sed, a stream editor

version 4.1.5, 30 January 2006

by Ken Pizzini, Paolo Bonzini

This file documents version 4.1.5 of GNU `sed`, a stream editor.

Published by the Free Software Foundation,
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA

sed, a stream editor

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 Invocation

Normally `sed` is invoked like this:

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

The full format for invoking `sed` is:

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

If you do not specify *INPUTFILE*, or if *INPUTFILE* is '`-`', `sed` filters the contents of the standard input. The *script* is actually the first non-option parameter, which `sed` specially considers a script and not an input file if (and only if) none of the other *options* specifies a script to be executed, that is if neither of the '`-e`' and '`-f`' options is specified.

`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. These options disable this automatic printing, and `sed` only produces output when explicitly told to via the `p` command.

`-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

[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

of the old file before renaming the temporary file, thereby making a backup copy[2]).

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.

`-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 see Chapter 7 [Reporting Bugs], page 30) 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.

`-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 less backslashes, but are a GNU extension and hence scripts that use them are not portable. See Appendix A [Extended regular expressions], page 32.

`-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.

`-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.)

---

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

`-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.

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.

3 `sed` Programs

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 catenation of all of the *script*s and *script-file*s passed in.

Each `sed` command consists of an optional address or address range, followed by a one-character command name and any additional command-specific code.

3.1 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 lines 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.[3] 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).

---

[3] 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`.

3.2 Selecting lines with sed

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.)

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, to select the odd-numbered lines, one would use `1~2`; 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`.

`$` 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.

/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 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 (\).

*/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.

*/regexp/*M
*\%regexp%*M
: The `M` modifier to regular-expression matching is a GNU `sed` extension which 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. `M` stands for *multi-line*.

If no addresses are given, then all lines are matched; if one address is given, then only lines matching that address are matched.

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).

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).

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.

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.

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

`addr1,~N`
: Matches *addr1* and the lines following *addr1* until the next line whose input line number is a multiple of *N*.

Appending the `!` character to the end of an address specification negates the sense of the match. That is, if the `!` character follows an address range, then only lines which do *not* match the address range will be selected. This also works for singleton addresses, and, perhaps perversely, for the null address.

3.3 Overview of Regular Expression Syntax

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 in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. 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. 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.

`\+` 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).

`.` Matches any character, including newline.

`^` Matches the null string at beginning of line, i.e. what appears after the circumflex must appear at the beginning of line. `^#include` will match only lines where '`#include`' is the first thing on line—if there are spaces before, for example, the match fails. `^` 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.

`$` It is the same as `^`, but refers to end of line. `$` 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*.

A leading `^` reverses the meaning of *list*, so that it matches any single character *not* in *list*. To include `]` in the list, make it the first character (after the `^` if needed), to include `-` in the list, make it the first or last; to include `^` put it after the first character.

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 3.9 [Escapes], page 13.

`*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 implicity 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 '`aaaaaabbbbbbb`'.

'`.*`'
'`.\+`'
: 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.*(.*)`'
: his 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'.

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

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 conformant) 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`*`]`
: This command only accepts a single address.

  Exit `sed` without processing any more commands or input. 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.

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

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

`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.

`{ `*`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.

3.5 The s Command

The syntax of the `s` (as in substitute) command is '`s/regexp/replacement/flags`'. 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.

The `s` command is probably the most important in `sed` and has a lot of different options. Its basic concept is simple: the `s` command attempts to match the pattern space against the supplied *regexp*; if the match is successful, then that portion of the pattern space which was matched is replaced with *replacement*.

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. 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`.

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*.

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` *`file-name`*

If the substitution was made, then write out the result to the named file. As a GNU `sed` extension, two special values of *file-name* are supported:

'/dev/stderr', which writes the result to the standard error, and '/dev/stdout', which writes to the standard output.[4]

`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 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. `M` stands for *multi-line*.

3.6 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/`
: (The `/` characters may be uniformly replaced by any other single character within any given `y` command.)

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

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).

`a\`
`text`
: As a GNU extension, this command accepts two addresses.

Queue 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.

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

As a GNU extension, if between the `a` and the newline there is other than a whitespace-`\` sequence, then the text of this line, starting at the first non-whitespace character after the `a`, is taken as the first line of the *text* block. (This enables a simplification in scripting a one-line add.) This extension also works with the `i` and `c` commands.

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

`i\`
`text`
: As a GNU extension, this command accepts two addresses.

  Immediately output the lines of text which follow this command (each but the last ending with a `\`, which are removed from the output).

`c\`
`text`
: Delete the lines matching the address or address-range, and output the lines of text which follow this command (each but the last ending with a `\`, which are removed from the output) in place of the last line (or in place of each line, if no addresses were specified). A new cycle is started after this command is done, since the pattern space will have been deleted.

`=`
: As a GNU extension, this command accepts two addresses.

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

`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`*
: As a GNU extension, this command accepts two addresses.

  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.

`w` *`filename`*
: Write the pattern space to *filename*. As a GNU `sed` extension, two special values of *file-name* are supported: '`/dev/stderr`', which writes the result to the standard error, and '`/dev/stdout`', which writes to the standard output.[5]

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

`D`
: Delete text in the pattern space up to the first newline. If any text is left, restart cycle with the resultant pattern space (without reading a new line of input), otherwise start a normal new cycle.

`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.

---

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

`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.

3.7 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 `s`ubstitution 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.

3.8 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 `sed`s.

`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 (like `r` does). 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.

`L n` This GNU `sed` extension fills and joins lines in pattern space to produce output lines of (at most) *n* characters, like `fmt` does; if *n* is omitted, the default as specified on the command line is used. This command is considered a failed experiment and unless there is enough request (which seems unlikely) will be removed in future versions.

`Q [exit-code]`

This command only accepts a single 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 `s`ubstitutions 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.

3.9 GNU Extensions for Escapes in Regular Expressions

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

[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.

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).

`\cx` Produces or matches CONTROL-`x`, where *x* is any character. The precise effect of '`\cx`' 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.

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

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

`\xxx` 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.

Other escapes match a particular character class and are valid only in regular expressions:

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

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

`\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.

`\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.

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

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

4 Some Sample Scripts

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

4.1 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
}

# del 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/
```

4.2 Increment a Number

This script is one of a few that demonstrate how to do arithmetic in `sed`. This is indeed possible,[7] 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 4.7 [cat -n], page 20, 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.

---

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

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

/[^0-9]/ d

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

# incr last digit only.  The first line adds a most-significant
# digit of 1 if we have to add a digit.
#
# The tn commands are not necessary, but make the thing
# faster

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/
```

4.3 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 nothing
/^.*\/\(.*\)\n\1/b

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

' | $apply_cmd
```

4.4 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
'
```

4.5 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`.[8]

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

[8] 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
```

```
/../! 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
```

4.6 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
```

4.7 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
```

4.8 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
```

4.9 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[9]: 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/

# 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/
```

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

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

4.10 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
```

4.11 Counting Lines

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

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

4.12 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
```

4.13 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.

4.14 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 mantain a 2-line window using `P` and `D`. This technique is often used in advanced `sed` scripts.

4.15 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
```

4.16 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
```

4.17 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 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
```

5 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.

6 Other Resources for Learning About `sed`

In addition to several books that have been written about `sed` (either specifically or as chapters in books which discuss shell programming), one can find out more about `sed` (including suggestions of a few books) from the FAQ for the `sed-users` mailing list, available from any of:

`http://www.student.northpark.edu/pemente/sed/sedfaq.html`
`http://sed.sf.net/grabbag/tutorials/sedfaq.html`

Also of interest are `http://www.student.northpark.edu/pemente/sed/index.htm` and `http://sed.sf.net/grabbag`, which include `sed` tutorials and other `sed`-related goodies.

The `sed-users` mailing list itself maintained by Sven Guckes. To subscribe, visit `http://groups.yahoo.com` and search for the `sed-users` mailing list.

7 Reporting Bugs

Email bug reports to `bonzini@gnu.org`. Be sure to include the word "sed" somewhere in the `Subject:` field. 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.

For example, the behavior of

```
sed N foo bar
```

would depend on whether foo has an even or an odd number of lines[10]. 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.

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

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

protected files. This is not a bug, but rather a consequence of how the Unix filesystem 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 writeable file in a read-only directory (but unbelievably nobody reports that as a bug. . .).

`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: one of those that have problems is Estonian.

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`'.

Appendix A Extended regular expressions

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

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.

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.

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W

Z

Command and Option Index

This is an alphabetical list of all `sed` commands and command-line options.

V

W

X

Y

Table of Contents