Grymoire Navigation

- Unix/Linux
  - Quotes
  - Bourne Shell
  - C Shell
  - File Permissions
  - Regular Expressions
  - grep
  - awk UPDATED
  - sed UPDATED
  - find
  - tar
  - inodes
- Security
  - IPv6
  - Wireless
    - Hardware
  - spam
- Deception
- PostScript
  - Halftones
- Privacy
  - Bill of Rights
- References
  - Top 10 reasons to avoid CSH
  - sed Chart PDF
  - awk Reference HTML
- Magic
- Search
- About
- Donate NEW

Google+: Bruce Barnett

Twitter: @grymoire

Blog: Wordpress

Sed - An Introduction and Tutorial by Bruce Barnett

Last modified: Thu Apr 23 16:37:48 EDT 2015

Quick Links

Sed Commands		
: label	# comment	{....} Block
= - print line number	a \ - Append	b label - Branch
c \ - change	d and D - Delete	g and G - Get
h and H - Hold	i \ - Insert	l - Look
n and N - Next	p and P - Print	q - Quit
r filename - Read File	s/..../..../ - Substitute	t label - Test
w filename - Write Filename	x - eXchange	y/..../..../ - Transform

Sed Pattern Flags
/g - Global
/I - Ignore Case
/p - Print
/w filename - Write Filename

Sed Command Line options	
Short Option (Long option)	Sed version
-n	Classic
-e script	Classic
-f scriptfile	Classic
-e script (--expression=script)	GNU sed
-f scriptfile (--file=scriptfile)	GNU sed
-h (--help)	GNU sed
-n (--quiet --silent)	GNU sed
-V (--version)	GNU sed
-r (--regexp-extended)	GNU sed
-i[SUFFIX] (--in-place[=SUFFIX])	GNU sed
-l N (--line-length=N)	GNU sed
-b (--binary)	GNU sed
-s (--separate)	GNU sed
-z (--null-data)	GNU sed
-u (--unbuffered)	GNU sed
(--follow-symlinks)	GNU sed
(--posix)	GNU sed
-i SUFFIX	Mac OS X, FreeBSD
-a	Mac OS X, FreeBSD
-l	Max OS X, FreeBSD
-E	Mac OS X, FreeBSD
-r	FreeBSD
-I SUFFIX	FreeBSD

Table of Contents

Note - You can click on the table of contents sections to jump to that section.

Then click on the section header of any section to jump back to the table of contents.

Original version written in 1994 and published in the Sun Observer

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. I'll start with the simple concepts and introduce the advanced topics later on.

When I first wrote this (in 1994), 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.

One way to think of this is that the old, "classic" version was the basis of GNU, FreeBSD and Solaris verisons of sed. And to help you understand what I had to work with, here is the **sed(1) manual page from Sun/Oracle** .

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 as 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" in the "old" file to "night" in the "new" file:

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

Or another way (for UNIX beginners),

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

and for those who want to test this:

```
echo day | sed s/day/night/
```

This will output "night".

I didn't put quotes around the argument because this example

didn't need them. If you read my earlier tutorial **on quotes** , you would understand why it doesn't need quotes. However, I recommend you do use quotes. If you have meta-characters in the command, quotes are necessary. And if you aren't sure, it's a good habit, and I will henceforth quote future examples to emphasize the "best practice." Using the strong (single quote) character, that would be:

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

I must emphasize that the sed editor changes exactly what you tell it to. So if you executed

```
echo Sunday | sed 's/day/night/'
```

This would output the word "Sunnight" because sed found the string "day" in the input.

Another important concept is that sed is line oriented. Suppose you have the input file:

```
one two three, one two three
four three two one
one hundred
```

and you used the command

```
sed 's/one/ONE/' <file
```

The output would be

```
ONE two three, one two three
```

```
four three two ONE
ONE hundred
```

Note
that
this
changed
"one"
to
"ONE"
once
on
each
line.
The
first
line
had
"one"
twice,
but
only
the
first
occurrence
was
changed.
That
is
the
default
behavior.
If
you
want
something
different,
you
will
have
to
use
some
of
the
options
that
are
available.
I'll
explain
them
later.

So
let's
continue.

There
are
four
parts
to
this
substitute
command:

```
s         Substitute command
/../../   Delimiter
one       Regular Expression Pattern Search Pattern
```

```
ONE       Replacement string
```

The search pattern is on the left hand side and the replacement string is on the right hand side.

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 any future sed

expert should know about. (You just finished section 1. There are only 63 more sections to cover. :-) Oh. And you may want to bookmark this page, ... just in case you don't finish.

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.
Some
call
this
a
'Picket
Fence'
and
it's
ugly.
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
colons:

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

Others use the "|" character.

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

Pick one you like. As long as it's not in the string you are looking for, anything goes. And remember that you need three delimiters. If you get a "Unterminated `s' command" it's because you are missing one of them.

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. I'll cover that later. If you don't want it to be so greedy (i.e. limit

the matching), you need to put restrictions on the match.

The first match for '[0-9]*' is the first character on the line, as this matches zero or more numbers. So if the input was "abc 123" the output would be unchanged (well, except for a space before the letters). 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
```

The string "abc" is unchanged, because it was not matched by the regular expression. If you wanted to eliminate "abc" from the output, you must expand the regular expression to match the rest of the line and explicitly exclude part of the expression using "(", ")" and "\1", which is the next topic.

Extended Regular Expressions

Let me add a quick comment here because

there
is
another
way
to
write
the
above
script.
"[0-9]*"
matches
zero
or
more
numbers.
"[0-9]
[0-9]*"
matches
one
or
more
numbers.
Another
way
to
do
this
is
to
use
the
"+"
meta-character
and
use
the
pattern
"[0-9]+"
as
the
"+"
is
a
special
character
when
using
"extended
regular
expressions."
Extended
regular
expressions
have
more
power,
but
sed
scripts
that
treated
"+"
as
a
normal
character
would
break.
Therefore

you must explicitly enable this extension with a command line option.

GNU sed turns this feature on if you use the "-r" command line option. So the above could also be written using

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

Mac OS X and FreeBSD uses **-E** instead of **-r** . For more information on extended regular expressions, see **Regular Expressions** and the **description of the -r command line argument**

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 parentheses (that is, parentheses with backslashes before them) remember a substring of the characters matched by the regular expression. You can use this to exclude part of the characters matched by the regular expression. The

"\1"
is
the
first
remembered
pattern,
and
the
"\2"
is
the
second
remembered
pattern.
Sed
has
up
to
nine
remembered
patterns.

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/'
```

I
should
elaborate
on
this.
Regular
expressions
are
greedy,
and
try
to
match
as
much
as
possible.
"[a-z]*"
matches

zero
or
more
lower
case
letters,
and
tries
to
match
as
many
characters
as
possible.
The
".*"
matches
zero
or
more
characters
after
the
first
match.
Since
the
first
one
grabs
all
of
the
contiguous
lower
case
letters,
the
second
matches
anything
else.
Therefore
if
you
type

```
echo abcd123 | sed 's/\([a-z]*\).*/\1/'
```

This
will
output
"abcd"
and
delete
the
numbers.

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/'
```

Note the space between the two remembered patterns. This is used to make sure two words are found. However, this will do nothing if a single word is found, or any lines with no letters. You may want to insist that words have at least one letter by using

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

or by using extended regular expressions (note

that
'('
and
')'
no
longer
need
to
have
a
backslash):

```
sed -r 's/([a-z]+) ([a-z]+)/\2 \1/' # Using GNU sed
sed -E 's/([a-z]+) ([a-z]+)/\2 \1/' # Using Apple Mac OS X
```

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

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

If
you
want
to
detect
duplicated
words,
you
can
use

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

or
with
extended
regular
expressions

```
sed -rn '/([a-z]+) \1/p' # GNU sed
sed -En '/([a-z]+) \1/p' # Mac OS X
```

This,
when
used
as
a
filter,
will
print
lines
with
duplicated
words.

The
numeric
value
can
have
up
to
nine
values:
"\1"
thru
"\9."
If
you
wanted
to
reverse
the
first
three
characters
on
a
line,
you
can
use

```
sed 's/^\(.\)\(.\)\(.\)/\3\2\1/'
```

Sed Pattern Flags

You
can
add
additional
flags
after
the
last
delimiter.
You
might

have noticed I used a 'p' at the end of the previous substitute command. I also added the '-n' option. Let me first cover the 'p' and other pattern flags. These flags can specify what happens when a match is found. Let me describe them.

/g - Global replacement

Most UNIX utilities 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 Solaris's *sed* (as I wrote this) 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 later.

/1, /2, etc. Specifying which occurrence

With no flags, the first matched substitution is changed. With the "g" option, all matches 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. Example:

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

You can combine a number with the g (global)

flag. For instance, if you want to leave the first word alone, but change the second, third, etc. to be DELETED instead, use /2g:

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

I've heard that combining the number with the g command does not work on The MacOS, and perhaps the FreeSBD version of sed as well.

Don't get /2 and \2 confused. The /2 is used at the end. \2 is used

in inside the replacement field.

Note the space after the "*" character. 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 works on your computer, 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
didn't
work
for
me
the
time
I
wrote
this.
Using
"[^:][^:]*"
as
a
work-around
doesn't
help
because
it
won't
match
an
non-existent
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
```

You can also do it the hard way by using 80 dots:

```
sed 's/^......................................................................................./&:/' <
```

/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
```

But
a
simpler
version
is
described
later

Write to a file with /w filename

There
is
one
more
flag
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, followed by a space, 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.

/I - Ignore Case

GNU has added another pattern flags - /I

This flag makes

the
pattern
match
case
insensitive.
This
will
match
abc,
aBc,
ABC,
AbC,
etc.:

```
sed -n '/abc/I p' <old >new
```

Note
that
a
space
after
the
'/I'
and
the
'p'
(print)
command
emphasizes
that
the
'p'
is
not
a
modifier
of
the
pattern
matching
process,
,
but
a
command
to
execute
after
the
pattern
matching.

Combining substitution flags

You
can
combine
flags
when
it
makes
sense.
Please
note
that

the
"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
discuss
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.

The long argument version is

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

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
```

Let's break this down into pieces. The *sed* substitute command changes every line that starts with a "#" into a blank line. *Grep* was used to filter out (delete) empty lines. *Wc* counts the number

of lines left. *Sed* has more commands that make *grep* unnecessary. And *grep* *-c* can replace *wc* *-l*. I'll discuss how you can duplicate some of *grep*'s functionality 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

an 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 if PATTERN is not in the file: e.g. nothing is changed. If PATTERN is in the file, then each line that has this is printed twice. Add the "-n" option and

the example acts like grep:

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

Nothing is printed, except those lines with PATTERN included.

The long argument of the -n command is either

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

or

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

Using 'sed /pattern/'

Sed has the ability to specify which lines are to be examined and/or modified, by specifying **addresses** before the command. I will just describe the simplest version for now -

the /PATTERN/ address. When used, only lines that match the pattern are given the command after the address. Briefly, when used with the /p flag, matching lines are printed twice:

```
sed '/PATTERN/p' file
```

And of course PATTERN is any regular expression.

Please note that if you do not include a command, such as the "p" for print, you will get an error. When I type

```
echo abc | sed '/a/'
```

I

get
the
error

```
sed: -e expression #1, char 3: missing command
```

Also,
you
don't
need
to,
but
I
recommend
that
you
place
a
space
after
the
pattern
and
the
command.
This
will
help
you
distinquish
between
flags
that
modify
the
pattern
matching,
and
commands
to
execute
after
the
pattern
is
matched.
Therefore
I
recommend
this
style:

```
sed '/PATTERN/ p' file
```

Using 'sed -n /pattern/p' to duplicate the function of

grep

If you want to duplicate the functionality of grep, combine the -n (noprint) option with the /p print flag:

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

sed -f scriptname

If you have a large number of *sed* commands, you can put them 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 a separate line.

The long argument version is

```
sed --file=sedscript <old >new
```

Also see **here on writing a script that executes sed directly**

sed in shell scripts

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 backslash 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
```

sed -V

The -V option will print the version of sed you are using. The long argument of the command is

```
sed --version
```

sed -h

The -h option will print a summary of the

sed commands. The long argument of the command is

```
sed --help
```

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 could not done with a *sed* interpreter script, because the first line must start with "#!/bin/sed -f" as I think "#!/bin/sed -nf" generated an error. It worked when I first wrote this (2008). Note that "#!/bin/sed -fn" does not work because sed thinks the filename of the script

is
"n".
However,

```
"#!/bin/sed -nf"
```

does
work.

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'

However
-
there
is
a
problem
with
this
script.
If
you
have
a
space
as
an
argument,
the
script
would
cause
a
syntax
error
A
better
version
would
protect
from
this
happening:

```
#!/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
```

This
would
allow
sed
to
act
as
the

grep
command.

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/XYZ/'$value'/' <<EOF
The value is XYZ
EOF
```

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/XYZ/'$value'/' <<EOF

will give a syntax error if the user types an answer that contains a space, like "a b c". Better form would be to put double quotes around the evaluation of the value:

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

I covered this in my **tutorial on**

quotation marks .
Click here to get file: **sed_hereis.sed**

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. Each command, 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.
You
may
have
to
remove
the
comments
to
run
the

script under a different (older) operating system, but you now know how to write a *sed* script to do 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. And if you don't understand that reference, imagine an 8-month-old child typing on a computer.

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
-e,
vi,
and
ed.
"cat
-e"
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
and
the
substitute
command
is
checked.
If
it
contains
"stop"
the
switch
is
turned
off.
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. It is in **Operating in a pattern range except**

for the patterns
But I have to cover some more basic principles.

Before I start discussing the various commands, I should explain that some commands cannot operate on a range of lines. I will let you know when I mention the commands. In this next section I will describe three commands, one of which 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 *CTRL-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 remove 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 buildup. All that prose and the solution is just matching squiggles. 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 Bourne 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 Bourne 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**

Operating in a pattern range except for the patterns

You may remember that I mentioned you can do a substitute on a pattern range, like changing "old" to "new" between a

begin/end pattern:

```
#!/bin/sh
sed '
        /begin/,/end/ s/old/new/
'
```

Another way to write this is to use the curly braces for grouping:

```
#!/bin/sh
sed '
        /begin/,/end/ {
            s/old/new/
        }
'
```

I think this makes the code clearer to understand, and easier to modify, as you will see below.

If you did not want to make any changes where the word "begin" occurred, you could simple add a new condition to skip

over
that
line:

```
#!/bin/sh
sed '
        /begin/,/end/ {
            /begin/n # skip over the line that has "begin" on it
            s/old/new/
        }
'
```

However,
skipping
over
the
line
that
has
"end"
is
trickier.
If
you
use
the
same
method
you
used
for
"begin"
then
the
sed
engine
will
not
see
the
"end"
to
stop
the
range
-
it
skips
over
that
as
well.
The
solution
is
to
do
a
substitute
on
all
lines
that
don't
have
the
"end"
by
using

```
#!/bin/sh
sed '
    /begin/,/end/ {
      /begin/n # skip over the line that has "begin" on it
      /end/ !{
        s/old/new/
      }
    }
'
```

Writing a file with the 'w' command

You may remember that the substitute command can write to a file. Here again is the example that will only write lines that start with an even number (and followed by a space):

```
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 subtlety 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
}'
```

Let
me
elaborate.
If
you
had
a
file
that
contains

```
Test first file
#INCLUDE <file1>
Test second file
#INCLUDE <file2>
```

you
could
use
the
command

```
sed_include1.sh file1<input|sed_include1.sh file2
```

to
include
the
specified
files.

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. The older versions of *sed* only allow one line as a comment, and it must be the first line. SunOS (and GNU's sed) 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 is 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
a\
Add this line after
c\
Change the line to this one
}'
```

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 Berkeley (and Linux) style sed is in /usr/bin, and the AT&T version (System V) is in /usr/5bin/.

To elaborate, the **/usr/bin/sed** command retains white space, while the **/usr/5bin/sed** strips 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
line:

```
#!/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 that earlier 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 got an error when I tried 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:

The next line is read from the input file and places it 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. The line count is incremented by one. Opening a new file does not reset this number. Each *sed* command is examined. If there is a restriction placed 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. 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.
If
you
have
read
carefully
up
to
here,
you
should
have
breezed
through
this.
You
may
want
to
refer
back,
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 is "CD." The operations of the "n," "d," and "p" commands can be summarized as:

Pattern Space	Next Input	Command	Output	New Pattern Space	New Text 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
```

Here's a sed example that convers all uppercase letters to lowercase letters, like the tr command:

```
sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/' <uppercase >lowercase
```

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,
and
you
are
using
classic
sed,
you
are
out
of
luck
-
unless
you
use
multi-line
editing.
(Hey
-
I
think
there
is
some
sort
of
theme
here!)

However,
GNU
sed
has
a
uppercase
and
lowercase
extension.

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 deletes 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,
a
step
earlier.
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. Neither the "p" nor the "P" command changes the patterns space.

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
```

Matching three lines with sed

You can match multiple lines in searches.

Here is a way to look for the string "skip3", and if found, delete that line and the next two lines.

```
#!/bin/sh
sed '/skip3/ {
          N
          N
          s/skip3\n.*\n.*/# 3 lines deleted/
}'
```

Note that it doesn't matter what the next two lines are. If you wanted to match 3 particular lines, it's a little more work.

This script looks for three lines, where the first line contains "one", the second contained "two" and the third contains "three", and if found, replace them with the string "1+2+3":

```
#!/bin/sh
sed '
/one/ {
      N
          /two/ {
                      N
                      /three/ {
                              N
                                      s/one\ntwo\nthree/1+2+3/
                                      }
                      }
          }
'
```

Matching patterns that span multiple lines

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,
and
then
use
sed
to
do
the
rest.
The
command
would
be

```
./sed_split <file
```

and
sed_split
would
be:

```
#!/bin/sh
tr ' ' '\012' |
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 backslash 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 Space	Next Input	Command	Output	New Pattern Space	New Text Input
AB	CD	n	<default>	CD	EF
AB	CD	N	-	AB\nCD	EF
AB	CD	d	-	-	EF
AB	CD	D	-	-	EF
AB	CD	p	AB	AB	CD
AB	CD	P	AB	AB	CD
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 humongous 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 humongous 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 its 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 preceded 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]*)//
        t again
'
```

An earlier version had a 'g' after the 's' expression. This is not needed.

Click here to get file: **delete_nested_parens.sh**

Debugging with l

The 'l' command will print the pattern space in an unambiguous form. Non-printing characters are printed in a C-style escaped format.

This can be useful when debugging a complex multi-line sed script.

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 documented ;

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. Also - some operating systems do NOT let you use semicolons. So if you see a script

with
semicolons,
and
it
does
not
work
on
a
non-Linux
system,
replace
the
semicolon
with
a
new
line
character.
(As
long
as
you
are
not
using
csh/tcsh,
but
that's
another
topic.

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,

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: "/\.*[]^$" For instance, if someone types a "/" then the substitute command will see four delimiters instead of three. You will also get syntax errors if you provide a "]" without

a
"]".
One
solution
is
to
have
the
user
put
a
backslash
before
any
of
these
characters
when
they
pass
it
as
an
argument.
However,
the
user
has
to
know
which
characters
are
special.
Another
solution
is
to
add
a
backslash
before
each
of
those
characters
in
the
script

```
#!/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.

Inserting binary characters

Dealing
with
binary
characters
can
be
trick,
expecially
when
writing
scripts
for
people
to
read.
I
can
insert
a
binary
character
using
an
editor
like
EMACS
but
if
I
show
the
binary
character,
the
terminal
may
change
it
to
show
it
to
you.

The
easiest
way
I
have

found to do this in a script in a portable fashion is to use the tr(1) command. It understands octal notations, and it can be output into a variable which can be used.

Here's a script that will replace the string "ding" with the ASCII bell character:

```
#!/bin/sh
BELL=`echo x | tr 'x' '\007'`
sed "s/ding/$BELL/"
```

Please note that I used double quotes. Since special characters are interpreted, you have to be careful

when you use this mechanism.

GNU sed Command Line arguments

One of the conventions UNIX systems have is to use single letters are command line arguments. This makes typing faster, and shorted, which is an advantage if you are in a contest. Normal people often find sed's terseness cryptic. You can improve the readability of sed scripts by using the long word equivalent

options. That is, instead of typing

```
sed -n 20p
```

You can type the long word version of the -n argument

```
sed --quiet 20p
```

Or

```
sed --silent 20p
```

The long form of sed's command line arguments always have 2 hyphens before their names. GNU sed has the following long-form command line arguments:

GNU Command Line Arguments	
Short Form	Long Form
-n	--quiet --silent
-e script	--expression=SCRIPT
-f SCRIPTFILE	--file=SCRIPTFILE

-i[SUFFIX]	--in-place[=SUFFIX]
-l N	--line-length=N
	--posix
-b	--binary
	--follow-symlinks
-r	--regular-extended
-s	--separate
-u	--unbuffered
	--help
	--version

Let's define each of these.

The -posix argument

The GNU version of sed has many features that are not available in other versions. When portability is important, test your script with the

-posix option. If you had an example that used a feature of GNU sed, such as the 'v' command to test the version number, such as

```
#this is a sed command file
v 4.0.1
# print the number of lines
$=
```

And you executed it with the command

```
sed -nf sedfile --posix <file
```

then the GNU version of sed program would give you a warning that your sed script is not compatible. It would report:

```
sed: -e expression #1, char 2: unknown command: `v'
```

The --version argument

You can

determine which version of sed you are using with the GNU sed --version command. This is what it outputs on my computer

```
# sed --version
GNU sed version 4.2.1
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE,
to the extent permitted by law.

GNU sed home page: <http://www.gnu.org/software/sed/>.
General help using GNU software: <http://www.gnu.org/gethelp/>.
E-mail bug reports to: <bug-gnu-utils@gnu.org>.
Be sure to include the word ``sed'' somewhere in the ``Subject:'' field.
```

The -h Help argument

The -h option will print a summary of the sed commands. The long argument of the command is

```
sed --help
```

It provides a nice summary of the command line

arguments.

The -l Line Length Argument

I've already described the 'l' command. The default line width for the 'l' command is 70 characters. This default value can be changed by adding the '-l N' option and specifying the maximum line length as the number after the '-l'.

```
sed -n -l 80 'l' <file
```

The long form version of the command line is

```
 sed -n --line-length=80 'l' <file
```

The -s

Separate argument

Normally, when you specify several files on the command line, sed concatenates the files into one stream, and then operates on that single stream. If you had three files, each with 100 lines, then the command

```
sed -n '1,10 p' file1 file2 file3
```

would only print the first 10 lines of file file1. The '-s' command tells GNU sed to treat the files are independent files, and to print out the first 10 lines

of
each
file,
which
is
similar
to
the
head
command.
Here's
another
example:
If
you
wanted
to
print
the
number
of
lines
of
each
file,
you
could
use
'wc
-l'
which
prints
the
number
of
lines,
and
the
filename,
for
each
file,
and
at
the
end
print
the
total
number
of
lines.
Here
is
a
simple
shell
script
that
does
something
similar,
just
using
sed:

```
#!/bin/sh
FILES=$*
sed -s -n '$=' $FILES # print the number of lines for each file
sed -n '$=' $FILES # print the total number of lines.
```

The
'wc
-l'
command
does
print
out
the
filenames,
unlike
the
above

script. A better emulation of the 'wc -l' command would execute the command in a loop, and print the filenames. Here is a more advanced script that does this, but it doesn't use the '-s' command:

```
#!/bin/sh
for F in "$@"
do
 NL=`sed -n '$=' < "$F" ` &&  printf "  %d %s\n" $NL "$F"
done
TOTAL=`sed -n '$=' "$@"`
printf "  %d total\n" $TOTAL
```

The -i in-place argument

I've already described in Editing multiple files the way I like to do this. For those who want a simpler method, GNU Sed

allows you to do this with a command line option - "-i". Let's assume that we are going to make the same simple change - adding a tab before each line. This is a way to do this for all files in a directory with the ".txt" extension in the current directory:

```
sed -i 's/^/\t/' *.txt
```

The long argument name version is

```
sed --in-place 's/^/\t/' *.txt
```

This verison deletes the original file. If you

are as cautious as I am, you may prefer to specify an extension, which is used to keep a copy of the original:

```
sed -i.tmp 's/^/\t/' *.txt
```

And the long argument name version is

```
sed --in-place=.tmp 's/^/\t/'  *.txt
```

In the last two versions, the original version of the "a.txt" file would have the name "a.txt.tmp". You can then delete the original files after you make sure all worked as you expected. Please consider the backup option, and heed my warning. You can easily delete the

backed-up original file, as long as the extension is unique.

The GNU version of sed allows you to use "-i" without an argument. The FreeBSD/Mac OS X does not. You **must** provide an extension for the FreeBSD/Mac OS X version. If you want to do in-place editing without creating a backup, you can use

```
sed -i ''  's/^/\t/'  *.txt
```

The --follow-symlinks argument

The in-place editing feature is

handy to have. But what happens if the file you are editing is a symbolic link to another file? Let's assume you have a file named "b" in a directory called "tmp", with a symbolic link to this file:

```
$ ls -l b
lrwxrwxrwx 1 barnett adm 6 Mar 16 16:03 b.txt -> tmp/b.txt
```

If you executed the above command to do in place editing, there will be a new file called "b.txt" in the current directory, and "tmp/b.txt" will be unchanged. Now you have two

versions of the file, one is changed (in the current directory), and one is not (in the "tmp" directory). And where you had a symbolic link, it has been replaced with a modified version of the original file. If you want to edit the real file, and keep the symbolic link in place, use the "--follow-symlinks" command line option:

```
sed -i --follow-symlinks 's/^/\t/' *.txt
```

This follows the symlink to the original location, and modifies the file in the "tmp" directory, If you specify an

extension, the original file will be found with that extension in the same directory ar the real source. Without the --follow-symlinks command line option, the "backup" file "b.tmp" will be in the same directory that held the symbolic link, and will still be a symbolic link - just renamed to give it a new extension.

The -b Binary argument

Unix and Linux systems consider the new line character "\n" to be the

end
of
the
line.
However,
MS-DOS,
Windows,
and
Cygwin
systems
end
each
line
with
"\r\n"
-
Carriage
return
and
line-feed.
If
you
are
using
any
of
these
operating
systems,
the
"-b"
or
--binary"
command
line
option
will
treat
the
carriage
return/new
line
combination
as
the
end
of
the
line.
Otherwise
the
carriage
return
is
treated
as
an
unprintable
character
immediately
before
the
end-of-line.
I
think.
(Note
to
self
-

verify this).

The -r Extended Regular Expression argument

When I mention patterns, such as "s/pattern/", the pattern is a regular expression. There are two common classes of regular expressions, the original "basic" expressions, and the "extended" regular expressions. For more on the differences see **My tutorial on regular expressions** and the **the section on extended regular expressions** . Because the meaning of certain

characters are different between the regular and extended expressions, you need a command line argument to enable *sed* to use the extension. To enable this extension, use the "-r" command, as mentioned in **the example on finding duplicated words on a line**

```
sed -r -n '/\([a-z]+\) \1/p'
```

or

```
sed --regular-extended -quiet '/\([a-z]+\) \1/p'
```

I already mentioned that Mac OS X and FreeBSD uses **-E** instead of **-r** .

The -u Unbuffered

argument

Normally - Unix and Linux systems apply some intelligence to handling standard output. It's assumed that if you are sending results to a terminal, you want the output as soon as it becomes available. However, if you are sending the output to a file, then it's assumed you want better performance, so it buffers the output until the buffer is full, and then the contents of

the buffer is written to the file. Let me elaborate on this. Let's assume for this example you have a very large file, and you are using sed to search for a string, and to print it when it is found:

```
sed -n '/MATCH/p' <file
```

Since the output is the terminal, as soon as a match is found, it is printed. However, if sed pipes its output to another program, it will buffer the results.

But there are times when you want immediate results. This is especially true when you are dealing with large files, or files that occasionally generate data. To summarize, you have lots of input data, and you want sed to process it, and then send this to another program that processes the results, but you want the results when it happens, and not delayed. Let me make up a simple example. It's contrived, but it does explain how this works. Here's a program

called SlowText that prints numbers from 1 to 60, once a second:

```
#!/bin/sh
for i in `seq 1 60`
do
  echo $i
  sleep 1
done
```

Let's use sed to search for lines that have the character '1', and have it send results to awk, which will calculate the square of that number. This would be the admittedly contrived script:

```
SlowText | sed -n '/1/p' | awk '{print $1*$1}'
```

This works, but because sed is buffering the results, we have to wait until the buffer fills up, or until the SlowText program exists, before we

the results. You can eliminate the buffering, and see the results as soon as SlowText outputs them, by using the "-u" option. With this option, you will see the squares printed as soon as possible:

```
SlowText | sed -un '/1/p' | awk '{print $1*$1}'
```

The long form of the argument is "--unbuffered".

Mac OS X and FreeBSD use the argument "-l".

GNU Sed 4.2.2 and later will also be unbuffered while reading files, not just writing them.

The -z Null Data argument

Normally, sed reads a line by reading a string of characters up to the end-of-line character (new line or carriage return). See the -b Binary command line argument
The GNU version of sed added a feature in version 4.2.2 to use the "NULL" character instead. This can be useful if you have files that use the NULL as a

record separator. Some GNU utilities can genertae output that uses a NULL instead a new line, such as "find . -print0" or "grep -lZ". This feature is useful if you are operating on filenames that might contain spaces or binary characters.

For instance, if you wanted to use "find" to search for files and you used the "-print0" option to print a NULL at the end of each

filename, you could use sed to delete the directory pathname:

```
find . -type f -print0  | sed -z 's:^.*/::' | xargs -0 echo
```

The above example is not terribly useful as the "xargs" use of echo does not retain the ability to retain spaces as part of the filename. But is does show how to use the sed "-z" command.

GNU grep also has a -Z option to search for strings in files, placing a "NULL" at the

end
of
each
filename
instead
of
a
new
line.
And
with
the
-l
command,
grep
will
print
the
filename
that
contains
the
string,
retaining
non-printing
and
binary
characters:

```
grep -lZ STRING */*/* | sed -z 's:^.*/::' | xargs -0 echo
```

This
feature
is
very
useful
when
users
have
the
ability
to
create
their
own
filenames.

FreeBSD Extensions

Apple
uses
the
FreeBSD
version
of
sed
for
Mac
OS
X
instead
of
the
GNU
sed.
However,

the FreeBSD version has a couple of additions.

The -a or delayed open Argument

Normally, as soon as sed starts up, it opens all files that are refered to by the "w" command. The FreeBSD version of sed has an option to delay this action until the "w" command is executed.

The -I in-place argument

FreeBSD added

a "-I" option that is similar to the **-i** option. The "-i" option treats the editing each file as a separate instance of *sed*. If the "-I" option is used, then line numbers do not get reset at the beginning of each line, and ranges of addresses continue from one file to the next. That is, if you used the range '/BEGIN/,/END/' and you used the "-I" option,

you can have the "BEGIN" in the first file, and "END" in the second file, and the commands executed within the range would span both files. If you used "-i", then the commands would not.

And like the **-i** option, the extension used to store the backup file must be specified.

-E or Extended Regular Expressions

I mentioned extended regular expressions **earlier** .

FreeBSD (and Mac OS X) uses "-E" to enable this. However, FreeBSD later added the -r command to be compatible with GNU sed.

Using word boundries

Someone once asked me to help them solve a tricky sed problem involving word boundaries. Let's suppose you have the following input

```
/usr/bin /usr/local/bin /usr/local /usr/local/project/bin
```

and you wanted to delete '/usr/local' but leave the other 3 paths alone. You

could use the simple (and incorrect) command:

```
sed 's@/usr/local@@'
```

which would output

```
/usr/bin /bin /usr/local /usr/local/project/bin
```

That is, it would mistakenly change '/usr/local/bin' to '/bin' and not delete '/usr/local' which was the intention of the programmer. The better method is to include spaces around the search:

```
sed 's@ /usr/local @ @'
```

However, this won't work if '/usr/local' is at the beginning, or at the end of the line. It also won't work

if '/usr/local' is the only path on the line. To handle these edge cases, you can simply describe all of these conditions as separate cases:

```
#!/bin/sh
sed '
s@ /usr/local @ @g
s@^/usr/local @@
s@ /usr/local$@@
s@^/usr/local$@@
'
```

This works fine if the string you are searching for is surrounded by a space. But what happens if the string is surrounded by other characters, which may be one of several possible characters? You can

always
make
up
your
own
class
of
characters
that
define
the
'end
of
a
word';
For
instance,
if
your
string
consists
of
alphanumeric
characters
and
the
slash,
the
class
of
characters
can
be
defined
by
'[a-zA-
Z0-9/]'
or
the
more
flexible
'[[:alnum:]/]'.
We
can
define
the
class
fo
characters
to
be
all
but
these,
by
using
the
caret,
i.e.
'[^[:alnum:]/]'.
And
unlike
the
space
before,
if
you
are
going

to use character classes, you may have to remember what these characters are and not delete them. So we can replace the space with '[^[:alnum:]/]' and then change the command to be

```
#!/bin/sh
sed '
s@\([^[:alnum:]/]\)/usr/local\([^[:alnum:]/]\)@\1\2@g
s@^/usr/local\([^[:alnum:]/]\)@\1@
s@\([^[:alnum:]/]\)/usr/local$@\1@
s@^/usr/local$@@
'
```

The first version would replace ' /usr/local ' with a single space. This method would replace ':/usr /local:' with '::' - because the redundant deliniators are not deleted. Be

sure to fix this if you need to.

This method always works, but it is inelegant and error prone. There are other methods, but they **may not be portable**. Solaris's version of sed used the special characters '\<' and '\>' as anchors that indicated a word boundary. So you could use

```
s@\</usr/local\>@@
```

However, the GNU version of sed says the usage of these special characters are

undefined.
According
to
the
manual
page:

```
Regex syntax clashes (problems with backslashes)
     `sed' uses the POSIX basic regular expression syntax.  According to
     the standard, the meaning of some escape sequences is undefined in
     this syntax;  notable in the case of `sed' are `\|', `\+', `\?',
     `\`', `\'', `\<', `\>', `\b', `\B', `\w', and `\W'.

     As in all GNU programs that use POSIX basic regular expressions,
     `sed' interprets these escape sequences as special characters.
     So, `x\+' matches one or more occurrences of `x'.  `abc\|def'
     matches either `abc' or `def'.
```

When
in
doubt,
experiment.

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
or
a
single
address
or
pattern.
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	Modification to Input Stream	Modification to Output Stream	Modification to Pattern Space	Modification to Hold Buffer
=	-	-	Y	-	-
a	Address	-	Y	-	-
b	Range	-	-	-	-
c	Range	-	Y	-	-
d	Range	Y	-	Y	-
D	Range	Y	-	Y	-
g	Range	-	-	Y	-
G	Range	-	-	Y	-
h	Range	-	-	-	Y
H	Range	-	-	-	Y
i	Address	-	Y	-	-
l	Address	-	Y	-	-
n	Range	Y	*	-	-
N	Range	Y	-	Y	-

p	Range	-	Y	-	-
P	Range	-	Y	-	-
q	Address	-	-	-	-
r	Address	-	Y	-	-
s	Range	-	-	Y	-
t	Range	-	-	-	-
w	Range	-	Y	-	-
x	Range	-	-	Y	Y
y	Range	-	-	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.

Check out my new **Sed Reference Chart**

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.

More References

This concludes my tutorial on sed. Other of my UNIX shell tutorials can be found here. Other shell tutorials and references can be found at

- FreeBSD Sed Man Page
- Apple/Mac OS X Sed Man Page
- GNU Sed Manual
- GNU Sed 4.2.2 Release

Notes
sed(1)
Seventh
Eddition
Unix
sed(1)
manual
page
from
Sun/Oracle
Heiner's
SHELLdorado
Chris
F.
A.
Johnson's
UNIX
Shell
Page
The
Wikipedia
Entry
on
SED
SED
one-liners

And
don't
forget
The
SED
FAQ

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
occurred
in
the
scripts.

Thanks for the feedback, gang

Thanks to Keelan Evans, Fredrik Nilsson, and Kurt McKee for spotting some typos. Thanks to Wim Stolker and Jose' Sebrosa as well. Thanks to Olivier Mengue. Thanks to Andrew M. Goth. Thanks to David P. Brown. Thanks to Axel Schulze for some corrections Thanks to Martin Jan for the corrections in sed format (grin) Thanks to David Ward for some

corrections
A
big
thanks
for
Fazl
Rahman
for
spotting
dozens
of
errors.
Thanks
to
Carl
Henrik
Lunde
who
suggested
an
improvement
to
convert2uc1.sh
A
big
thanks
to
Bryan
Hyun
Huh
who
spotted
an
error
in
the
table
and
reference
chart
Thanks
for
input
from

Marten Jan
Gordon Wilson
Tom Konantz
Peter Bratton
Grant Root
Keith Briggs
Zoltan Miklos
Peggy Russell
Lorens Kockkum.net
John Poulin
Rihards
Corey Richardson
Eric Mathison
Ildar Mulyukov
Tom Zhu
Abhijeet Rastogi **@shadyabhi**
Steve LeBlanc **@sleveo**
dontforget yourtowel **@whatissixbynine**
Yiming
Fei Wang
Kenneth R. Beesley
Duncan Sung W. Kim **@DuncanSungWKim**
Juan Eugenio Abadie
Zander Hill **@_ZPH**
Rob Smith
Peter Moore

This document was translated by troff2html v0.21 on September 22, 2001 and then manually edited to make it compliant with: