that matches
# append it to the hold buffer
H
# the hold buffer contains 2 lines
# get the next line
n
# and add it to the hold buffer
H
# now print it back to the pattern space
x
# and print it.
p
# add the three hyphens as a marker
a\
---
}'
```

Click here to get file: grep3a.sh

Keeping more than one line in the hold buffer

The "H" command allows you to combine several lines in the hold buffer. It acts like the "N" command as lines are appended to the buffer, with a "\n" between the lines. You can save several lines in the hold buffer, and print them only if a particular pattern is found later.

As an example, take a file that uses spaces as the first character of a line as a continuation character. The files */etc/termcap*, */etc/printcap*, *makefile* and mail messages use spaces or tabs to indicate a continuing of an entry. If you wanted to print the entry before a word, you could use this script. I use a "^I" to indicate an actual tab character:

```
#!/bin/sh
# print previous entry
sed -n '
/^[ ^I]/!{
# line does not start with a space or tab,
# does it have the pattern we are interested in?
'/$1/' {
```

```
# yes it does. print three dashes
i\
---
# get hold buffer, save current line
x
# now print what was in the hold buffer
p
# get the original line back
x
}
# store it in the hold buffer
h
}
# what about lines that start
# with a space or tab?
/^[ ^I]/ {
# append it to the hold buffer
H
}'
```

Click here to get file: grep_previous.sh

You can also use the "H" to extend the context grep. In this example, the program prints out the two lines before the pattern, instead of a single line. The method to limit this to two lines is to use the "s" command to keep one new line, and deleting extra lines. I call it *grep4*:

```
#!/bin/sh

# grep4: prints out 4 lines around pattern
# if there is only one argument, exit

case $# in
1);;
*) echo "Usage: $0 pattern";exit;;
esac;

sed -n '
'/$1/' !{
# does not match - add this line to the hold space
H
# bring it back into the pattern space
x
# Two lines would look like .*\n.*
# Three lines look like .*\n.*\n.*
# Delete extra lines - keep two
s/^.*\n\(.*\n.*\)$/\1/
# now put the two lines (at most) into
```

```
# the hold buffer again
x
}
'/$1/' {
# matches - append the current line
H
# get the next line
n
# append that one also
H
# bring it back, but keep the current line in
# the hold buffer. This is the line after the pattern,
# and we want to place it in hold in case the next line
# has the desired pattern
x
# print the 4 lines
p
# add the mark
a\
---
}'
```

Click here to get file: grep4.sh
You can modify this to print any number of lines around a pattern. As you can see, you must remember what is in the hold space, and what is in the pattern space. There are other ways to write the same routine.

Get with g or G

Instead of exchanging the hold space with the pattern space, you can copy the hold space to the pattern space with the "g" command. This deletes the pattern space. If you want to append to the pattern space, use the "G" command. This adds a new line to the pattern space, and copies the hold space after the new line.

Here is another version of the "grep3" command. It works just like the previous one, but is implemented differently. This illustrates that *sed* has more than one way to solve many problems. What is important is you understand your problem, and document your solution:

```
#!/bin/sh
# grep3 version c: use 'G' instead of H

# if there is only one argument, exit

case $# in
1);;
*) echo "Usage: $0 pattern";exit;;
```

```
esac;

# again - I hope the argument doesn't contain a /

sed -n '
'/$1/' !{
# put the non-matching line in the hold buffer
h
}
'/$1/' {
# found a line that matches
# add the next line to the pattern space
N
# exchange the previous line with the
# 2 in pattern space
x
# now add the two lines back
G
# and print it.
p
# add the three hyphens as a marker
a\
---
# remove first 2 lines
s/.*\n.*\n\(.*\)$/\1/
# and place in the hold buffer for next time
h
}'
```

Click here to get file: grep3c.sh

The "G" command makes it easy to have two copies of a line. Suppose you wanted to the convert the first hexadecimal number to uppercase, and don't want to use the script I described in an earlier column

```
#!/bin/sh
# change the first hex number to upper case format
# uses sed twice
# used as a filter
# convert2uc <in >out
sed '
s/ /\
/' | \
sed ' {
y/abcdef/ABCDEF/
N
s/\n/ /
}'
```

Click here to get file: convert2uc.sh

Here is a solution that does not require two invocations of *sed:*

```
#!/bin/sh
# convert2uc version b
# change the first hex number to upper case format
# uses sed once
# used as a filter
# convert2uc <in >out
sed '
{
# remember the line
h
#change the current line to upper case
y/abcdef/ABCDEF/
# add the old line back
G
# Keep the first word of the first line,
# and second word of the second line
# with one humongeous regular expression
s/^\([^ ]*\) .*\n[^ ]* \(.*\)/\1 \2/
}'
```

Click here to get file: convert2uc1.sh
Carl Henrik Lunde suggested a way to make this simpler. I was working too hard.

```
#!/bin/sh
# convert2uc version b
# change the first hex number to upper case format
# uses sed once
# used as a filter
# convert2uc <in >out
sed '
{
# remember the line
h
#change the current line to upper case
y/abcdef/ABCDEF/
# add the old line back
G
# Keep the first word of the first line,
# and second word of the second line
# with one humongeous regular expression
s/ .* / / # delete all but the first and last word
}'
```

Click here to get file: convert2uc2.sh
This example only converts the letters "a" through "f" to upper case. This was chosen to make the script easier to print in these narrow columns. You can easily modify the script to convert all letters to uppercase, or to change the first letter, second word, etc.

Flow Control

As you learn about *sed* you realize that it has it's own programming language. It is true that it's a very specialized and simple language. What language would be complete without a method of changing the flow control? There are three commands *sed* uses for this. You can specify a label with an text string followed by a colon. The "b" command branches to the label. The label follows the command. If no label is there, branch to the end of the script. The "t" command is used to test conditions. Before I discuss the "t" command, I will show you an example using the "b" command.

This example remembers paragraphs, and if it contains the pattern (specified by an argument), the script prints out the entire paragraph.

```
#!/bin/sh
sed -n '
# if an empty line, check the paragraph
/^$/ b para
# else add it to the hold buffer
H
# at end of file, check paragraph
$ b para
# now branch to end of script
b
# this is where a paragraph is checked for the pattern
:para
# return the entire paragraph
# into the pattern space
x
# look for the pattern, if there - print
/'$1'/ p
'
```

Click here to get file: grep_paragraph.sh

Testing with t

You can execute a branch if a pattern is found. You may want to execute a branch only if a substitution is made. The command "t label" will branch to the label if the last substitute command modified the pattern space.

One use for this is recursive patterns. Suppose you wanted to remove white space inside parenthesis. These parentheses might be nested. That is, you would want to delete a string that looked like "(((())))." The *sed* expressions

```
sed 's/([ ^I]*)/g'
```

would only remove the innermost set. You would have to pipe the data through the script four times to remove each set or parenthesis. You could use the regular expression

```
sed 's/([ ^I()]*)/g'
```

but that would delete non-matching sets of parenthesis. The "t" command would solve this:

```
#!/bin/sh
sed '
:again
s/([ ^I]*)//g
t again
'
```

Click here to get file: delete_nested_parens.sh

An alternate way of adding comments

There is one way to add comments in a *sed* script if you don't have a version that supports it. Use the "a" command with the line number of zero:

```
#!/bin/sh
sed '
/begin/ {
0i\
This is a comment\
It can cover several lines\
It will work with any version of sed
}'
```

Click here to get file: sed_add_comments.sh

The poorly undocumented ;

There is one more *sed* command that isn't well documented. It is the ";" command. This can be used to combined several *sed* commands on one line. Here is the *grep4* script I described earlier, but without the comments or error checking and with semicolons between commands:

```
#!/bin/sh
sed -n '
'/$1/' !{;H;x;s/^.*\n\(.*\n.*\)$/\1/;x;}
'/$1/' {;H;n;H;x;p;a\
---
}'
```

Click here to get file: grep4a.sh

Yessireebob! Definitely character building. I think I have made my point. As far as I am concerned, the only time the semicolon is useful is when you want to type the *sed* script on the command line. If you are going to place it in a script, format it so it is readable. I have mentioned earlier that many versions of *sed* do not support comments except on the first line. You may want to write your scripts with comments in them, and install them in "binary" form without comments. This should not be difficult. After all, you have become a *sed* guru by now. I won't even tell you how to write a script to strip out comments. That would be insulting your intelligence.

Passing regular expressions as arguments

In the earlier scripts, I mentioned that you would have problems if you passed an argument to the script that had a slash in it. In fact, and regular expression might cause you problems. A script like the following is asking to be broken some day:

```
#!/bin/sh
sed 's/'$1'//g'
```

If the argument contains any of these characters in it you may get a broken script: "/\.*[]^$." One solution is to add a backslash before each of those characters:

```
#!/bin/sh
arg=`sed 's:[\/\.\*\[\]\^\$]:&:g <<EndOfThisMess
$1
EndOfThisMess
`
sed 's/'$arg'//g'
```

Click here to get file: sed_with_regular_expressions.sh

or

```
#!/bin/sh
arg=`echo $1 | sed 's:[\/\.\*\[\]\^\$\]:\\&:g`
sed 's/'$arg'//g
```

Click here to get file: sed_with_regular_expressions1.sh
If you were searching for the pattern "^../," the script would convert this into "\^\.\.\/" before passing it to *sed*.

Command Summary

As I promised earlier, here is a table that summarizes the different commands. The second column specifies if the command can have a range or pair of addresses (with a 2) or a single address or pattern (with a 1). The next four columns specifies which of the four buffers or streams are modified by the command. Some commands only affect the output stream, others only affect the hold buffer. If you remember that the pattern space is output (unless a "-n" was given to *sed*), this table should help you keep track of the various commands.

Command	Address or Range	Modifications to Input Stream	Output Stream	Pattern Space	Hold Buffer
=	-	-	Y	-	-
a	1	-	Y	-	-
b	2	-	-	-	-
c	2	-	Y	-	-
d	2	Y	-	Y	-
D	2	Y	-	Y	-
g	2	-	-	Y	-
G	2	-	-	Y	-
h	2	-	-	-	Y
H	2	-	-	-	Y
i	1	-	Y	-	-
l	1	-	Y	-	-
n	2	Y	*	-	-
N	2	Y	-	Y	-
p	2	-	Y	-	-
P	2	-	Y	-	-
q	1	-	-	-	-
r	1	-	Y	-	-
s	2	-	-	Y	-
t	2	-	-	-	-
w	2	-	Y	-	-
x	2	-	-	Y	Y
y	2	-	-	Y	-

The "n" command may or may not generate output, depending on the "-n" option. The "r" command can only have one address, despite the documentation.

In Conclusion

This concludes my tutorial on *sed*. It is possible to find shorter forms of some of my scripts. However, I chose these examples to illustrate some basic constructs. I wanted clarity, not obscurity. I hope you enjoyed it.

Other of my Unix shell tutorials can be found here. Other shell tutorials can be found at Heiner's SHELLdorado and Chris F. A. Johnson's Unix Shell Page

This document was originally converted from NROFF to TEXT to HTML.
please forgive errors in the translation.
If you are confused, grab the actual script if possible. No translations occured in the scripts.

Thanks to Carl Henrik Lunde who suggested an improvement to convert2uc1.sh *This document was translated by troff2html v0.21 on September 22, 2001 and then manually edited to make it compliant with:*