GNU sed, a stream editor

version 4.5, 8 April 2018

by Ken Pizzini, Paolo Bonzini

This file documents version 4.5 of GNU `sed`, a stream editor.

Table of Contents

1 Introduction 1

2 Running sed 1

2.1 Overview 1
2.2 Command-Line Options 2
2.3 Exit status 5

3 `sed` scripts 5

3.1 `sed` script overview 5
3.2 `sed` commands summary 6
3.3 The `s` Command 8
3.4 Often-Used Commands 10
3.5 Less Frequently-Used Commands 12
3.6 Commands for `sed` gurus 17
3.7 Commands Specific to GNU `sed` 17
3.8 Multiple commands syntax 18
3.8.1 Commands Requiring a newline 19

4 Addresses: selecting lines 21

4.1 Addresses overview 22
4.2 Selecting lines by numbers 22
4.3 selecting lines by text matching 23
4.4 Range Addresses 24

5 Regular Expressions: selecting text 26

5.1 Overview of regular expression in `sed` 26
5.2 Basic (BRE) and extended (ERE) regular expression 26
5.3 Overview of basic regular expression syntax 27
5.4 Overview of extended regular expression syntax 29
5.5 Character Classes and Bracket Expressions 30
5.6 regular expression extensions 32
5.7 Back-references and Subexpressions 33
5.8 Escape Sequences - specifying special characters 34
5.8.1 Escaping Precedence 34
5.9 Multibyte characters and Locale Considerations 35
5.9.1 Invalid multibyte characters 35
5.9.2 Upper/Lower case conversion 37
5.9.3 Multibyte regexp character classes 37

6 Advanced `sed`: cycles and buffers 37
6.1 How `sed` Works 37
6.2 Hold and Pattern Buffers 38
6.3 Multiline techniques - using D,G,H,N,P to process multiple lines .. 38
6.4 Branching and Flow Control 40
6.4.1 Branching and Cycles 41
6.4.2 Branching example: joining lines 42

7 Some Sample Scripts 43
7.1 Joining lines 43
7.2 Centering Lines 44
7.3 Increment a Number 45
7.4 Rename Files to Lower Case 46
7.5 Print `bash` Environment 48
7.6 Reverse Characters of Lines 49
7.7 Text search across multiple lines 50
7.8 Line length adjustment 51
7.9 Reverse Lines of Files 52
7.10 Numbering Lines 53
7.11 Numbering Non-blank Lines 54
7.12 Counting Characters 55
7.13 Counting Words 56
7.14 Counting Lines 57
7.15 Printing the First Lines 58
7.16 Printing the Last Lines 58
7.17 Make Duplicate Lines Unique 58
7.18 Print Duplicated Lines of Input 59
7.19 Remove All Duplicated Lines 60
7.20 Squeezing Blank Lines 60

8 GNU `sed`'s Limitations and Non-limitations .. 62

9 Other Resources for Learning About `sed` 62

10 Reporting Bugs 63

Appendix A GNU Free Documentation License .. 66

Concept Index 74

Command and Option Index 78

1 Introduction

`sed` is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as `ed`), `sed` works by making only one pass over the input(s), and is consequently more efficient. But it is `sed`'s ability to filter text in a pipeline which particularly distinguishes it from other types of editors.

2 Running sed

This chapter covers how to run `sed`. Details of `sed` scripts and individual `sed` commands are discussed in the next chapter.

2.1 Overview

Normally `sed` is invoked like this:

```
sed SCRIPT INPUTFILE...
```

For example, to replace all occurrences of '`hello`' to '`world`' in the file `input.txt`:

```
sed 's/hello/world/' input.txt > output.txt
```

If you do not specify *INPUTFILE*, or if *INPUTFILE* is `-`, `sed` filters the contents of the standard input. The following commands are equivalent:

```
sed 's/hello/world/' input.txt > output.txt
sed 's/hello/world/' < input.txt > output.txt
cat input.txt | sed 's/hello/world/' - > output.txt
```

`sed` writes output to standard output. Use `-i` to edit files in-place instead of printing to standard output. See also the `W` and `s///w` commands for writing output to other files. The following command modifies `file.txt` and does not produce any output:

```
sed -i 's/hello/world/' file.txt
```

By default `sed` prints all processed input (except input that has been modified/deleted by commands such as `d`). Use `-n` to suppress output, and the `p` command to print specific lines. The following command prints only line 45 of the input file:

```
sed -n '45p' file.txt
```

`sed` treats multiple input files as one long stream. The following example prints the first line of the first file (`one.txt`) and the last line of the last file (`three.txt`). Use `-s` to reverse this behavior.

```
sed -n  '1p ; $p' one.txt two.txt three.txt
```

Without `-e` or `-f` options, `sed` uses the first non-option parameter as the *script*, and the following non-option parameters as input files. If `-e` or `-f` options are used to specify a *script*, all non-option parameters are taken as input files. Options `-e` and `-f` can be combined, and can appear multiple times (in which case the final effective *script* will be concatenation of all the individual *script*s).

The following examples are equivalent:

```
sed 's/hello/world/' input.txt > output.txt
```

```
sed -e 's/hello/world/' input.txt > output.txt
sed --expression='s/hello/world/' input.txt > output.txt

echo 's/hello/world/' > myscript.sed
sed -f myscript.sed input.txt > output.txt
sed --file=myscript.sed input.txt > output.txt
```

2.2 Command-Line Options

The full format for invoking `sed` is:

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

`sed` may be invoked with the following command-line options:

`--version`
: Print out the version of `sed` that is being run and a copyright notice, then exit.

`--help`
: Print a usage message briefly summarizing these command-line options and the bug-reporting address, then exit.

`-n`
`--quiet`
`--silent`
: By default, `sed` prints out the pattern space at the end of each cycle through the script (see Section 6.1 [How `sed` works], page 37). These options disable this automatic printing, and `sed` only produces output when explicitly told to via the `p` command.

`-e` *script*
`--expression=`*script*
: Add the commands in *script* to the set of commands to be run while processing the input.

`-f` *script-file*
`--file=`*script-file*
: Add the commands contained in the file *script-file* to the set of commands to be run while processing the input.

`-i[`*SUFFIX*`]`
`--in-place[=`*SUFFIX*`]`
: This option specifies that files are to be edited in-place. GNU `sed` does this by creating a temporary file and sending output to this file rather than to the standard output.[1].

  This option implies `-s`.

  When the end of the file is reached, the temporary file is renamed to the output file's original name. The extension, if supplied, is used to modify the name of the old file before renaming the temporary file, thereby making a backup copy[2]).

[1] This applies to commands such as `=`, `a`, `c`, `i`, `l`, `p`. You can still write to the standard output by using the `w` or `W` commands together with the `/dev/stdout` special file

[2] Note that GNU `sed` creates the backup file whether or not any output is actually changed.

This rule is followed: if the extension doesn't contain a `*`, then it is appended to the end of the current filename as a suffix; if the extension does contain one or more `*` characters, then *each* asterisk is replaced with the current filename. This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix, or even to place backup copies of the original files into another directory (provided the directory already exists).

If no extension is supplied, the original file is overwritten without making a backup.

For the DJGPP port, if only SFN support is available, the backup file name will be truncated to the well known `8+3` length. This rule is followed: the suffix will remove as many characters as necessary from a potentially existing extension to fit into the 3 characters long space available for extensions; if a prefix is given, it will shift to the right as many as characters are necessary to fit into the 8 characters long space available for file names. As example, the following command:

```
sed -ibck_*_up s/foobar/raboof/ filename.txt
```

will produce a backup file for `filename.txt` with `bck_file._up` as backup file name. As can be seen the suffix `_up` is 3 characters long an overwrites the file name's extension `ext` completely. The prefix `bck_` is 4 characters long and occupies the place of the first 4 characters of the file name, so that the last 4 characters of the original file name are lost.

Because `-i` takes an optional argument, it should not be followed by other short options:

`sed -Ei '...' FILE`
: Same as `-E -i` with no backup suffix - `FILE` will be edited in-place without creating a backup.

`sed -iE '...' FILE`
: This is equivalent to `--in-place=E`, creating `FILEE` as backup of `FILE`

Be cautious of using `-n` with `-i`: the former disables automatic printing of lines and the latter changes the file in-place without a backup. Used carelessly (and without an explicit `p` command), the output file will be empty:

```
# WRONG USAGE: 'FILE' will be truncated.
sed -ni 's/foo/bar/' FILE
```

`-l` *N*
`--line-length=`*N*
: Specify the default line-wrap length for the `l` command. A length of 0 (zero) means to never wrap long lines. If not specified, it is taken to be 70.

`--posix`
: GNU `sed` includes several extensions to POSIX sed. In order to simplify writing portable scripts, this option disables all the extensions that this manual documents, including additional commands. Most of the extensions accept `sed` programs that are outside the syntax mandated by POSIX, but some of them (such as the behavior of the `N` command described in Chapter 10 [Reporting

Bugs], page 63) actually violate the standard. If you want to disable only the latter kind of extension, you can set the `POSIXLY_CORRECT` variable to a non-empty value.

`-b`
`--binary` This option is available on every platform, but is only effective where the operating system makes a distinction between text files and binary files. When such a distinction is made—as is the case for MS-DOS, Windows, Cygwin—text files are composed of lines separated by a carriage return *and* a line feed character, and `sed` does not see the ending CR. When this option is specified, `sed` will open input files in binary mode, thus not requesting this special processing and considering lines to end at a line feed.

`--follow-symlinks`
This option is available only on platforms that support symbolic links and has an effect only if option `-i` is specified. In this case, if the file that is specified on the command line is a symbolic link, `sed` will follow the link and edit the ultimate destination of the link. The default behavior is to break the symbolic link, so that the link destination will not be modified.

`-E`
`-r`
`--regexp-extended`
Use extended regular expressions rather than basic regular expressions. Extended regexps are those that `egrep` accepts; they can be clearer because they usually have fewer backslashes. Historically this was a GNU extension, but the `-E` extension has since been added to the POSIX standard (http://austingroupbugs.net/view.php?id=528), so use `-E` for portability. GNU sed has accepted `-E` as an undocumented option for years, and *BSD seds have accepted `-E` for years as well, but scripts that use `-E` might not port to other older systems. See Section 5.4 [Extended regular expressions], page 29.

`-s`
`--separate`
By default, `sed` will consider the files specified on the command line as a single continuous long stream. This GNU `sed` extension allows the user to consider them as separate files: range addresses (such as '`/abc/,/def/`') are not allowed to span several files, line numbers are relative to the start of each file, `$` refers to the last line of each file, and files invoked from the `R` commands are rewound at the start of each file.

`--sandbox`
In sandbox mode, `e/w/r` commands are rejected - programs containing them will be aborted without being run. Sandbox mode ensures `sed` operates only on the input files designated on the command line, and cannot run external programs.

`-u`
`--unbuffered`
: Buffer both input and output as minimally as practical. (This is particularly useful if the input is coming from the likes of '`tail -f`', and you wish to see the transformed output as soon as possible.)

`-z`
`--null-data`
`--zero-terminated`
: Treat the input as a set of lines, each terminated by a zero byte (the ASCII '`NUL`' character) instead of a newline. This option can be used with commands like '`sort -z`' and '`find -print0`' to process arbitrary file names.

If no `-e`, `-f`, `--expression`, or `--file` options are given on the command-line, then the first non-option argument on the command line is taken to be the *script* to be executed.

If any command-line parameters remain after processing the above, these parameters are interpreted as the names of input files to be processed. A file name of '`-`' refers to the standard input stream. The standard input will be processed if no file names are specified.

2.3 Exit status

An exit status of zero indicates success, and a nonzero value indicates failure. GNU `sed` returns the following exit status error values:

0
: Successful completion.

1
: Invalid command, invalid syntax, invalid regular expression or a GNU `sed` extension command used with `--posix`.

2
: One or more of the input file specified on the command line could not be opened (e.g. if a file is not found, or read permission is denied). Processing continued with other files.

4
: An I/O error, or a serious processing error during runtime, GNU `sed` aborted immediately.

Additionally, the commands `q` and `Q` can be used to terminate `sed` with a custom exit code value (this is a GNU `sed` extension):

```
$ echo | sed 'Q42' ; echo $?
42
```

3 sed scripts

3.1 sed script overview

A `sed` program consists of one or more `sed` commands, passed in by one or more of the `-e`, `-f`, `--expression`, and `--file` options, or the first non-option argument if zero of these options are used. This document will refer to "the" `sed` script; this is understood to mean the in-order concatenation of all of the *script*s and *script-file*s passed in. See Section 2.1 [Overview], page 1.

sed commands follow this syntax:

```
[addr]X[options]
```

X is a single-letter sed command. [addr] is an optional line address. If [addr] is specified, the command *X* will be executed only on the matched lines. [addr] can be a single line number, a regular expression, or a range of lines (see Chapter 4 [sed addresses], page 21). Additional [options] are used for some sed commands.

The following example deletes lines 30 to 35 in the input. 30,35 is an address range. d is the delete command:

```
sed '30,35d' input.txt > output.txt
```

The following example prints all input until a line starting with the word 'foo' is found. If such line is found, sed will terminate with exit status 42. If such line was not found (and no other error occurred), sed will exit with status 0. /^foo/ is a regular-expression address. q is the quit command. 42 is the command option.

```
sed '/^foo/q42' input.txt > output.txt
```

Commands within a *script* or *script-file* can be separated by semicolons (;) or newlines (ASCII 10). Multiple scripts can be specified with -e or -f options.

The following examples are all equivalent. They perform two sed operations: deleting any lines matching the regular expression /^foo/, and replacing all occurrences of the string 'hello' with 'world':

```
sed '/^foo/d ; s/hello/world/' input.txt > output.txt

sed -e '/^foo/d' -e 's/hello/world/' input.txt > output.txt

echo '/^foo/d' > script.sed
echo 's/hello/world/' >> script.sed
sed -f script.sed input.txt > output.txt

echo 's/hello/world/' > script2.sed
sed -e '/^foo/d' -f script2.sed input.txt > output.txt
```

Commands a, c, i, due to their syntax, cannot be followed by semicolons working as command separators and thus should be terminated with newlines or be placed at the end of a *script* or *script-file*. Commands can also be preceded with optional non-significant whitespace characters. See Section 3.8 [Multiple commands syntax], page 18.

3.2 sed commands summary

The following commands are supported in GNU sed. Some are standard POSIX commands, while other are GNU extensions. Details and examples for each command are in the following sections. (Mnemonics) are shown in parentheses.

a\
text Append *text* after a line.

a *text* Append *text* after a line (alternative syntax).

b *label* Branch unconditionally to *label*. The *label* may be omitted, in which case the next cycle is started.

`c\`
text
: Replace (change) lines with *text*.

`c` *text*
: Replace (change) lines with *text* (alternative syntax).

`d`
: Delete the pattern space; immediately start next cycle.

`D`
: If pattern space contains newlines, delete text in the pattern space up to the first newline, and restart cycle with the resultant pattern space, without reading a new line of input.

  If pattern space contains no newline, start a normal new cycle as if the `d` command was issued.

`e`
: Executes the command that is found in pattern space and replaces the pattern space with the output; a trailing newline is suppressed.

`e` *command*
: Executes *command* and sends its output to the output stream. The command can run across multiple lines, all but the last ending with a back-slash.

`F`
: (filename) Print the file name of the current input file (with a trailing newline).

`g`
: Replace the contents of the pattern space with the contents of the hold space.

`G`
: Append a newline to the contents of the pattern space, and then append the contents of the hold space to that of the pattern space.

`h`
: (hold) Replace the contents of the hold space with the contents of the pattern space.

`H`
: Append a newline to the contents of the hold space, and then append the contents of the pattern space to that of the hold space.

`i\`
text
: insert *text* before a line.

`i` *text*
: insert *text* before a line (alternative syntax).

`l`
: Print the pattern space in an unambiguous form.

`n`
: (next) If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then `sed` exits without processing any more commands.

`N`
: Add a newline to the pattern space, then append the next line of input to the pattern space. If there is no more input then `sed` exits without processing any more commands.

`p`
: Print the pattern space.

`P`
: Print the pattern space, up to the first <newline>.

`q`*[exit-code]*
: (quit) Exit `sed` without processing any more commands or input.

`Q`*[exit-code]*
: (quit) This command is the same as `q`, but will not print the contents of pattern space. Like `q`, it provides the ability to return an exit code to the caller.

`r filename`
: Reads file *filename*.

`R filename`
: Queue a line of *filename* to be read and inserted into the output stream at the end of the current cycle, or when the next input line is read.

`s/regexp/replacement/[flags]`
: (substitute) Match the regular-expression against the content of the pattern space. If found, replace matched string with *replacement*.

`t label`
: (test) Branch to *label* only if there has been a successful substitution since the last input line was read or conditional branch was taken. The *label* may be omitted, in which case the next cycle is started.

`T label`
: (test) Branch to *label* only if there have been no successful substitutions since the last input line was read or conditional branch was taken. The *label* may be omitted, in which case the next cycle is started.

`v [version]`
: (version) This command does nothing, but makes `sed` fail if GNU `sed` extensions are not supported, or if the requested version is not available.

`w filename`
: Write the pattern space to *filename*.

`W filename`
: Write to the given filename the portion of the pattern space up to the first newline

`x`
: Exchange the contents of the hold and pattern spaces.

`y/src/dst/`
: Transliterate any characters in the pattern space which match any of the *source-chars* with the corresponding character in *dest-chars*.

`z`
: (zap) This command empties the content of pattern space.

`#`
: A comment, until the next newline.

`{ cmd ; cmd ... }`
: Group several commands together.

`=`
: Print the current input line number (with a trailing newline).

`: label`
: Specify the location of *label* for branch commands (`b`, `t`, `T`).

3.3 The `s` Command

The `s` command (as in substitute) is probably the most important in `sed` and has a lot of different options. The syntax of the `s` command is '`s/regexp/replacement/flags`'.

Its basic concept is simple: the `s` command attempts to match the pattern space against the supplied regular expression *regexp*; if the match is successful, then that portion of the pattern space which was matched is replaced with *replacement*.

For details about *regexp* syntax see Section 4.3 [Regular Expression Addresses], page 23.

The *replacement* can contain `\n` (*n* being a number from 1 to 9, inclusive) references, which refer to the portion of the match which is contained between the *n*th `\(` and its matching `\)`. Also, the *replacement* can contain unescaped `&` characters which reference the whole matched portion of the pattern space.

The `/` characters may be uniformly replaced by any other single character within any given `s` command. The `/` character (or whatever other character is used in its stead) can appear in the *regexp* or *replacement* only if it is preceded by a `\` character.

Finally, as a GNU `sed` extension, you can include a special sequence made of a backslash and one of the letters `L`, `l`, `U`, `u`, or `E`. The meaning is as follows:

`\L` Turn the replacement to lowercase until a `\U` or `\E` is found,

`\l` Turn the next character to lowercase,

`\U` Turn the replacement to uppercase until a `\L` or `\E` is found,

`\u` Turn the next character to uppercase,

`\E` Stop case conversion started by `\L` or `\U`.

When the `g` flag is being used, case conversion does not propagate from one occurrence of the regular expression to another. For example, when the following command is executed with '`a-b-`' in pattern space:

```
s/\(b\?\)-/x\u\1/g
```

the output is '`axxB`'. When replacing the first '`-`', the '`\u`' sequence only affects the empty replacement of '`\1`'. It does not affect the `x` character that is added to pattern space when replacing `b-` with `xB`.

On the other hand, `\l` and `\u` do affect the remainder of the replacement text if they are followed by an empty substitution. With '`a-b-`' in pattern space, the following command:

```
s/\(b\?\)-/\u\1x/g
```

will replace '`-`' with '`X`' (uppercase) and '`b-`' with '`Bx`'. If this behavior is undesirable, you can prevent it by adding a '`\E`' sequence—after '`\1`' in this case.

To include a literal `\`, `&`, or newline in the final replacement, be sure to precede the desired `\`, `&`, or newline in the *replacement* with a `\`.

The `s` command can be followed by zero or more of the following *flags*:

`g` Apply the replacement to *all* matches to the *regexp*, not just the first.

`number` Only replace the *number*th match of the *regexp*.

interaction in `s` command Note: the POSIX standard does not specify what should happen when you mix the `g` and *number* modifiers, and currently there is no widely agreed upon meaning across `sed` implementations. For GNU `sed`, the interaction is defined to be: ignore matches before the *number*th, and then match and replace all matches from the *number*th on.

`p` If the substitution was made, then print the new pattern space.

Note: when both the `p` and `e` options are specified, the relative ordering of the two produces very different results. In general, `ep` (evaluate then print) is what you want, but operating the other way round can be useful for debugging. For

this reason, the current version of GNU sed interprets specially the presence of p options both before and after e, printing the pattern space before and after evaluation, while in general flags for the s command show their effect just once. This behavior, although documented, might change in future versions.

w *filename*
: If the substitution was made, then write out the result to the named file. As a GNU sed extension, two special values of *filename* are supported: /dev/stderr, which writes the result to the standard error, and /dev/stdout, which writes to the standard output.[3]

e
: This command allows one to pipe input from a shell command into pattern space. If a substitution was made, the command that is found in pattern space is executed and pattern space is replaced with its output. A trailing newline is suppressed; results are undefined if the command to be executed contains a NUL character. This is a GNU sed extension.

I
i
: The I modifier to regular-expression matching is a GNU extension which makes sed match *regexp* in a case-insensitive manner.

M
m
: The M modifier to regular-expression matching is a GNU sed extension which directs GNU sed to match the regular expression in *multi-line* mode. The modifier causes ^ and $ to match respectively (in addition to the normal behavior) the empty string after a newline, and the empty string before a newline. There are special character sequences (\` and \') which always match the beginning or the end of the buffer. In addition, the period character does not match a new-line character in multi-line mode.

3.4 Often-Used Commands

If you use sed at all, you will quite likely want to know these commands.

#
: [No addresses allowed.]

  The # character begins a comment; the comment continues until the next newline.

  If you are concerned about portability, be aware that some implementations of sed (which are not POSIX conforming) may only support a single one-line comment, and then only when the very first character of the script is a #.

  Warning: if the first two characters of the sed script are #n, then the -n (no-autoprint) option is forced. If you want to put a comment in the first line of your script and that comment begins with the letter 'n' and you do not want this behavior, then be sure to either use a capital 'N', or place at least one space before the 'n'.

q [*exit-code*]
: Exit sed without processing any more commands or input.

[3] This is equivalent to p unless the -i option is being used.

Example: stop after printing the second line:

```
$ seq 3 | sed 2q
1
2
```

This command accepts only one address. Note that the current pattern space is printed if auto-print is not disabled with the `-n` options. The ability to return an exit code from the `sed` script is a GNU `sed` extension.

See also the GNU `sed` extension `Q` command which quits silently without printing the current pattern space.

`d` Delete the pattern space; immediately start next cycle.

Example: delete the second input line:

```
$ seq 3 | sed 2d
1
3
```

`p` Print out the pattern space (to the standard output). This command is usually only used in conjunction with the `-n` command-line option.

Example: print only the second input line:

```
$ seq 3 | sed -n 2p
2
```

`n` If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then `sed` exits without processing any more commands.

This command is useful to skip lines (e.g. process every Nth line).

Example: perform substitution on every 3rd line (i.e. two `n` commands skip two lines):

```
$ seq 6 | sed 'n;n;s/./x/'
1
2
x
4
5
x
```

GNU `sed` provides an extension address syntax of *first~step* to achieve the same result:

```
$ seq 6 | sed '0~3s/./x/'
1
2
x
4
5
x
```

`{ commands }`

A group of commands may be enclosed between `{` and `}` characters. This is particularly useful when you want a group of commands to be triggered by a single address (or address-range) match.

Example: perform substitution then print the second input line:

```
$ seq 3 | sed -n '2{s/2/X/ ; p}'
X
```

3.5 Less Frequently-Used Commands

Though perhaps less frequently used than those in the previous section, some very small yet useful `sed` scripts can be built with these commands.

`y/source-chars/dest-chars/`

Transliterate any characters in the pattern space which match any of the *source-chars* with the corresponding character in *dest-chars*.

Example: transliterate '`a-j`' into '`0-9`':

```
$ echo hello world | sed 'y/abcdefghij/0123456789/'
74llo worl3
```

(The `/` characters may be uniformly replaced by any other single character within any given `y` command.)

Instances of the `/` (or whatever other character is used in its stead), `\`, or newlines can appear in the *source-chars* or *dest-chars* lists, provide that each instance is escaped by a `\`. The *source-chars* and *dest-chars* lists *must* contain the same number of characters (after de-escaping).

See the `tr` command from GNU coreutils for similar functionality.

`a text` Appending *text* after a line. This is a GNU extension to the standard `a` command - see below for details.

Example: Add the word '`hello`' after the second line:

```
$ seq 3 | sed '2a hello'
1
2
hello
3
```

Leading whitespace after the `a` command is ignored. The text to add is read until the end of the line.

`a\`
text Appending *text* after a line.

Example: Add '`hello`' after the second line (⊣ indicates printed output lines):

```
$ seq 3 | sed '2a\
hello'
⊣1
⊣2
⊣hello
```

```
⊣3
```

The `a` command queues the lines of text which follow this command (each but the last ending with a `\`, which are removed from the output) to be output at the end of the current cycle, or when the next input line is read.

As a GNU extension, this command accepts two addresses.

Escape sequences in *text* are processed, so you should use `\\` in *text* to print a single backslash.

The commands resume after the last line without a backslash (`\`) - '`world`' in the following example:

```
$ seq 3 | sed '2a\
hello\
world
3s/./X/'
⊣1
⊣2
⊣hello
⊣world
⊣X
```

As a GNU extension, the `a` command and *text* can be separated into two `-e` parameters, enabling easier scripting:

```
$ seq 3 | sed -e '2a\' -e hello
1
2
hello
3

$ sed -e '2a\' -e "$VAR"
```

`i` *text*

insert *text* before a line. This is a GNU extension to the standard `i` command - see below for details.

Example: Insert the word '`hello`' before the second line:

```
$ seq 3 | sed '2i hello'
1
hello
2
3
```

Leading whitespace after the `i` command is ignored. The text to add is read until the end of the line.

`i\`
text

Immediately output the lines of text which follow this command.

Example: Insert '`hello`' before the second line (⊣ indicates printed output lines):

```
$ seq 3 | sed '2i\
hello'
```

```
⊣1
⊣hello
⊣2
⊣3
```

As a GNU extension, this command accepts two addresses.

Escape sequences in *text* are processed, so you should use `\\` in *text* to print a single backslash.

The commands resume after the last line without a backslash (`\`) - '`world`' in the following example:

```
$ seq 3 | sed '2i\
hello\
world
s/./X/'
⊣X
⊣hello
⊣world
⊣X
⊣X
```

As a GNU extension, the `i` command and *text* can be separated into two `-e` parameters, enabling easier scripting:

```
$ seq 3 | sed -e '2i\' -e hello
1
hello
2
3

$ sed -e '2i\' -e "$VAR"
```

`c` *text* Replaces the line(s) with *text*. This is a GNU extension to the standard `c` command - see below for details.

Example: Replace the 2nd to 9th lines with the word '`hello`':

```
$ seq 10 | sed '2,9c hello'
1
hello
10
```

Leading whitespace after the `c` command is ignored. The text to add is read until the end of the line.

`c\`
text Delete the lines matching the address or address-range, and output the lines of text which follow this command.

Example: Replace 2nd to 4th lines with the words '`hello`' and '`world`' (⊣ indicates printed output lines):

```
$ seq 5 | sed '2,4c\
hello\
world'
```

```
⊣1
⊣hello
⊣world
⊣5
```

If no addresses are given, each line is replaced.

A new cycle is started after this command is done, since the pattern space will have been deleted. In the following example, the `c` starts a new cycle and the substitution command is not performed on the replaced text:

```
$ seq 3 | sed '2c\
hello
s/./X/'
⊣X
⊣hello
⊣X
```

As a GNU extension, the `c` command and *text* can be separated into two `-e` parameters, enabling easier scripting:

```
$ seq 3 | sed -e '2c\' -e hello
1
hello
3

$ sed -e '2c\' -e "$VAR"
```

`=` Print out the current input line number (with a trailing newline).

```
$ printf '%s\n' aaa bbb ccc | sed =
1
aaa
2
bbb
3
ccc
```

As a GNU extension, this command accepts two addresses.

`l` *n* Print the pattern space in an unambiguous form: non-printable characters (and the `\` character) are printed in C-style escaped form; long lines are split, with a trailing `\` character to indicate the split; the end of each line is marked with a `$`.

n specifies the desired line-wrap length; a length of 0 (zero) means to never wrap long lines. If omitted, the default as specified on the command line is used. The *n* parameter is a GNU `sed` extension.

`r` *filename*

Reads file *filename*. Example:

```
$ seq 3 | sed '2r/etc/hostname'
1
2
fencepost.gnu.org
```

3

Queue the contents of *filename* to be read and inserted into the output stream at the end of the current cycle, or when the next input line is read. Note that if *filename* cannot be read, it is treated as if it were an empty file, without any error indication.

As a GNU `sed` extension, the special value `/dev/stdin` is supported for the file name, which reads the contents of the standard input.

As a GNU extension, this command accepts two addresses. The file will then be reread and inserted on each of the addressed lines.

`w` *filename*

Write the pattern space to *filename*. As a GNU `sed` extension, two special values of *filename* are supported: `/dev/stderr`, which writes the result to the standard error, and `/dev/stdout`, which writes to the standard output.[4]

The file will be created (or truncated) before the first input line is read; all `w` commands (including instances of the `w` flag on successful `s` commands) which refer to the same *filename* are output without closing and reopening the file.

`D` If pattern space contains no newline, start a normal new cycle as if the `d` command was issued. Otherwise, delete text in the pattern space up to the first newline, and restart cycle with the resultant pattern space, without reading a new line of input.

`N` Add a newline to the pattern space, then append the next line of input to the pattern space. If there is no more input then `sed` exits without processing any more commands.

When `-z` is used, a zero byte (the ascii '`NUL`' character) is added between the lines (instead of a new line).

By default `sed` does not terminate if there is no 'next' input line. This is a GNU extension which can be disabled with `--posix`. See [N command on the last line], page 63.

`P` Print out the portion of the pattern space up to the first newline.

`h` Replace the contents of the hold space with the contents of the pattern space.

`H` Append a newline to the contents of the hold space, and then append the contents of the pattern space to that of the hold space.

`g` Replace the contents of the pattern space with the contents of the hold space.

`G` Append a newline to the contents of the pattern space, and then append the contents of the hold space to that of the pattern space.

`x` Exchange the contents of the hold and pattern spaces.

[4] This is equivalent to `p` unless the `-i` option is being used.

3.6 Commands for sed gurus

In most cases, use of these commands indicates that you are probably better off programming in something like awk or Perl. But occasionally one is committed to sticking with sed, and these commands can enable one to write quite convoluted scripts.

`: label` [No addresses allowed.]

Specify the location of *label* for branch commands. In all other respects, a no-op.

`b label` Unconditionally branch to *label*. The *label* may be omitted, in which case the next cycle is started.

`t label` Branch to *label* only if there has been a successful substitution since the last input line was read or conditional branch was taken. The *label* may be omitted, in which case the next cycle is started.

3.7 Commands Specific to GNU sed

These commands are specific to GNU sed, so you must use them with care and only when you are sure that hindering portability is not evil. They allow you to check for GNU sed extensions or to do tasks that are required quite often, yet are unsupported by standard seds.

`e [command]`

This command allows one to pipe input from a shell command into pattern space. Without parameters, the e command executes the command that is found in pattern space and replaces the pattern space with the output; a trailing newline is suppressed.

If a parameter is specified, instead, the e command interprets it as a command and sends its output to the output stream. The command can run across multiple lines, all but the last ending with a back-slash.

In both cases, the results are undefined if the command to be executed contains a NUL character.

Note that, unlike the r command, the output of the command will be printed immediately; the r command instead delays the output to the end of the current cycle.

`F` Print out the file name of the current input file (with a trailing newline).

`Q [exit-code]`

This command accepts only one address.

This command is the same as q, but will not print the contents of pattern space. Like q, it provides the ability to return an exit code to the caller.

This command can be useful because the only alternative ways to accomplish this apparently trivial function are to use the -n option (which can unnecessarily complicate your script) or resorting to the following snippet, which wastes time by reading the whole file without any visible effect:

```
:eat
$d          Quit silently on the last line
```

```
N       Read another line, silently
g       Overwrite pattern space each time to save memory
b eat
```

`R` *filename*
: Queue a line of *filename* to be read and inserted into the output stream at the end of the current cycle, or when the next input line is read. Note that if *filename* cannot be read, or if its end is reached, no line is appended, without any error indication.

  As with the `r` command, the special value `/dev/stdin` is supported for the file name, which reads a line from the standard input.

`T` *label*
: Branch to *label* only if there have been no successful substitutions since the last input line was read or conditional branch was taken. The *label* may be omitted, in which case the next cycle is started.

`v` *version*
: This command does nothing, but makes `sed` fail if GNU `sed` extensions are not supported, simply because other versions of `sed` do not implement it. In addition, you can specify the version of `sed` that your script requires, such as `4.0.5`. The default is `4.0` because that is the first version that implemented this command.

  This command enables all GNU extensions even if `POSIXLY_CORRECT` is set in the environment.

`W` *filename*
: Write to the given filename the portion of the pattern space up to the first newline. Everything said under the `w` command about file handling holds here too.

`z`
: This command empties the content of pattern space. It is usually the same as '`s/.*//`', but is more efficient and works in the presence of invalid multibyte sequences in the input stream. POSIX mandates that such sequences are *not* matched by '`.`', so that there is no portable way to clear `sed`'s buffers in the middle of the script in most multibyte locales (including UTF-8 locales).

3.8 Multiple commands syntax

There are several methods to specify multiple commands in a `sed` program.

Using newlines is most natural when running a sed script from a file (using the `-f` option).

On the command line, all `sed` commands may be separated by newlines. Alternatively, you may specify each command as an argument to an `-e` option:

```
$ seq 6 | sed '1d
3d
5d'
2
4
6

$ seq 6 | sed -e 1d -e 3d -e 5d
2
4
6
```

A semicolon (';') may be used to separate most simple commands:

```
$ seq 6 | sed '1d;3d;5d'
2
4
6
```

The {,},b,t,T,: commands can be separated with a semicolon (this is a non-portable GNU sed extension).

```
$ seq 4 | sed '{1d;3d}'
2
4

$ seq 6 | sed '{1d;3d};5d'
2
4
6
```

Labels used in b,t,T,: commands are read until a semicolon. Leading and trailing white-space is ignored. In the examples below the label is 'x'. The first example works with GNU sed. The second is a portable equivalent. For more information about branching and labels see Section 6.4 [Branching and flow control], page 40.

```
$ seq 3 | sed '/1/b x ; s/^/=/ ; :x ; 3d'
1
=2

$ seq 3 | sed -e '/1/bx' -e 's/^/=/' -e ':x' -e '3d'
1
=2
```

3.8.1 Commands Requiring a newline

The following commands cannot be separated by a semicolon and require a newline:

a,c,i (append/change/insert)
: All characters following a,c,i commands are taken as the text to append/change/insert. Using a semicolon leads to undesirable results:

```
$ seq 2 | sed '1aHello ; 2d'
1
Hello ; 2d
2
```

Separate the commands using `-e` or a newline:

```
$ seq 2 | sed -e 1aHello -e 2d
1
Hello
```

```
$ seq 2 | sed '1aHello
2d'
1
Hello
```

Note that specifying the text to add (‘`Hello`’) immediately after `a`,`c`,`i` is itself a GNU `sed` extension. A portable, POSIX-compliant alternative is:

```
$ seq 2 | sed '1a\
Hello
2d'
1
Hello
```

`#` (comment)

All characters following ‘`#`’ until the next newline are ignored.

```
$ seq 3 | sed '# this is a comment ; 2d'
1
2
3
```

```
$ seq 3 | sed '# this is a comment
2d'
1
3
```

`r`,`R`,`w`,`W` (reading and writing files)

The `r`,`R`,`w`,`W` commands parse the filename until end of the line. If whitespace, comments or semicolons are found, they will be included in the filename, leading to unexpected results:

```
$ seq 2 | sed '1w hello.txt ; 2d'
1
2

$ ls -log
total 4
-rw-rw-r-- 1 2 Jan 23 23:03 hello.txt ; 2d

$ cat 'hello.txt ; 2d'
1
```

Note that `sed` silently ignores read/write errors in `r`,`R`,`w`,`W` commands (such as missing files). In the following example, `sed` tries to read a file named '`hello.txt ; N`'. The file is missing, and the error is silently ignored:

```
$ echo x | sed '1rhello.txt ; N'
x
```

`e` (command execution)

Any characters following the `e` command until the end of the line will be sent to the shell. If whitespace, comments or semicolons are found, they will be included in the shell command, leading to unexpected results:

```
$ echo a | sed '1e touch foo#bar'
a

$ ls -1
foo#bar

$ echo a | sed '1e touch foo ; s/a/b/'
sh: 1: s/a/b/: not found
a
```

`s///[we]` (substitute with `e` or `w` flags)

In a substitution command, the `w` flag writes the substitution result to a file, and the `e` flag executes the subsitution result as a shell command. As with the `r/R/w/W/e` commands, these must be terminated with a newline. If whitespace, comments or semicolons are found, they will be included in the shell command or filename, leading to unexpected results:

```
$ echo a | sed 's/a/b/w1.txt#foo'
b

$ ls -1
1.txt#foo
```

4 Addresses: selecting lines

4.1 Addresses overview

Addresses determine on which line(s) the `sed` command will be executed. The following command replaces the word '`hello`' with '`world`' only on line 144:

```
sed '144s/hello/world/' input.txt > output.txt
```

If no addresses are given, the command is performed on all lines. The following command replaces the word '`hello`' with '`world`' on all lines in the input file:

```
sed 's/hello/world/' input.txt > output.txt
```

Addresses can contain regular expressions to match lines based on content instead of line numbers. The following command replaces the word '`hello`' with '`world`' only in lines containing the word '`apple`':

```
sed '/apple/s/hello/world/' input.txt > output.txt
```

An address range is specified with two addresses separated by a comma (`,`). Addresses can be numeric, regular expressions, or a mix of both. The following command replaces the word '`hello`' with '`world`' only in lines 4 to 17 (inclusive):

```
sed '4,17s/hello/world/' input.txt > output.txt
```

Appending the `!` character to the end of an address specification (before the command letter) negates the sense of the match. That is, if the `!` character follows an address or an address range, then only lines which do *not* match the addresses will be selected. The following command replaces the word '`hello`' with '`world`' only in lines *not* containing the word '`apple`':

```
sed '/apple/!s/hello/world/' input.txt > output.txt
```

The following command replaces the word '`hello`' with '`world`' only in lines 1 to 3 and 18 till the last line of the input file (i.e. excluding lines 4 to 17):

```
sed '4,17!s/hello/world/' input.txt > output.txt
```

4.2 Selecting lines by numbers

Addresses in a `sed` script can be in any of the following forms:

`number`
: Specifying a line number will match only that line in the input. (Note that `sed` counts lines continuously across all input files unless `-i` or `-s` options are specified.)

`$`
: This address matches the last line of the last file of input, or the last line of each file when the `-i` or `-s` options are specified.

`first~step`
: This GNU extension matches every *step*th line starting with line *first*. In particular, lines will be selected when there exists a non-negative *n* such that the current line-number equals *first* + (*n* * *step*). Thus, one would use `1~2` to select the odd-numbered lines and `0~2` for even-numbered lines; to pick every third line starting with the second, '`2~3`' would be used; to pick every fifth line starting with the tenth, use '`10~5`'; and '`50~0`' is just an obscure way of saying `50`.

  The following commands demonstrate the step address usage:

  ```
  $ seq 10 | sed -n '0~4p'
  ```

```
4
8

$ seq 10 | sed -n '1~3p'
1
4
7
10
```

4.3 selecting lines by text matching

GNU `sed` supports the following regular expression addresses. The default regular expression is Section 5.3 [Basic Regular Expression (BRE)], page 27. If `-E` or `-r` options are used, The regular expression should be in Section 5.4 [Extended Regular Expression (ERE)], page 29, syntax. See Section 5.2 [BRE vs ERE], page 26.

/regexp/ This will select any line which matches the regular expression *regexp*. If *regexp* itself includes any `/` characters, each must be escaped by a backslash (\).

The following command prints lines in `/etc/passwd` which end with '`bash`'[5]:

```
sed -n '/bash$/p' /etc/passwd
```

The empty regular expression '`//`' repeats the last regular expression match (the same holds if the empty regular expression is passed to the `s` command). Note that modifiers to regular expressions are evaluated when the regular expression is compiled, thus it is invalid to specify them together with the empty regular expression.

\%regexp%

(The `%` may be replaced by any other single character.)

This also matches the regular expression *regexp*, but allows one to use a different delimiter than `/`. This is particularly useful if the *regexp* itself contains a lot of slashes, since it avoids the tedious escaping of every `/`. If *regexp* itself includes any delimiter characters, each must be escaped by a backslash (\).

The following two commands are equivalent. They print lines which start with '`/home/alice/documents/`':

```
sed -n '/^\/home\/alice\/documents\//p'
sed -n '\%^/home/alice/documents/%p'
sed -n '\;^/home/alice/documents/;p'
```

/regexp/`I`
\%regexp%`I`

The `I` modifier to regular-expression matching is a GNU extension which causes the *regexp* to be matched in a case-insensitive manner.

---

[5] There are of course many other ways to do the same, e.g.

```
grep 'bash$' /etc/passwd
awk -F: '$7 == "/bin/bash"' /etc/passwd
```

In many other programming languages, a lower case `i` is used for case-insensitive regular expression matching. However, in `sed` the `i` is used for the insert command (see [insert command], page 13).

Observe the difference between the following examples.

In this example, `/b/I` is the address: regular expression with `I` modifier. `d` is the delete command:

```
$ printf "%s\n" a b c | sed '/b/Id'
a
c
```

Here, `/b/` is the address: a regular expression. `i` is the insert command. `d` is the value to insert. A line with '`d`' is then inserted above the matched line:

```
$ printf "%s\n" a b c | sed '/b/id'
a
d
b
c
```

/regexp/`M`
`\%`*regexp*`%M`

The `M` modifier to regular-expression matching is a GNU `sed` extension which directs GNU `sed` to match the regular expression in *multi-line* mode. The modifier causes `^` and `$` to match respectively (in addition to the normal behavior) the empty string after a newline, and the empty string before a newline. There are special character sequences (`\‘` and `\’`) which always match the beginning or the end of the buffer. In addition, the period character does not match a new-line character in multi-line mode.

Regex addresses operate on the content of the current pattern space. If the pattern space is changed (for example with `s///` command) the regular expression matching will operate on the changed text.

In the following example, automatic printing is disabled with `-n`. The `s/2/X/` command changes lines containing '`2`' to '`X`'. The command `/[0-9]/p` matches lines with digits and prints them. Because the second line is changed before the `/[0-9]/` regex, it will not match and will not be printed:

```
$ seq 3 | sed -n 's/2/X/ ; /[0-9]/p'
1
3
```

4.4 Range Addresses

An address range can be specified by specifying two addresses separated by a comma (`,`). An address range matches lines starting from where the first address matches, and continues until the second address matches (inclusively):

```
$ seq 10 | sed -n '4,6p'
4
5
6
```

If the second address is a *regexp*, then checking for the ending match will start with the line *following* the line which matched the first address: a range will always span at least two lines (except of course if the input stream ends).

```
$ seq 10 | sed -n '4,/[0-9]/p'
4
5
```

If the second address is a *number* less than (or equal to) the line matching the first address, then only the one line is matched:

```
$ seq 10 | sed -n '4,1p'
4
```

GNU `sed` also supports some special two-address forms; all these are GNU extensions:

`0,/regexp/`
: A line number of `0` can be used in an address specification like `0,/regexp/` so that `sed` will try to match *regexp* in the first input line too. In other words, `0,/regexp/` is similar to `1,/regexp/`, except that if *addr2* matches the very first line of input the `0,/regexp/` form will consider it to end the range, whereas the `1,/regexp/` form will match the beginning of its range and hence make the range span up to the *second* occurrence of the regular expression.

  Note that this is the only place where the `0` address makes sense; there is no 0-th line and commands which are given the `0` address in any other way will give an error.

  The following examples demonstrate the difference between starting with address 1 and 0:

```
$ seq 10 | sed -n '1,/[0-9]/p'
1
2

$ seq 10 | sed -n '0,/[0-9]/p'
1
```

`addr1,+N`
: Matches *addr1* and the *N* lines following *addr1*.

```
$ seq 10 | sed -n '6,+2p'
6
7
8
```

  addr1 can be a line number or a regular expression.

`addr1,~N`
: Matches *addr1* and the lines following *addr1* until the next line whose input line number is a multiple of *N*. The following command prints starting at line 6, until the next line which is a multiple of 4 (i.e. line 8):

```
$ seq 10 | sed -n '6,~4p'
6
7
8
```

  addr1 can be a line number or a regular expression.

5 Regular Expressions: selecting text

5.1 Overview of regular expression in `sed`

To know how to use `sed`, people should understand regular expressions (*regexp* for short). A regular expression is a pattern that is matched against a subject string from left to right. Most characters are *ordinary*: they stand for themselves in a pattern, and match the corresponding characters. Regular expressions in `sed` are specified between two slashes.

The following command prints lines containing the word '`hello`':

```
sed -n '/hello/p'
```

The above example is equivalent to this `grep` command:

```
grep 'hello'
```

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *special characters*, which do not stand for themselves but instead are interpreted in some special way.

The character `^` (caret) in a regular expression matches the beginning of the line. The character `.` (dot) matches any single character. The following `sed` command matches and prints lines which start with the letter '`b`', followed by any single character, followed by the letter '`d`':

```
$ printf "%s\n" abode bad bed bit bid byte body | sed -n '/^b.d/p'
bad
bed
bid
body
```

The following sections explain the meaning and usage of special characters in regular expressions.

5.2 Basic (BRE) and extended (ERE) regular expression

Basic and extended regular expressions are two variations on the syntax of the specified pattern. Basic Regular Expression (BRE) syntax is the default in `sed` (and similarly in `grep`). Use the POSIX-specified `-E` option (`-r`, `--regexp-extended`) to enable Extended Regular Expression (ERE) syntax.

In GNU `sed`, the only difference between basic and extended regular expressions is in the behavior of a few special characters: '`?`', '`+`', parentheses, braces ('`{}`'), and '`|`'.

With basic (BRE) syntax, these characters do not have special meaning unless prefixed backslash ('\'); While with extended (ERE) syntax it is reversed: these characters are special unless they are prefixed with backslash ('\').

Desired pattern	**Basic (BRE) Syntax**	**Extended (ERE) Syntax**

literal '`+`' (plus sign)	`$ echo 'a+b=c' > foo`
	`$ sed -n '/a+b/p' foo`
	`a+b=c`
		`$ echo 'a+b=c' > foo`
		`$ sed -E -n '/a\+b/p' foo`
		`a+b=c`

One or more 'a' characters followed by 'b' (plus sign as special metacharacter)	`$ echo aab > foo`
	`$ sed -n '/a\+b/p' foo`
	`aab`
		`$ echo aab > foo`
		`$ sed -E -n '/a+b/p' foo`
		`aab`

5.3 Overview of basic regular expression syntax

Here is a brief description of regular expression syntax as used in `sed`.

`char` A single ordinary character matches itself.

`*` Matches a sequence of zero or more instances of matches for the preceding regular expression, which must be an ordinary character, a special character preceded by \, a `.`, a grouped regexp (see below), or a bracket expression. As a GNU extension, a postfixed regular expression can also be followed by `*`; for example, `a**` is equivalent to `a*`. POSIX 1003.1-2001 says that `*` stands for itself when it appears at the start of a regular expression or subexpression, but many nonGNU implementations do not support this and portable scripts should instead use `\*` in these contexts.

`.` Matches any character, including newline.

`^` Matches the null string at beginning of the pattern space, i.e. what appears after the circumflex must appear at the beginning of the pattern space.

In most scripts, pattern space is initialized to the content of each line (see Section 6.1 [How `sed` works], page 37). So, it is a useful simplification to think of `^#include` as matching only lines where '`#include`' is the first thing on line—if there are spaces before, for example, the match fails. This simplification is valid as long as the original content of pattern space is not modified, for example with an `s` command.

`^` acts as a special character only at the beginning of the regular expression or subexpression (that is, after `\(` or `\|`). Portable scripts should avoid `^` at the beginning of a subexpression, though, as POSIX allows implementations that treat `^` as an ordinary character in that context.

`$` It is the same as `^`, but refers to end of pattern space. `$` also acts as a special character only at the end of the regular expression or subexpression (that is, before `\)` or `\|`), and its use at the end of a subexpression is not portable.

`[`*`list`*`]`
`[^`*`list`*`]` Matches any single character in *list*: for example, `[aeiou]` matches all vowels. A list may include sequences like *`char1-char2`*, which matches any character between (inclusive) *char1* and *char2*. See Section 5.5 [Character Classes and Bracket Expressions], page 30.

`\+` As `*`, but matches one or more. It is a GNU extension.

`\?` As `*`, but only matches zero or one. It is a GNU extension.

`\{`*`i`*`\}` As `*`, but matches exactly *i* sequences (*i* is a decimal integer; for portability, keep it between 0 and 255 inclusive).

`\{`*i*`,`*j*`\}` Matches between *i* and *j*, inclusive, sequences.

`\{`*i*`,\}` Matches more than or equal to *i* sequences.

`\(`*regexp*`\)`
Groups the inner *regexp* as a whole, this is used to:

- Apply postfix operators, like `\(abcd\)*`: this will search for zero or more whole sequences of '`abcd`', while `abcd*` would search for '`abc`' followed by zero or more occurrences of '`d`'. Note that support for `\(abcd\)*` is required by POSIX 1003.1-2001, but many non-GNU implementations do not support it and hence it is not universally portable.
- Use back references (see below).

regexp1`\|`*regexp2*
Matches either *regexp1* or *regexp2*. Use parentheses to use complex alternative regular expressions. The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. It is a GNU extension.

regexp1regexp2
Matches the concatenation of *regexp1* and *regexp2*. Concatenation binds more tightly than `\|`, `^`, and `$`, but less tightly than the other regular expression operators.

`\`*digit* Matches the *digit*-th `\(...\)` parenthesized subexpression in the regular expression. This is called a *back reference*. Subexpressions are implicitly numbered by counting occurrences of `\(` left-to-right.

`\n` Matches the newline character.

`\`*char* Matches *char*, where *char* is one of `$`, `*`, `.`, `[`, `\`, or `^`. Note that the only C-like backslash sequences that you can portably assume to be interpreted are `\n` and `\\`; in particular `\t` is not portable, and matches a '`t`' under most implementations of `sed`, rather than a tab character.

Note that the regular expression matcher is greedy, i.e., matches are attempted from left to right and, if two or more matches are possible starting at the same character, it selects the longest.

Examples:

'`abcdef`' Matches '`abcdef`'.

'`a*b`' Matches zero or more '`a`'s followed by a single '`b`'. For example, '`b`' or '`aaaaab`'.

'`a\?b`' Matches '`b`' or '`ab`'.

'`a\+b\+`' Matches one or more '`a`'s followed by one or more '`b`'s: '`ab`' is the shortest possible match, but other examples are '`aaaab`' or '`abbbbb`' or '`aaaaaabbbbbb`'.

'`.*`'
'`.\+`' These two both match all the characters in a string; however, the first matches every string (including the empty string), while the second matches only strings containing at least one character.

‘`^main.*(.*)`’
: This matches a string starting with ‘`main`’, followed by an opening and closing parenthesis. The ‘`n`’, ‘`(`’ and ‘`)`’ need not be adjacent.

‘`^#`’
: This matches a string beginning with ‘`#`’.

‘`\\$`’
: This matches a string ending with a single backslash. The regexp contains two backslashes for escaping.

‘`\$`’
: Instead, this matches a string consisting of a single dollar sign, because it is escaped.

‘`[a-zA-Z0-9]`’
: In the C locale, this matches any ASCII letters or digits.

‘`[^ tab]\+`’
: (Here *`tab`* stands for a single tab character.) This matches a string of one or more characters, none of which is a space or a tab. Usually this means a word.

‘`^\(.*\)\n\1$`’
: This matches a string consisting of two equal substrings separated by a newline.

‘`.\{9\}A$`’
: This matches nine characters followed by an ‘`A`’ at the end of a line.

‘`^.\{15\}A`’
: This matches the start of a string that contains 16 characters, the last of which is an ‘`A`’.

5.4 Overview of extended regular expression syntax

The only difference between basic and extended regular expressions is in the behavior of a few characters: ‘`?`’, ‘`+`’, parentheses, braces (‘`{}`’), and ‘`|`’. While basic regular expressions require these to be escaped if you want them to behave as special characters, when using extended regular expressions you must escape them if you want them *to match a literal character*. ‘`|`’ is special here because ‘`\|`’ is a GNU extension – standard basic regular expressions do not provide its functionality.

Examples:

`abc?`
: becomes ‘`abc\?`’ when using extended regular expressions. It matches the literal string ‘`abc?`’.

`c\+`
: becomes ‘`c+`’ when using extended regular expressions. It matches one or more ‘`c`’s.

`a\{3,\}`
: becomes ‘`a{3,}`’ when using extended regular expressions. It matches three or more ‘`a`’s.

`\(abc\)\{2,3\}`
: becomes ‘`(abc){2,3}`’ when using extended regular expressions. It matches either ‘`abcabc`’ or ‘`abcabcabc`’.

`\(abc*\)\1`
: becomes ‘`(abc*)\1`’ when using extended regular expressions. Backreferences must still be escaped when using extended regular expressions.

`a\|b` becomes '`a|b`' when using extended regular expressions. It matches '`a`' or '`b`'.

5.5 Character Classes and Bracket Expressions

A *bracket expression* is a list of characters enclosed by '`[`' and '`]`'. It matches any single character in that list; if the first character of the list is the caret '`^`', then it matches any character **not** in the list. For example, the following command replaces the words '`gray`' or '`grey`' with '`blue`':

```
sed  's/gr[ae]y/blue/'
```

Bracket expressions can be used in both Section 5.3 [basic], page 27, and Section 5.4 [extended], page 29, regular expressions (that is, with or without the `-E`/`-r` options).

Within a bracket expression, a *range expression* consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive. In the default C locale, the sorting sequence is the native character order; for example, '`[a-d]`' is equivalent to '`[abcd]`'.

Finally, certain named classes of characters are predefined within bracket expressions, as follows.

These named classes must be used *inside* brackets themselves. Correct usage:

```
$ echo 1 | sed 's/[[:digit:]]/X/'
X
```

Incorrect usage is rejected by newer `sed` versions. Older versions accepted it but treated it as a single bracket expression (which is equivalent to '`[dgit:]`', that is, only the characters *d/g/i/t/:*):

```
# current GNU sed versions - incorrect usage rejected
$ echo 1 | sed 's/[:digit:]/X/'
sed: character class syntax is [[:space:]], not [:space:]

# older GNU sed versions
$ echo 1 | sed 's/[:digit:]/X/'
1
```

'`[:alnum:]`'
: Alphanumeric characters: '`[:alpha:]`' and '`[:digit:]`'; in the '`C`' locale and ASCII character encoding, this is the same as '`[0-9A-Za-z]`'.

'`[:alpha:]`'
: Alphabetic characters: '`[:lower:]`' and '`[:upper:]`'; in the '`C`' locale and ASCII character encoding, this is the same as '`[A-Za-z]`'.

'`[:blank:]`'
: Blank characters: space and tab.

'`[:cntrl:]`'
: Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL). In other character sets, these are the equivalent characters, if any.

'`[:digit:]`'
: Digits: `0 1 2 3 4 5 6 7 8 9`.

‘`[:graph:]`’
: Graphical characters: ‘`[:alnum:]`’ and ‘`[:punct:]`’.

‘`[:lower:]`’
: Lower-case letters; in the ‘`C`’ locale and ASCII character encoding, this is `a b c d e f g h i j k l m n o p q r s t u v w x y z`.

‘`[:print:]`’
: Printable characters: ‘`[:alnum:]`’, ‘`[:punct:]`’, and space.

‘`[:punct:]`’
: Punctuation characters; in the ‘`C`’ locale and ASCII character encoding, this is `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ‘ { | } ~`.

‘`[:space:]`’
: Space characters: in the ‘`C`’ locale, this is tab, newline, vertical tab, form feed, carriage return, and space.

‘`[:upper:]`’
: Upper-case letters: in the ‘`C`’ locale and ASCII character encoding, this is `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`.

‘`[:xdigit:]`’
: Hexadecimal digits: `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`.

Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket expression.

Most meta-characters lose their special meaning inside bracket expressions:

‘`]`’
: ends the bracket expression if it's not the first list item. So, if you want to make the ‘`]`’ character a list item, you must put it first.

‘`-`’
: represents the range if it's not first or last in a list or the ending point of a range.

‘`^`’
: represents the characters not in the list. If you want to make the ‘`^`’ character a list item, place it anywhere but first.

TODO: incorporate this paragraph (copied verbatim from BRE section).

The characters `$`, `*`, `.`, `[`, and `\` are normally not special within *list*. For example, `[\*]` matches either ‘`\`’ or ‘`*`’, because the `\` is not special here. However, strings like `[.ch.]`, `[=a=]`, and `[:space:]` are special within *list* and represent collating symbols, equivalence classes, and character classes, respectively, and `[` is therefore special within *list* when it is followed by `.`, `=`, or `:`. Also, when not in `POSIXLY_CORRECT` mode, special escapes like `\n` and `\t` are recognized within *list*. See Section 5.8 [Escapes], page 34.

‘`[.`’
: represents the open collating symbol.

‘`.]`’
: represents the close collating symbol.

‘`[=`’
: represents the open equivalence class.

‘`=]`’
: represents the close equivalence class.

‘`[:`’
: represents the open character class symbol, and should be followed by a valid character class name.

‘`:]`’
: represents the close character class symbol.

5.6 regular expression extensions

The following sequences have special meaning inside regular expressions (used in Section 4.3 [addresses], page 23, and the `s` command).

These can be used in both Section 5.3 [basic], page 27, and Section 5.4 [extended], page 29, regular expressions (that is, with or without the `-E`/`-r` options).

`\w` Matches any "word" character. A "word" character is any letter or digit or the underscore character.

```
$ echo "abc %-= def." | sed 's/\w/X/g'
XXX %-= XXX.
```

`\W` Matches any "non-word" character.

```
$ echo "abc %-= def." | sed 's/\W/X/g'
abcXXXXXdefX
```

`\b` Matches a word boundary; that is it matches if the character to the left is a "word" character and the character to the right is a "non-word" character, or vice-versa.

```
$ echo "abc %-= def." | sed 's/\b/X/g'
XabcX %-= XdefX.
```

`\B` Matches everywhere but on a word boundary; that is it matches if the character to the left and the character to the right are either both "word" characters or both "non-word" characters.

```
$ echo "abc %-= def." | sed 's/\B/X/g'
aXbXc X%X-X=X dXeXf.X
```

`\s` Matches whitespace characters (spaces and tabs). Newlines embedded in the pattern/hold spaces will also match:

```
$ echo "abc %-= def." | sed 's/\s/X/g'
abcX%-=Xdef.
```

`\S` Matches non-whitespace characters.

```
$ echo "abc %-= def." | sed 's/\S/X/g'
XXX XXX XXXX
```

`\<` Matches the beginning of a word.

```
$ echo "abc %-= def." | sed 's/\</X/g'
Xabc %-= Xdef.
```

`\>` Matches the end of a word.

```
$ echo "abc %-= def." | sed 's/\>/X/g'
abcX %-= defX.
```

`\‘` Matches only at the start of pattern space. This is different from `^` in multi-line mode.

Compare the following two examples:

```
$ printf "a\nb\nc\n" | sed 'N;N;s/^/X/gm'
Xa
```

```
Xb
Xc

$ printf "a\nb\nc\n" | sed 'N;N;s/\`/X/gm'
Xa
b
c
```

`\'` Matches only at the end of pattern space. This is different from `$` in multi-line mode.

5.7 Back-references and Subexpressions

back-references are regular expression commands which refer to a previous part of the matched regular expression. Back-references are specified with backslash and a single digit (e.g. '`\1`'). The part of the regular expression they refer to is called a *subexpression*, and is designated with parentheses.

Back-references and subexpressions are used in two cases: in the regular expression search pattern, and in the *replacement* part of the `s` command (see Section 4.3 [Regular Expression Addresses], page 23, and Section 3.3 [The "s" Command], page 8).

In a regular expression pattern, back-references are used to match the same content as a previously matched subexpression. In the following example, the subexpression is '`.`' - any single character (being surrounded by parentheses makes it a subexpression). The back-reference '`\1`' asks to match the same content (same character) as the sub-expression.

The command below matches words starting with any character, followed by the letter '`o`', followed by the same character as the first.

```
$ sed -E -n '/^(.)o\1$/p' /usr/share/dict/words
bob
mom
non
pop
sos
tot
wow
```

Multiple subexpressions are automatically numbered from left-to-right. This command searches for 6-letter palindromes (the first three letters are 3 subexpressions, followed by 3 back-references in reverse order):

```
$ sed -E -n '/^(.)(.)(.)\3\2\1$/p' /usr/share/dict/words
redder
```

In the `s` command, back-references can be used in the *replacement* part to refer back to subexpressions in the *regexp* part.

The following example uses two subexpressions in the regular expression to match two space-separated words. The back-references in the *replacement* part prints the words in a different order:

```
$ echo "James Bond" | sed -E 's/(.*) (.*)/The name is \2, \1 \2./'
The name is Bond, James Bond.
```

When used with alternation, if the group does not participate in the match then the back-reference makes the whole match fail. For example, '`a(.)|b\1`' will not match '`ba`'. When multiple regular expressions are given with `-e` or from a file ('`-f file`'), back-references are local to each expression.

5.8 Escape Sequences - specifying special characters

Until this chapter, we have only encountered escapes of the form '`\^`', which tell `sed` not to interpret the circumflex as a special character, but rather to take it literally. For example, '`\*`' matches a single asterisk rather than zero or more backslashes.

This chapter introduces another kind of escape[6]—that is, escapes that are applied to a character or sequence of characters that ordinarily are taken literally, and that `sed` replaces with a special character. This provides a way of encoding non-printable characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters in a `sed` script but when a script is being prepared in the shell or by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

The list of these escapes is:

`\a` Produces or matches a BEL character, that is an "alert" (ASCII 7).

`\f` Produces or matches a form feed (ASCII 12).

`\n` Produces or matches a newline (ASCII 10).

`\r` Produces or matches a carriage return (ASCII 13).

`\t` Produces or matches a horizontal tab (ASCII 9).

`\v` Produces or matches a so called "vertical tab" (ASCII 11).

`\c`*x* Produces or matches CONTROL-*x*, where *x* is any character. The precise effect of '`\c`*x*' is as follows: if *x* is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus '`\cz`' becomes hex 1A, but '`\c{`' becomes hex 3B, while '`\c;`' becomes hex 7B.

`\d`*xxx* Produces or matches a character whose decimal ASCII value is *xxx*.

`\o`*xxx* Produces or matches a character whose octal ASCII value is *xxx*.

`\x`*xx* Produces or matches a character whose hexadecimal ASCII value is *xx*.

'`\b`' (backspace) was omitted because of the conflict with the existing "word boundary" meaning.

5.8.1 Escaping Precedence

GNU `sed` processes escape sequences *before* passing the text onto the regular-expression matching of the `s///` command and Address matching. Thus the follwing two commands are equivalent ('`0x5e`' is the hexadecimal ASCII value of the character '`^`'):

[6] All the escapes introduced here are GNU extensions, with the exception of `\n`. In basic regular expression mode, setting `POSIXLY_CORRECT` disables them inside bracket expressions.

```
$ echo 'a^c' | sed 's/^/b/'
ba^c

$ echo 'a^c' | sed 's/\x5e/b/'
ba^c
```

As are the following ('0x5b','0x5d' are the hexadecimal ASCII values of '[',']', respectively):

```
$ echo abc | sed 's/[a]/x/'
Xbc
$ echo abc | sed 's/\x5ba\x5d/x/'
Xbc
```

However it is recommended to avoid such special characters due to unexpected edge-cases. For example, the following are not equivalent:

```
$ echo 'a^c' | sed 's/\^/b/'
abc

$ echo 'a^c' | sed 's/\\\x5e/b/'
a^c
```

5.9 Multibyte characters and Locale Considerations

GNU `sed` processes valid multibyte characters in multibyte locales (e.g. `UTF-8`).[7]

The following example uses the Greek letter Capital Sigma (Σ, Unicode code point `0x03A3`). In a `UTF-8` locale, `sed` correctly processes the Sigma as one character despite it being 2 octets (bytes):

```
$ locale | grep LANG
LANG=en_US.UTF-8

$ printf 'a\u03A3b'
aΣb

$ printf 'a\u03A3b' | sed 's/./X/g'
XXX

$ printf 'a\u03A3b' | od -tx1 -An
 61 ce a3 62
```

To force `sed` to process octets separately, use the `C` locale (also known as the `POSIX` locale):

```
$ printf 'a\u03A3b' | LC_ALL=C sed 's/./X/g'
XXXX
```

5.9.1 Invalid multibyte characters

`sed`'s regular expressions *do not* match invalid multibyte sequences in a multibyte locale.

[7] Some regexp edge-cases depends on the operating system and libc implementation. The examples shown are known to work as-expected on GNU/Linux systems using glibc.

In the following examples, the ascii value `0xCE` is an incomplete multibyte character (shown here as ⊗). The regular expression '`.`' does not match it:

```
$ printf 'a\xCEb\n'
a⊗e

$ printf 'a\xCEb\n' | sed 's/./X/g'
X⊗X

$ printf 'a\xCEc\n' | sed 's/./X/g' | od -tx1c -An
  58  ce  58  0a
   X       X   \n
```

Similarly, the 'catch-all' regular expression '`.*`' does not match the entire line:

```
$ printf 'a\xCEc\n' | sed 's/.*//' | od -tx1c -An
  ce  63  0a
       c  \n
```

GNU `sed` offers the special `z` command to clear the current pattern space regardless of invalid multibyte characters (i.e. it works like `s/.*//` but also removes invalid multibyte characters):

```
$ printf 'a\xCEc\n' | sed 'z' | od -tx1c -An
   0a
   \n
```

Alternatively, force the `C` locale to process each octet separately (every octet is a valid character in the `C` locale):

```
$ printf 'a\xCEc\n' | LC_ALL=C sed 's/.*//' | od -tx1c -An
  0a
  \n
```

`sed`'s inability to process invalid multibyte characters can be used to detect such invalid sequences in a file. In the following examples, the `\xCE\xCE` is an invalid multibyte sequence, while `\xCE\A3` is a valid multibyte sequence (of the Greek Sigma character).

The following `sed` program removes all valid characters using `s/.//g`. Any content left in the pattern space (the invalid characters) are added to the hold space using the `H` command. On the last line (`$`), the hold space is retrieved (`x`), newlines are removed (`s/\n//g`), and any remaining octets are printed unambiguously (`l`). Thus, any invalid multibyte sequences are printed as octal values:

```
$ printf 'ab\nc\n\xCE\xCEde\n\xCE\xA3f\n' > invalid.txt

$ cat invalid.txt
ab
c
⊗⊗de
Σf

$ sed -n 's/.//g ; H ; ${x;s/\n//g;l}' invalid.txt
\316\316$
```

With a few more commands, sed can print the exact line number corresponding to each invalid characters (line 3). These characters can then be removed by forcing the C locale and using octal escape sequences:

```
$ sed -n 's/.//g;=;l' invalid.txt | paste - -  | awk '$2!="$"'
3       \316\316$

$ LC_ALL=C sed '3s/\o316\o316//' invalid.txt > fixed.txt
```

5.9.2 Upper/Lower case conversion

GNU sed's substitute command (s) supports upper/lower case conversions using \U,\L codes. These conversions support multibyte characters:

```
$ printf 'ABC\u03a3\n'
ABCΣ

$ printf 'ABC\u03a3\n' | sed 's/.*/\L&/'
abcσ
```

See Section 3.3 [The "s" Command], page 8.

5.9.3 Multibyte regexp character classes

In other locales, the sorting sequence is not specified, and '[a-d]' might be equivalent to '[abcd]' or to '[aBbCcDd]', or it might fail to match any character, or the set of characters that it matches might even be erratic. To obtain the traditional interpretation of bracket expressions, you can use the 'C' locale by setting the LC_ALL environment variable to the value 'C'.

```
# TODO: is there any real-world system/locale where 'A'
#       is replaced by '-' ?
$ echo A | sed 's/[a-z]/-/'
A
```

Their interpretation depends on the LC_CTYPE locale; for example, '[[:alnum:]]' means the character class of numbers and letters in the current locale.

TODO: show example of collation

```
# TODO: this works on glibc systems, not on musl-libc/freebsd/macosx.
$ printf 'clich\n' | LC_ALL=fr_FR.utf8 sed 's/[[=e=]]/X/g'
clichX
```

6 Advanced sed: cycles and buffers

6.1 How sed Works

sed maintains two data buffers: the active *pattern* space, and the auxiliary *hold* space. Both are initially empty.

sed operates by performing the following cycle on each line of input: first, sed reads one line from the input stream, removes any trailing newline, and places it in the pattern space. Then commands are executed; each command can have an address associated to it:

addresses are a kind of condition code, and a command is only executed if the condition is verified before the command is to be executed.

When the end of the script is reached, unless the `-n` option is in use, the contents of pattern space are printed out to the output stream, adding back the trailing newline if it was removed.[8] Then the next cycle starts for the next input line.

Unless special commands (like '`D`') are used, the pattern space is deleted between two cycles. The hold space, on the other hand, keeps its data between cycles (see commands '`h`', '`H`', '`x`', '`g`', '`G`' to move data between both buffers).

6.2 Hold and Pattern Buffers

TODO

6.3 Multiline techniques - using D,G,H,N,P to process multiple lines

Multiple lines can be processed as one buffer using the `D`,`G`,`H`,`N`,`P`. They are similar to their lowercase counterparts (`d`,`g`, `h`,`n`,`p`), except that these commands append or subtract data while respecting embedded newlines - allowing adding and removing lines from the pattern and hold spaces.

They operate as follows:

`D` *deletes* line from the pattern space until the first newline, and restarts the cycle.

`G` *appends* line from the hold space to the pattern space, with a newline before it.

`H` *appends* line from the pattern space to the hold space, with a newline before it.

`N` *appends* line from the input file to the pattern space.

`P` *prints* line from the pattern space until the first newline.

The following example illustrates the operation of `N` and `D` commands:

```
$ seq 6 | sed -n 'N;l;D'
1\n2$
2\n3$
3\n4$
4\n5$
5\n6$
```

1. `sed` starts by reading the first line into the pattern space (i.e. '`1`').
2. At the beginning of every cycle, the `N` command appends a newline and the next line to the pattern space (i.e. '`1`', '`\n`', '`2`' in the first cycle).
3. The `l` command prints the content of the pattern space unambiguously.
4. The `D` command then removes the content of pattern space up to the first newline (leaving '`2`' at the end of the first cycle).

[8] Actually, if `sed` prints a line without the terminating newline, it will nevertheless print the missing newline as soon as more text is sent to the same output stream, which gives the "least expected surprise" even though it does not make commands like '`sed -n p`' exactly identical to `cat`.

5. At the next cycle the `N` command appends a newline and the next input line to the pattern space (e.g. '`2`', '`\n`', '`3`').

A common technique to process blocks of text such as paragraphs (instead of line-by-line) is using the following construct:

```
sed '/./{H;$!d} ; x ; s/REGEXP/REPLACEMENT/'
```

1. The first expression, `/./{H;$!d}` operates on all non-empty lines, and adds the current line (in the pattern space) to the hold space. On all lines except the last, the pattern space is deleted and the cycle is restarted.
2. The other expressions `x` and `s` are executed only on empty lines (i.e. paragraph separators). The `x` command fetches the accumulated lines from the hold space back to the pattern space. The `s///` command then operates on all the text in the paragraph (including the embedded newlines).

The following example demonstrates this technique:

```
$ cat input.txt
a a a aa aaa
aaaa aaaa aa
aaaa aaa aaa

bbbb bbb bbb
bb bb bbb bb
bbbbbbbb bbb

ccc ccc cccc
cccc ccccc c
cc cc cc cc

$ sed '/./{H;$!d} ; x ; s/^/\nSTART-->/ ; s/$/\n<--END/' input.txt

START-->
a a a aa aaa
aaaa aaaa aa
aaaa aaa aaa
<--END

START-->
bbbb bbb bbb
bb bb bbb bb
bbbbbbbb bbb
<--END

START-->
ccc ccc cccc
cccc ccccc c
cc cc cc cc
<--END
```

For more annotated examples, see Section 7.7 [Text search across multiple lines], page 50, and Section 7.8 [Line length adjustment], page 51.

6.4 Branching and Flow Control

The branching commands `b`, `t`, and `T` enable changing the flow of `sed` programs.

By default, `sed` reads an input line into the pattern buffer, then continues to processes all commands in order. Commands without addresses affect all lines. Commands with addresses affect only matching lines. See Section 6.1 [Execution Cycle], page 37, and Section 4.1 [Addresses overview], page 22.

`sed` does not support a typical `if/then` construct. Instead, some commands can be used as conditionals or to change the default flow control:

`d` delete (clears) the current pattern space, and restart the program cycle without processing the rest of the commands and without printing the pattern space.

`D` delete the contents of the pattern space *up to the first newline*, and restart the program cycle without processing the rest of the commands and without printing the pattern space.

```
[addr]X
[addr]{ X ; X ; X }
/regexp/X
/regexp/{ X ; X ; X }
```

Addresses and regular expressions can be used as an `if/then` conditional: If *[addr]* matches the current pattern space, execute the command(s). For example: The command `/^#/d` means: *if* the current pattern matches the regular expression `^#` (a line starting with a hash), *then* execute the `d` command: delete the line without printing it, and restart the program cycle immediately.

`b` branch unconditionally (that is: always jump to a label, skipping or repeating other commands, without restarting a new cycle). Combined with an address, the branch can be conditionally executed on matched lines.

`t` branch conditionally (that is: jump to a label) *only if* a `s///` command has succeeded since the last input line was read or another conditional branch was taken.

`T` similar but opposite to the `t` command: branch only if there has been *no* successful substitutions since the last input line was read.

The following two `sed` programs are equivalent. The first (contrived) example uses the `b` command to skip the `s///` command on lines containing '`1`'. The second example uses an address with negation ('`!`') to perform substitution only on desired lines. The `y///` command is still executed on all lines:

```
$ printf '%s\n' a1 a2 a3 | sed -E '/1/bx ; s/a/z/ ; :x ; y/123/456/'
a4
z5
z6

$ printf '%s\n' a1 a2 a3 | sed -E '/1/!s/a/z/ ; y/123/456/'
a4
z5
z6
```

6.4.1 Branching and Cycles

The `b`,`t` and `T` commands can be followed by a label (typically a single letter). Labels are defined with a colon followed by one or more letters (e.g. '`:x`'). If the label is omitted the branch commands restart the cycle. Note the difference between branching to a label and restarting the cycle: when a cycle is restarted, `sed` first prints the current content of the pattern space, then reads the next input line into the pattern space; Jumping to a label (even if it is at the beginning of the program) does not print the pattern space and does not read the next input line.

The following program is a no-op. The `b` command (the only command in the program) does not have a label, and thus simply restarts the cycle. On each cycle, the pattern space is printed and the next input line is read:

```
$ seq 3 | sed b
1
2
3
```

The following example is an infinite-loop - it doesn't terminate and doesn't print anything. The `b` command jumps to the '`x`' label, and a new cycle is never started:

```
$ seq 3 | sed ':x ; bx'

# The above command requires gnu sed (which supports additional
# commands following a label, without a newline). A portable equivalent:
#     sed -e ':x' -e bx
```

Branching is often complemented with the `n` or `N` commands: both commands read the next input line into the pattern space without waiting for the cycle to restart. Before reading the next input line, `n` prints the current pattern space then empties it, while `N` appends a newline and the next input line to the pattern space.

Consider the following two examples:

```
$ seq 3 | sed ':x ; n ; bx'
1
2
3

$ seq 3 | sed ':x ; N ; bx'
1
2
3
```

- Both examples do not inf-loop, despite never starting a new cycle.
- In the first example, the `n` commands first prints the content of the pattern space, empties the pattern space then reads the next input line.
- In the second example, the `N` commands appends the next input line to the pattern space (with a newline). Lines are accumulated in the pattern space until there are no more input lines to read, then the `N` command terminates the `sed` program. When the program terminates, the end-of-cycle actions are performed, and the entire pattern space is printed.
- The second example requires GNU `sed`, because it uses the non-POSIX-standard behavior of `N`. See the "`N` command on the last line" paragraph in Chapter 10 [Reporting Bugs], page 63.
- To further examine the difference between the two examples, try the following commands:

```
printf '%s\n' aa bb cc dd | sed ':x ; n ; = ; bx'
printf '%s\n' aa bb cc dd | sed ':x ; N ; = ; bx'
printf '%s\n' aa bb cc dd | sed ':x ; n ; s/\n/***/ ; bx'
printf '%s\n' aa bb cc dd | sed ':x ; N ; s/\n/***/ ; bx'
```

6.4.2 Branching example: joining lines

As a real-world example of using branching, consider the case of quoted-printable (`https://en.wikipedia.org/wiki/Quoted-printable`) files, typically used to encode email messages. In these files long lines are split and marked with a *soft line break* consisting of a single '`=`' character at the end of the line:

```
$ cat jaques.txt
All the wor=
ld's a stag=
e,
And all the=
 men and wo=
men merely =
players:
They have t=
heir exits =
and their e=
ntrances;
And one man=
 in his tim=
e plays man=
y parts.
```

The following program uses an address match '`/=$/`' as a conditional: If the current pattern space ends with a '`=`', it reads the next input line using `N`, replaces all '`=`' characters which are followed by a newline, and unconditionally branches (`b`) to the beginning of the program without restarting a new cycle. If the pattern space does not ends with '`=`', the default action is performed: the pattern space is printed and a new cycle is started:

```
$ sed ':x ; /=$/ { N ; s/=\n//g ; bx }' jaques.txt
All the world's a stage,
And all the men and women merely players:
They have their exits and their entrances;
And one man in his time plays many parts.
```

Here's an alternative program with a slightly different approach: On all lines except the last, `N` appends the line to the pattern space. A substitution command then removes soft line breaks ('`=`' at the end of a line, i.e. followed by a newline) by replacing them with an empty string. *if* the substitution was successful (meaning the pattern space contained a line which should be joined), The conditional branch command `t` jumps to the beginning of the program without completing or restarting the cycle. If the substitution failed (meaning there were no soft line breaks), The `t` command will *not* branch. Then, `P` will print the pattern space content until the first newline, and `D` will delete the pattern space content until the first new line. (To learn more about `N`, `P` and `D` commands see Section 6.3 [Multiline techniques], page 38).

```
$ sed ':x ; $!N ; s/=\n// ; tx ; P ; D' jaques.txt
All the world's a stage,
And all the men and women merely players:
They have their exits and their entrances;
And one man in his time plays many parts.
```

For more line-joining examples see Section 7.1 [Joining lines], page 43.

7 Some Sample Scripts

Here are some `sed` scripts to guide you in the art of mastering `sed`.

7.1 Joining lines

This section uses `N`, `D` and `P` commands to process multiple lines, and the `b` and `t` commands for branching. See Section 6.3 [Multiline techniques], page 38, and Section 6.4 [Branching and flow control], page 40.

Join specific lines (e.g. if lines 2 and 3 need to be joined):

```
$ cat lines.txt
hello
hel
lo
hello

$ sed '2{N;s/\n//;}' lines.txt
hello
hello
hello
```

Join backslash-continued lines:

```
$ cat 1.txt
this \
```

```
is \
a \
long \
line
and another \
line

$ sed -e ':x /\\$/ { N; s/\\\n//g ; bx }'  1.txt
this is a long line
and another line


#TODO: The above requires gnu sed.
#      non-gnu seds need newlines after ':' and 'b'
```

Join lines that start with whitespace (e.g SMTP headers):

```
$ cat 2.txt
Subject: Hello
    World
Content-Type: multipart/alternative;
    boundary=94eb2c190cc6370f06054535da6a
Date: Tue, 3 Jan 2017 19:41:16 +0000 (GMT)
Authentication-Results: mx.gnu.org;
       dkim=pass header.i=@gnu.org;
       spf=pass
Message-ID: <abcdef@gnu.org>
From: John Doe <jdoe@gnu.org>
To: Jane Smith <jsmith@gnu.org>

$ sed -E ':a ; $!N ; s/\n\s+/ / ; ta ; P ; D' 2.txt
Subject: Hello World
Content-Type: multipart/alternative; boundary=94eb2c190cc6370f06054535da6a
Date: Tue, 3 Jan 2017 19:41:16 +0000 (GMT)
Authentication-Results: mx.gnu.org; dkim=pass header.i=@gnu.org; spf=pass
Message-ID: <abcdef@gnu.org>
From: John Doe <jdoe@gnu.org>
To: Jane Smith <jsmith@gnu.org>

# A portable (non-gnu) variation:
#   sed -e :a -e '$!N;s/\n  */ /;ta' -e 'P;D'
```

7.2 Centering Lines

This script centers all lines of a file on a 80 columns width. To change that width, the number in `\{...\}` must be replaced, and the number of added spaces also must be changed.

Note how the buffer commands are used to separate parts in the regular expressions to be matched—this is a common technique.

```
#!/usr/bin/sed -f

# Put 80 spaces in the buffer
1 {
  x
  s/^$/          /
  s/^.*$/&&&&&&&&/
  x
}

# delete leading and trailing spaces
y/tab/ /
s/^ *//
s/ *$//

# add a newline and 80 spaces to end of line
G

# keep first 81 chars (80 + a newline)
s/^\(.\{81\}\).*$/\1/

# \2 matches half of the spaces, which are moved to the beginning
s/^\(.*\)\n\(.*\)\2/\2\1/
```

7.3 Increment a Number

This script is one of a few that demonstrate how to do arithmetic in `sed`. This is indeed possible,[9] but must be done manually.

To increment one number you just add 1 to last digit, replacing it by the following digit. There is one exception: when the digit is a nine the previous digits must be also incremented until you don't have a nine.

This solution by Bruno Haible is very clever and smart because it uses a single buffer; if you don't have this limitation, the algorithm used in Section 7.10 [cat -n], page 53, is faster. It works by replacing trailing nines with an underscore, then using multiple `s` commands to increment the last digit, and then again substituting underscores with zeros.

```
#!/usr/bin/sed -f

/[^0-9]/ d

# replace all trailing 9s by _ (any other character except digits, could
# be used)
:d
s/9\(_*\)$/_\1/
td
```

[9] `sed` guru Greg Ubben wrote an implementation of the `dc` RPN calculator! It is distributed together with sed.

```
# incr last digit only.  The first line adds a most-significant
# digit of 1 if we have to add a digit.

s/^\(_*\)$/1\1/; tn
s/8\(_*\)$/9\1/; tn
s/7\(_*\)$/8\1/; tn
s/6\(_*\)$/7\1/; tn
s/5\(_*\)$/6\1/; tn
s/4\(_*\)$/5\1/; tn
s/3\(_*\)$/4\1/; tn
s/2\(_*\)$/3\1/; tn
s/1\(_*\)$/2\1/; tn
s/0\(_*\)$/1\1/; tn

:n
y/_/0/
```

7.4 Rename Files to Lower Case

This is a pretty strange use of `sed`. We transform text, and transform it to be shell commands, then just feed them to shell. Don't worry, even worse hacks are done when using `sed`; I have seen a script converting the output of `date` into a `bc` program!

The main body of this is the `sed` script, which remaps the name from lower to upper (or vice-versa) and even checks out if the remapped name is the same as the original name. Note how the script is parameterized using shell variables and proper quoting.

```
#! /bin/sh
# rename files to lower/upper case...
#
# usage:
#    move-to-lower *
#    move-to-upper *
# or
#    move-to-lower -R .
#    move-to-upper -R .
#

help()
{
        cat << eof
Usage: $0 [-n] [-r] [-h] files...

-n      do nothing, only see what would be done
-R      recursive (use find)
-h      this message
files   files to remap to lower case
```

```
Examples:
       $0 -n *        (see if everything is ok, then...)
       $0 *

       $0 -R .

eof
}

apply_cmd='sh'
finder='echo "$@" | tr " " "\n"'
files_only=

while :
do
    case "$1" in
        -n) apply_cmd='cat' ;;
        -R) finder='find "$@" -type f';;
        -h) help ; exit 1 ;;
        *) break ;;
    esac
    shift
done

if [ -z "$1" ]; then
        echo Usage: $0 [-h] [-n] [-r] files...
        exit 1
fi

LOWER='abcdefghijklmnopqrstuvwxyz'
UPPER='ABCDEFGHIJKLMNOPQRSTUVWXYZ'

case `basename $0` in
        *upper*) TO=$UPPER; FROM=$LOWER ;;
        *)       FROM=$UPPER; TO=$LOWER ;;
esac

eval $finder | sed -n '

# remove all trailing slashes
s/\/*$//

# add ./ if there is no path, only a filename
/\//! s/^/.\//

# save path+filename
h
```

```
# remove path
s/.*\///

# do conversion only on filename
y/'$FROM'/'$TO'/

# now line contains original path+file, while
# hold space contains the new filename
x

# add converted file name to line, which now contains
# path/file-name\nconverted-file-name
G

# check if converted file name is equal to original file name,
# if it is, do not print anything
/^.*\/\(.*\)\n\1/b

# escape special characters for the shell
s/["$`\\]/\\&/g

# now, transform path/fromfile\n, into
# mv path/fromfile path/tofile and print it
s/^\(.*\/\)\(.*\)\n\(.*\)$/mv "\1\2" "\1\3"/p

' | $apply_cmd
```

7.5 Print `bash` Environment

This script strips the definition of the shell functions from the output of the `set` Bourne-shell command.

```
#!/bin/sh

set | sed -n '
:x


# if no occurrence of `=()' print and load next line
/=()/! { p; b; }
/ () $/! { p; b; }

# possible start of functions section
# save the line in case this is a var like FOO="() "
h
```

```
# if the next line has a brace, we quit because
# nothing comes after functions
n
/^{/ q

# print the old line
x; p

# work on the new line now
x; bx
'
```

7.6 Reverse Characters of Lines

This script can be used to reverse the position of characters in lines. The technique moves two characters at a time, hence it is faster than more intuitive implementations.

Note the `tx` command before the definition of the label. This is often needed to reset the flag that is tested by the `t` command.

Imaginative readers will find uses for this script. An example is reversing the output of `banner`.[10]

```
#!/usr/bin/sed -f

/../! b

# Reverse a line.  Begin embedding the line between two newlines
s/^.*$/\
&\
/

# Move first character at the end.  The regexp matches until
# there are zero or one characters between the markers
tx
:x
s/\(\n.\)\(.*\)\(.\n\)/\3\2\1/
tx

# Remove the newline markers
s/\n//g
```

[10] This requires another script to pad the output of banner; for example

```
#! /bin/sh

banner -w $1 $2 $3 $4 |
  sed -e :a -e '/^.\{0,'$1'\}$/ { s/$/ /; ba; }' |
  ~/sedscripts/reverseline.sed
```

7.7 Text search across multiple lines

This section uses `N` and `D` commands to search for consecutive words spanning multiple lines. See Section 6.3 [Multiline techniques], page 38.

These examples deal with finding doubled occurrences of words in a document.

Finding doubled words in a single line is easy using GNU `grep` and similarly with GNU `sed`:

```
$ cat two-cities-dup1.txt
It was the best of times,
it was the worst of times,
it was the the age of wisdom,
it was the age of foolishness,

$ grep -E '\b(\w+)\s+\1\b' two-cities-dup1.txt
it was the the age of wisdom,

$ grep -n -E '\b(\w+)\s+\1\b' two-cities-dup1.txt
3:it was the the age of wisdom,

$ sed -En '/\b(\w+)\s+\1\b/p' two-cities-dup1.txt
it was the the age of wisdom,

$ sed -En '/\b(\w+)\s+\1\b/{=;p}' two-cities-dup1.txt
3
it was the the age of wisdom,
```

- The regular expression '`\b\w+\s+`' searches for word-boundary ('`\b`'), followed by one-or-more word-characters ('`\w+`'), followed by whitespace ('`\s+`'). See Section 5.6 [regexp extensions], page 32.
- Adding parentheses around the '`(\w+)`' expression creates a subexpression. The regular expression pattern '`(PATTERN)\s+\1`' defines a subexpression (in the parentheses) followed by a back-reference, separated by whitespace. A successful match means the *PATTERN* was repeated twice in succession. See Section 5.7 [Back-references and Subexpressions], page 33.
- The word-boundery expression ('`\b`') at both ends ensures partial words are not matched (e.g. '`the then`' is not a desired match).
- The `-E` option enables extended regular expression syntax, alleviating the need to add backslashes before the parenthesis. See Section 5.4 [ERE syntax], page 29.

When the doubled word span two lines the above regular expression will not find them as `grep` and `sed` operate line-by-line.

By using `N` and `D` commands, `sed` can apply regular expressions on multiple lines (that is, multiple lines are stored in the pattern space, and the regular expression works on it):

```
$ cat two-cities-dup2.txt
It was the best of times, it was the
worst of times, it was the
the age of wisdom,
```

```
it was the age of foolishness,

$ sed -En '{N; /\b(\w+)\s+\1\b/{=;p} ; D}'  two-cities-dup2.txt
3
worst of times, it was the
the age of wisdom,
```

- The `N` command appends the next line to the pattern space (thus ensuring it contains two consecutive lines in every cycle).
- The regular expression uses '`\s+`' for word separator which matches both spaces and newlines.
- The regular expression matches, the entire pattern space is printed with `p`. No lines are printed by default due to the `-n` option.
- The `D` removes the first line from the pattern space (up until the first newline), readying it for the next cycle.

See the GNU `coreutils` manual for an alternative solution using `tr -s` and `uniq` at `https://gnu.org/s/coreutils/manual/html_node/Squeezing-and-deleting.html`.

7.8 Line length adjustment

This section uses `N` and `D` commands to search for consecutive words spanning multiple lines, and the `b` command for branching. See Section 6.3 [Multiline techniques], page 38, and Section 6.4 [Branching and flow control], page 40.

This (somewhat contrived) example deal with formatting and wrapping lines of text of the following input file:

```
$ cat two-cities-mix.txt
It was the best of times, it was
the worst of times, it
was the age of
wisdom,
it
was
the age
of foolishness,
```

The following sed program wraps lines at 40 characters:

```
$ cat wrap40.sed
# outer loop
:x

# Appead a newline followed by the next input line to the pattern buffer
N

# Remove all newlines from the pattern buffer
s/\n/ /g


# Inner loop
:y

# Add a newline after the first 40 characters
s/(.{40,40})/\1\n/

# If there is a newline in the pattern buffer
# (i.e. the previous substitution added a newline)
/\n/ {
    # There are newlines in the pattern buffer -
    # print the content until the first newline.
    P

   # Remove the printed characters and the first newline
   s/.*\n//

   # branch to label 'y' - repeat inner loop
   by
 }

# No newlines in the pattern buffer - Branch to label 'x' (outer loop)
# and read the next input line
bx
```

The wrapped output:

```
$ sed -E -f wrap40.sed two-cities-mix.txt
It was the best of times, it was the wor
st of times, it was the age of wisdom, i
t was the age of foolishness,
```

7.9 Reverse Lines of Files

This one begins a series of totally useless (yet interesting) scripts emulating various Unix commands. This, in particular, is a `tac` workalike.

Note that on implementations other than GNU `sed` this script might easily overflow internal buffers.

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

# reverse all lines of input, i.e. first line became last, ...

# from the second line, the buffer (which contains all previous lines)
# is *appended* to current line, so, the order will be reversed
1! G

# on the last line we're done -- print everything
$ p

# store everything on the buffer again
h
```

7.10 Numbering Lines

This script replaces '`cat -n`'; in fact it formats its output exactly like GNU `cat` does.

Of course this is completely useless and for two reasons: first, because somebody else did it in C, second, because the following Bourne-shell script could be used for the same purpose and would be much faster:

```
#! /bin/sh
sed -e "=" $@ | sed -e '
  s/^/      /
  N
  s/^ *\(......\)\n/\1  /
'
```

It uses `sed` to print the line number, then groups lines two by two using `N`. Of course, this script does not teach as much as the one presented below.

The algorithm used for incrementing uses both buffers, so the line is printed as soon as possible and then discarded. The number is split so that changing digits go in a buffer and unchanged ones go in the other; the changed digits are modified in a single step (using a `y` command). The line number for the next line is then composed and stored in the hold space, to be used in the next iteration.

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

# Prime the pump on the first line
x
/^$/ s/^.*$/1/

# Add the correct line number before the pattern
G
h
```

```
# Format it and print it
s/^/      /
s/^ *\(......\)\n/\1  /p

# Get the line number from hold space; add a zero
# if we're going to add a digit on the next line
g
s/\n.*$//
/^9*$/ s/^/0/

# separate changing/unchanged digits with an x
s/.9*$/x&/

# keep changing digits in hold space
h
s/^.*x//
y/0123456789/1234567890/
x

# keep unchanged digits in pattern space
s/x.*$//

# compose the new number, remove the newline implicitly added by G
G
s/\n//
h
```

7.11 Numbering Non-blank Lines

Emulating '`cat -b`' is almost the same as '`cat -n`'—we only have to select which lines are to be numbered and which are not.

The part that is common to this script and the previous one is not commented to show how important it is to comment `sed` scripts properly...

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

/^$/ {
  p
  b
}
```

```
# Same as cat -n from now
x
/^$/ s/^.*$/1/
G
h
s/^/      /
s/^ *\(.....\)\n/\1  /p
x
s/\n.*$//
/^9*$/ s/^/0/
s/.9*$/x&/
h
s/^.*x//
y/0123456789/1234567890/
x
s/x.*$//
G
s/\n//
h
```

7.12 Counting Characters

This script shows another way to do arithmetic with `sed`. In this case we have to add possibly large numbers, so implementing this by successive increments would not be feasible (and possibly even more complicated to contrive than this script).

The approach is to map numbers to letters, kind of an abacus implemented with `sed`. '`a`'s are units, '`b`'s are tens and so on: we simply add the number of characters on the current line as units, and then propagate the carry to tens, hundreds, and so on.

As usual, running totals are kept in hold space.

On the last line, we convert the abacus form back to decimal. For the sake of variety, this is done with a loop rather than with some 80 `s` commands[11]: first we convert units, removing '`a`'s from the number; then we rotate letters so that tens become '`a`'s, and so on until no more letters remain.

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

# Add n+1 a's to hold space (+1 is for the newline)
s/./a/g
H
x
s/\n/a/
```

[11] Some implementations have a limit of 199 commands per script

```
# Do the carry.  The t's and b's are not necessary,
# but they do speed up the thing
t a
: a;  s/aaaaaaaaaa/b/g; t b; b done
: b;  s/bbbbbbbbbb/c/g; t c; b done
: c;  s/cccccccccc/d/g; t d; b done
: d;  s/dddddddddd/e/g; t e; b done
: e;  s/eeeeeeeeee/f/g; t f; b done
: f;  s/ffffffffff/g/g; t g; b done
: g;  s/gggggggggg/h/g; t h; b done
: h;  s/hhhhhhhhhh//g

: done
$! {
  h
  b
}

# On the last line, convert back to decimal

: loop
/a/! s/[b-h]*/&0/
s/aaaaaaaaa/9/
s/aaaaaaaa/8/
s/aaaaaaa/7/
s/aaaaaa/6/
s/aaaaa/5/
s/aaaa/4/
s/aaa/3/
s/aa/2/
s/a/1/

: next
y/bcdefgh/abcdefg/
/[a-h]/ b loop
p
```

7.13 Counting Words

This script is almost the same as the previous one, once each of the words on the line is converted to a single 'a' (in the previous script each letter was changed to an 'a').

It is interesting that real `wc` programs have optimized loops for '`wc -c`', so they are much slower at counting words rather than characters. This script's bottleneck, instead, is arithmetic, and hence the word-counting one is faster (it has to manage smaller numbers).

Again, the common parts are not commented to show the importance of commenting `sed` scripts.

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

```
# Convert words to a's
s/[ tab][ tab]*/ /g
s/^/ /
s/ [^ ][^ ]*/a /g
s/ //g

# Append them to hold space
H
x
s/\n//

# From here on it is the same as in wc -c.
/aaaaaaaaaa/! bx;   s/aaaaaaaaaa/b/g
/bbbbbbbbbb/! bx;   s/bbbbbbbbbb/c/g
/cccccccccc/! bx;   s/cccccccccc/d/g
/dddddddddd/! bx;   s/dddddddddd/e/g
/eeeeeeeeee/! bx;   s/eeeeeeeeee/f/g
/ffffffffff/! bx;   s/ffffffffff/g/g
/gggggggggg/! bx;   s/gggggggggg/h/g
s/hhhhhhhhhh//g
:x
$! { h; b; }
:y
/a/! s/[b-h]*/&0/
s/aaaaaaaaa/9/
s/aaaaaaaa/8/
s/aaaaaaa/7/
s/aaaaaa/6/
s/aaaaa/5/
s/aaaa/4/
s/aaa/3/
s/aa/2/
s/a/1/
y/bcdefgh/abcdefg/
/[a-h]/ by
p
```

7.14 Counting Lines

No strange things are done now, because `sed` gives us '`wc -l`' functionality for free!!! Look:

```
#!/usr/bin/sed -nf
$=
```

7.15 Printing the First Lines

This script is probably the simplest useful `sed` script. It displays the first 10 lines of input; the number of displayed lines is right before the `q` command.

```
#!/usr/bin/sed -f
10q
```

7.16 Printing the Last Lines

Printing the last *n* lines rather than the first is more complex but indeed possible. *n* is encoded in the second line, before the bang character.

This script is similar to the `tac` script in that it keeps the final output in the hold space and prints it at the end:

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

1! {; H; g; }
1,10 !s/[^\n]*\n//
$p
h
```

Mainly, the scripts keeps a window of 10 lines and slides it by adding a line and deleting the oldest (the substitution command on the second line works like a `D` command but does not restart the loop).

The "sliding window" technique is a very powerful way to write efficient and complex `sed` scripts, because commands like `P` would require a lot of work if implemented manually.

To introduce the technique, which is fully demonstrated in the rest of this chapter and is based on the `N`, `P` and `D` commands, here is an implementation of `tail` using a simple "sliding window."

This looks complicated but in fact the working is the same as the last script: after we have kicked in the appropriate number of lines, however, we stop using the hold space to keep inter-line state, and instead use `N` and `D` to slide pattern space by one line:

```
#!/usr/bin/sed -f

1h
2,10 {; H; g; }
$q
1,9d
N
D
```

Note how the first, second and fourth line are inactive after the first ten lines of input. After that, all the script does is: exiting on the last line of input, appending the next input line to pattern space, and removing the first line.

7.17 Make Duplicate Lines Unique

This is an example of the art of using the `N`, `P` and `D` commands, probably the most difficult to master.

```
#!/usr/bin/sed -f
h

:b
# On the last line, print and exit
$b
N
/^\(.*\)\n\1$/ {
    # The two lines are identical.  Undo the effect of
    # the n command.
    g
    bb
}

# If the N command had added the last line, print and exit
$b

# The lines are different; print the first and go
# back working on the second.
P
D
```

As you can see, we maintain a 2-line window using `P` and `D`. This technique is often used in advanced `sed` scripts.

7.18 Print Duplicated Lines of Input

This script prints only duplicated lines, like '`uniq -d`'.

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

$b
N
/^\(.*\)\n\1$/ {
    # Print the first of the duplicated lines
    s/.*\n//
    p

    # Loop until we get a different line
    :b
    $b
    N
    /^\(.*\)\n\1$/ {
        s/.*\n//
        bb
    }
}
```

```
# The last line cannot be followed by duplicates
$b

# Found a different one.  Leave it alone in the pattern space
# and go back to the top, hunting its duplicates
D
```

7.19 Remove All Duplicated Lines

This script prints only unique lines, like '`uniq -u`'.

```
#!/usr/bin/sed -f

# Search for a duplicate line --- until that, print what you find.
$b
N
/^\(.*\)\n\1$/ ! {
    P
    D
}

:c
# Got two equal lines in pattern space.  At the
# end of the file we simply exit
$d

# Else, we keep reading lines with N until we
# find a different one
s/.*\n//
N
/^\(.*\)\n\1$/ {
    bc
}

# Remove the last instance of the duplicate line
# and go back to the top
D
```

7.20 Squeezing Blank Lines

As a final example, here are three scripts, of increasing complexity and speed, that implement the same function as '`cat -s`', that is squeezing blank lines.

The first leaves a blank line at the beginning and end if there are some already.

```
#!/usr/bin/sed -f
```

```
# on empty lines, join with next
# Note there is a star in the regexp
:x
/^\n*$/ {
N
bx
}

# now, squeeze all '\n', this can be also done by:
# s/^\(\n\)*/\1/
s/\n*/\
/
```

This one is a bit more complex and removes all empty lines at the beginning. It does leave a single blank line at end if one was there.

```
#!/usr/bin/sed -f

# delete all leading empty lines
1,/^./{
/./!d
}

# on an empty line we remove it and all the following
# empty lines, but one
:x
/./!{
N
s/^\n$//
tx
}
```

This removes leading and trailing blank lines. It is also the fastest. Note that loops are completely done with `n` and `b`, without relying on `sed` to restart the script automatically at the end of a line.

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

# delete all (leading) blanks
/./!d

# get here: so there is a non empty
:x
# print it
p
# get next
n
# got chars? print it again, etc...
/./bx
```

```
# no, don't have chars: got an empty line
:z
# get next, if last line we finish here so no trailing
# empty lines are written
n
# also empty? then ignore it, and get next... this will
# remove ALL empty lines
/./!bz

# all empty lines were deleted/ignored, but we have a non empty.  As
# what we want to do is to squeeze, insert a blank line artificially
i\

bx
```

8 GNU sed's Limitations and Non-limitations

For those who want to write portable `sed` scripts, be aware that some implementations have been known to limit line lengths (for the pattern and hold spaces) to be no more than 4000 bytes. The POSIX standard specifies that conforming `sed` implementations shall support at least 8192 byte line lengths. GNU `sed` has no built-in limit on line length; as long as it can `malloc()` more (virtual) memory, you can feed or construct lines as long as you like.

However, recursion is used to handle subpatterns and indefinite repetition. This means that the available stack space may limit the size of the buffer that can be processed by certain patterns.

9 Other Resources for Learning About sed

For up to date information about GNU `sed` please visit `https://www.gnu.org/software/sed/`.

Send general questions and suggestions to `sed-devel@gnu.org`. Visit the mailing list archives for past discussions at `https://lists.gnu.org/archive/html/sed-devel/`.

The following resources provide information about `sed` (both GNU `sed` and other variations). Note these not maintained by GNU `sed` developers.

- sed `$HOME`: `http://sed.sf.net`
- sed FAQ: `http://sed.sf.net/sedfaq.html`
- seder's grabbag: `http://sed.sf.net/grabbag`
- The `sed-users` mailing list maintained by Sven Guckes: `http://groups.yahoo.com/group/sed-users/` (note this is *not* the GNU `sed` mailing list).

10 Reporting Bugs

Email bug reports to `bug-sed@gnu.org`. Also, please include the output of '`sed --version`' in the body of your report if at all possible.

Please do not send a bug report like this:

```
while building frobme-1.3.4
$ configure
error  sed: file sedscr line 1: Unknown option to 's'
```

If GNU `sed` doesn't configure your favorite package, take a few extra minutes to identify the specific problem and make a stand-alone test case. Unlike other programs such as C compilers, making such test cases for `sed` is quite simple.

A stand-alone test case includes all the data necessary to perform the test, and the specific invocation of `sed` that causes the problem. The smaller a stand-alone test case is, the better. A test case should not involve something as far removed from `sed` as "try to configure frobme-1.3.4". Yes, that is in principle enough information to look for the bug, but that is not a very practical prospect.

Here are a few commonly reported bugs that are not bugs.

`N` command on the last line

Most versions of `sed` exit without printing anything when the `N` command is issued on the last line of a file. GNU `sed` prints pattern space before exiting unless of course the `-n` command switch has been specified. This choice is by design.

Default behavior (gnu extension, non-POSIX conforming):

```
$ seq 3 | sed N
1
2
3
```

To force POSIX-conforming behavior:

```
$ seq 3 | sed --posix N
1
2
```

For example, the behavior of

```
sed N foo bar
```

would depend on whether foo has an even or an odd number of lines[12]. Or, when writing a script to read the next few lines following a pattern match, traditional implementations of `sed` would force you to write something like

```
/foo/{ $!N; $!N; $!N; $!N; $!N; $!N; $!N; $!N; $!N }
```

instead of just

```
/foo/{ N;N;N;N;N;N;N;N;N; }
```

In any case, the simplest workaround is to use `$d;N` in scripts that rely on the traditional behavior, or to set the `POSIXLY_CORRECT` variable to a non-empty value.

[12] which is the actual "bug" that prompted the change in behavior

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`’.

This syntax may cause problems when running scripts written for other `sed`s. Some `sed` programs have been written with the assumption that `\|` and `\+` match the literal characters `|` and `+`. Such scripts must be modified by removing the spurious backslashes if they are to be used with modern implementations of `sed`, like GNU `sed`.

On the other hand, some scripts use `s|abc\|def||g` to remove occurrences of *either* `abc` or `def`. While this worked until `sed` 4.0.x, newer versions interpret this as removing the string `abc|def`. This is again undefined behavior according to POSIX, and this interpretation is arguably more robust: older `sed`s, for example, required that the regex matcher parsed `\/` as `/` in the common case of escaping a slash, which is again undefined behavior; the new behavior avoids this, and this is good because the regex matcher is only partially under our control.

In addition, this version of `sed` supports several escape characters (some of which are multi-character) to insert non-printable characters in scripts (`\a`, `\c`, `\d`, `\o`, `\r`, `\t`, `\v`, `\x`). These can cause similar problems with scripts written for other `sed`s.

`-i` clobbers read-only files

In short, ‘`sed -i`’ will let you delete the contents of a read-only file, and in general the `-i` option (see Chapter 2 [Invocation], page 1) lets you clobber protected files. This is not a bug, but rather a consequence of how the Unix file system works.

The permissions on a file say what can happen to the data in that file, while the permissions on a directory say what can happen to the list of files in that directory. ‘`sed -i`’ will not ever open for writing a file that is already on disk. Rather, it will work on a temporary file that is finally renamed to the original name: if you rename or delete files, you’re actually modifying the contents of the directory, so the operation depends on the permissions of the directory, not of the file. For this same reason, `sed` does not let you use `-i` on a writable file in a read-only directory, and will break hard or symbolic links when `-i` is used on such a file.

`0a` does not work (gives an error)

There is no line 0. 0 is a special address that is only used to treat addresses like `0,/RE/` as active when the script starts: if you write `1,/abc/d` and the first line includes the word ‘`abc`’, then that match would be ignored because address ranges must span at least two lines (barring the end of the file); but what you probably wanted is to delete every line up to the first one including ‘`abc`’, and this is obtained with `0,/abc/d`.

`[a-z]` is case insensitive

You are encountering problems with locales. POSIX mandates that `[a-z]` uses the current locale's collation order – in C parlance, that means using `strcoll(3)` instead of `strcmp(3)`. Some locales have a case-insensitive collation order, others don't.

Another problem is that `[a-z]` tries to use collation symbols. This only happens if you are on the GNU system, using GNU libc's regular expression matcher instead of compiling the one supplied with GNU sed. In a Danish locale, for example, the regular expression `^[a-z]$` matches the string '`aa`', because this is a single collating symbol that comes after '`a`' and before '`b`'; '`ll`' behaves similarly in Spanish locales, or '`ij`' in Dutch locales.

To work around these problems, which may cause bugs in shell scripts, set the `LC_COLLATE` and `LC_CTYPE` environment variables to '`C`'.

`s/.*//` does not clear pattern space

This happens if your input stream includes invalid multibyte sequences. POSIX mandates that such sequences are *not* matched by '`.`', so that '`s/.*//`' will not clear pattern space as you would expect. In fact, there is no way to clear sed's buffers in the middle of the script in most multibyte locales (including UTF-8 locales). For this reason, GNU `sed` provides a 'z' command (for 'zap') as an extension.

To work around these problems, which may cause bugs in shell scripts, set the `LC_COLLATE` and `LC_CTYPE` environment variables to '`C`'.

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

This is a general index of all issues discussed in this manual, with the exception of the `sed` commands and command-line options.

–

–expression, example 1
–file, example 1
-e, example 1, 6
-f, example 1, 6
-i, example 1
-n, example 1
-s, example 1

;

;, command separator 6

0

`0` address 64

A

`a`, and semicolons 6
Additional reading about `sed` 62
addr1,+N 25
addr1,~N 25
address range, example 6
Address, as a regular expression 23
Address, last line 22
Address, numeric 22
addresses, excluding 22
Addresses, in `sed` scripts 22
addresses, negating 22
addresses, numeric 22
addresses, range 22
addresses, regular expression 22
addresses, syntax 5
alphabetic characters 30
alphanumeric characters 30
Append hold space to pattern space 16
Append next input line to pattern space 16
Append pattern space to hold space 16
Appending text after a line 12

B

b, joining lines with 42
b, versus t 42
back-reference 33
Backreferences, in regular expressions 8
blank characters 30
bracket expression 30
Branch to a label, if `s///` failed 18
Branch to a label, if `s///` succeeded 17
Branch to a label, unconditionally 17
branching and n, N 41
branching, infinite loop 41
branching, joining lines 42
Buffer spaces, pattern and hold 37
Bugs, reporting 63

C

`c`, and semicolons 6
case insensitive, regular expression 23
Case-insensitive matching 10
Caveat — #n on first line 10
character class 30
character classes 30
classes of characters 30
Command groups 12
Comments, in scripts 10
Conditional branch 17, 18
control characters 30
Copy hold space into pattern space 16
Copy pattern space into hold space 16
cycle, restarting 41

D

`d`, example 6
Delete first line from pattern space 16
digit characters 30
Disabling autoprint, from command line 2

E

empty regular expression 23
Emptying pattern space 18, 65
Evaluate Bourne-shell commands 17
Evaluate Bourne-shell commands, after substitution 10
example, address range 6
example, regular expression 6
Exchange hold space with pattern space 16
Excluding lines 22
exit status 5
exit status, example 5
Extended regular expressions, choosing 4
Extended regular expressions, syntax 29

F

File name, printing 17
Files to be processed as input 5
Flow of control in scripts 17

G

Global substitution 9
GNU extensions, `/dev/stderr` file 10, 16
GNU extensions, `/dev/stdin` file 16, 18
GNU extensions, `/dev/stdout` file 2, 10, 16
GNU extensions, `0` address 25, 64
GNU extensions, 0,*addr2* addressing 25
GNU extensions, *addr1*,+*N* addressing 25
GNU extensions, *addr1*,~*N* addressing 25
GNU extensions, branch if `s///` failed 18
GNU extensions, case modifiers in `s` commands .. 9
GNU extensions, checking for their presence 18
GNU extensions, disabling 3
GNU extensions, emptying pattern space ... 18, 65
GNU extensions, evaluating Bourne-shell commands 10, 17
GNU extensions, extended regular expressions ... 4
GNU extensions, `g` and *number* modifier 9
GNU extensions, `I` modifier 10, 23
GNU extensions, in-place editing 2, 64
GNU extensions, `M` modifier 10, 24
GNU extensions, modifiers and the empty regular expression 23
GNU extensions, '*n*~*m*' addresses 22
GNU extensions, quitting silently 17
GNU extensions, `R` command 18
GNU extensions, reading a file a line at a time 18
GNU extensions, returning an exit code 10, 17
GNU extensions, setting line length 15
GNU extensions, special escapes 34, 64
GNU extensions, special two-address forms 25
GNU extensions, subprocesses 10, 17
GNU extensions, to basic regular expressions 27, 28, 64
GNU extensions, two addresses supported by most commands 13, 14, 15, 16
GNU extensions, unlimited line length 62
GNU extensions, writing first line to a file 18
Goto, in scripts 17
graphic characters 31
Greedy regular expression matching 28
Grouping commands 12

H

hexadecimal digits 31
Hold space, appending from pattern space 16
Hold space, appending to pattern space 16
Hold space, copy into pattern space 16
Hold space, copying pattern space into 16
Hold space, definition 37
Hold space, exchange with pattern space 16

I

`i`, and semicolons 6
In-place editing 64
In-place editing, activating 2
In-place editing, Perl-style backup file names 2
infinite loop, branching 41
Inserting text before a line 13

J

joining lines with branching 42
joining quoted-printable lines 42

L

labels 41
Labels, in scripts 17
Last line, selecting 22
Line length, setting 3, 15
Line number, printing 15
Line selection 22
Line, selecting by number 22
Line, selecting by regular expression match 23
Line, selecting last 22
List pattern space 15
lower-case letters 31

M

Mixing `g` and *number* modifiers in the `s` command 9
multiple files 1
multiple `sed` commands 6

N

n, and branching ... 41
N, and branching ... 41
named character classes ... 30
newline, command separator ... 6
Next input line, append to pattern space ... 16
Next input line, replace pattern space with ... 11
Non-bugs, `0` address ... 64
Non-bugs, in-place editing ... 64
Non-bugs, localization-related ... 65
Non-bugs, `N` command on the last line ... 63
Non-bugs, regex syntax clashes ... 64
numeric addresses ... 22
numeric characters ... 30

O

omitting labels ... 41
output ... 1
output, suppressing ... 1

P

p, example ... 1
paragraphs, processing ... 39
parameters, script ... 1
Parenthesized substrings ... 8
Pattern space, definition ... 37
Portability, comments ... 10
Portability, line length limitations ... 62
Portability, `N` command on the last line ... 63
`POSIXLY_CORRECT` behavior,
bracket expressions ... 31
`POSIXLY_CORRECT` behavior, enabling ... 3
`POSIXLY_CORRECT` behavior, escapes ... 34
`POSIXLY_CORRECT` behavior, `N` command ... 63
Print first line from pattern space ... 16
printable characters ... 31
Printing file name ... 17
Printing line number ... 15
Printing text unambiguously ... 15
processing paragraphs ... 39
punctuation characters ... 31

Q

`q`, example ... 6
Q, example ... 5
Quitting ... 10, 17
quoted-printable lines, joining ... 42

R

range addresses ... 22
range expression ... 30
Range of lines ... 24
Range with start address of zero ... 25
Read next input line ... 11
Read text from a file ... 15, 18
regex addresses and input lines ... 24
regex addresses and pattern space ... 24
regular expression addresses ... 22
regular expression, example ... 6
Replace hold space with copy of pattern space ... 16
Replace pattern space with copy of hold space ... 16
Replacing all text matching regexp in a line ... 9
Replacing only *n*th match of regexp in a line ... 9
Replacing selected lines with other text ... 14
Requiring GNU `sed` ... 18
restarting a cycle ... 41

S

Sandbox mode ... 4
script parameter ... 1
Script structure ... 5
Script, from a file ... 2
Script, from command line ... 2
`sed` commands syntax ... 5
`sed` commands, multiple ... 6
`sed` script structure ... 5
Selecting lines to process ... 22
Selecting non-matching lines ... 22
semicolons, command separator ... 6
Several lines, selecting ... 24
Slash character, in regular expressions ... 23
space characters ... 31
Spaces, pattern and hold ... 37
Special addressing forms ... 25
standard input ... 1
Standard input, processing as input ... 5
standard output ... 1
stdin ... 1
stdout ... 1
Stream editor ... 1
subexpression ... 33
Subprocesses ... 10, 17
Substitution of text, options ... 9
suppressing output ... 1
syntax, addresses ... 5
syntax, `sed` commands ... 5

T

t, joining lines with 42
t, versus b 42
Text, appending 12
Text, deleting 11
Text, insertion 13
Text, printing 11
Text, printing after substitution 9
Text, writing to a file after substitution 10
Transliteration 12

U

Unbuffered I/O, choosing 5
upper-case letters 31
Usage summary, printing 2

V

Version, printing 2

W

whitespace characters 31
Working on separate files 4
Write first line to a file 18
Write to a file 16

X

xdigit class 31

Z

Zero, as range start address 25

Command and Option Index

This is an alphabetical list of all `sed` commands and command-line options.

#

`# (comments)` 10

—

`--binary` 4
`--expression` 2
`--file` 2
`--follow-symlinks` 4
`--help` 2
`--in-place` 2
`--line-length` 3
`--null-data` 5
`--posix` 3
`--quiet` 2
`--regexp-extended` 4
`--sandbox` 4
`--separate` 4
`--silent` 2
`--unbuffered` 5
`--version` 2
`--zero-terminated` 5
`-b` 4
`-e` 2
`-E` 4
`-f` 2
`-i` 2
`-l` 3
`-n` 2
`-n, forcing from within a script` 10
`-r` 4
`-s` 4
`-u` 5
`-z` 5

:

`: (label) command` 17

=

`= (print line number) command` 15

{

`{}` command grouping 12

A

`a (append text lines) command` 12
`alnum` character class 30
`alpha` character class 30

B

`b (branch) command` 17
`blank` character class 30

C

`c (change to text lines) command` 14
`cntrl` character class 30

D

`d (delete) command` 11
`D (delete first line) command` 16
`digit` character class 30

E

`e (evaluate) command` 17

F

`F (File name) command` 17

G

`g (get) command` 16
`G (appending Get) command` 16
`graph` character class 31

H

`h (hold) command` 16
`H (append Hold) command` 16

I

`i (insert text lines) command` 13

L

`l (list unambiguously) command` 15
`lower` character class 31

N

`n (next-line) command` 11
`N (append Next line) command` 16

P

p (print) command 11
P (print first line) command 16
print character class 31
punct character class 31

Q

q (quit) command 10
Q (silent Quit) command 17

R

r (read file) command 15
R (read line) command 18

S

s command, option flags 9
space character class 31

T

t (test and branch if successful) command ... 17
T (test and branch if failed) command 18

U

upper character class 31

V

v (version) command 18

W

w (write file) command 16
W (write first line) command 18

X

x (eXchange) command 16
xdigit character class 31

Y

y (transliterate) command 12

Z

z (Zap) command 18

Table of Contents

1 Introduction 1

2 Running sed 1

2.1 Overview 1
2.2 Command-Line Options 2
2.3 Exit status 5

3 `sed` scripts 5

3.1 `sed` script overview 5
3.2 `sed` commands summary 6
3.3 The `s` Command 8
3.4 Often-Used Commands 10
3.5 Less Frequently-Used Commands 12
3.6 Commands for `sed` gurus 17
3.7 Commands Specific to GNU `sed` 17
3.8 Multiple commands syntax 18
3.8.1 Commands Requiring a newline 19

4 Addresses: selecting lines 21

4.1 Addresses overview 22
4.2 Selecting lines by numbers 22
4.3 selecting lines by text matching 23
4.4 Range Addresses 24

5 Regular Expressions: selecting text 26

5.1 Overview of regular expression in `sed` 26
5.2 Basic (BRE) and extended (ERE) regular expression 26
5.3 Overview of basic regular expression syntax 27
5.4 Overview of extended regular expression syntax 29
5.5 Character Classes and Bracket Expressions 30
5.6 regular expression extensions 32
5.7 Back-references and Subexpressions 33
5.8 Escape Sequences - specifying special characters 34
5.8.1 Escaping Precedence 34
5.9 Multibyte characters and Locale Considerations 35
5.9.1 Invalid multibyte characters 35
5.9.2 Upper/Lower case conversion 37
5.9.3 Multibyte regexp character classes 37

6 Advanced sed: cycles and buffers 37
6.1 How sed Works 37
6.2 Hold and Pattern Buffers 38
6.3 Multiline techniques - using D,G,H,N,P to process multiple lines .. 38
6.4 Branching and Flow Control 40
6.4.1 Branching and Cycles 41
6.4.2 Branching example: joining lines 42

7 Some Sample Scripts 43
7.1 Joining lines 43
7.2 Centering Lines 44
7.3 Increment a Number 45
7.4 Rename Files to Lower Case 46
7.5 Print bash Environment 48
7.6 Reverse Characters of Lines 49
7.7 Text search across multiple lines 50
7.8 Line length adjustment 51
7.9 Reverse Lines of Files 52
7.10 Numbering Lines 53
7.11 Numbering Non-blank Lines 54
7.12 Counting Characters 55
7.13 Counting Words 56
7.14 Counting Lines 57
7.15 Printing the First Lines 58
7.16 Printing the Last Lines 58
7.17 Make Duplicate Lines Unique 58
7.18 Print Duplicated Lines of Input 59
7.19 Remove All Duplicated Lines 60
7.20 Squeezing Blank Lines 60

8 GNU sed's Limitations and Non-limitations .. 62

9 Other Resources for Learning About sed 62

10 Reporting Bugs 63

Appendix A GNU Free Documentation License .. 66

Concept Index 74

Command and Option Index 78