Chapter
Computational Biology
pp 175-193

Date: 29 January 2013

Sed

Röbbe Wünschiers

10.1007/978-3-642-34749-8_12
Copyright information

Abstract

Sed (**s**tream **ed**itor) is a non-interactive, line-oriented, stream editor. Need to exchange decimal delimiters or replace text strings in a gigabyte seized file? Sed is you solution.
Sed (**s**tream **ed**itor) is a non-interactive, line-oriented, stream editor. What does that mean?

Non-interactive means that the editor takes its editing commands from either the command line or a file. Once started, it runs through the whole text file that you wish to edit. That sounds worse than it is. Sed does not really edit and change the original text file itself but prints the editing result to the standard output. If you want to keep the result, you just pipe it into a file.

Line-oriented means that Sed treats the input file line by line. A line ends with (and a new line begins after) the (invisible) newline character. This meta character is only visible as a line break when you look at the text file with, for example, `cat` or Vim. This has the disadvantage that, as with `egrep`, you cannot edit text that is spanning multiple lines. If you would like to change the words "*gene expression*" to "*transcription*" you can only do that when "*gene expression*" sits in one single line. If the end of one line contained the word "*gene*" and the beginning of the following line contained the word "*expression*", this would not be recognized. It is, however, possible to delete, insert and change multiple lines.

Stream-oriented means that Sed "swallows" a whole file at once. We say: the file is treated like a stream. Imagine a text file as a pile of sheets of paper. Each sheet represents one line of the text file. Now you take one sheet, edit it according to given rules and put it aside. This you do for all sheets—you cannot stop until you have edited the last sheet. Thus, you edit a stream of sheets—Sed is editing a stream of lines.

Sed is a very old Unix tool. It has its roots in `ed`, a *line editor*. `ed`, however, has been completely replaced by editors like Vim. In the next section we will learn why Sed survived.

12.1 When to Use Sed?

Why should I learn how to use a simple non-interactive editor when I know how to use Vim?—you might ask. Indeed, this is a valid question in times when everybody talks about economics. Is it economic to learn Sed? Yes! If you want to bake a cake "from scratch", you can either mix the dough quickly by hand, or set up the Braun super mixer for the same job. Sed is the "handy" way. It is quick and has very little memory requirements. This is an advantage if you work with large files (some Megabytes). Sed loads one line into the computer's memory (RAM), edits the line and prints the output either onto the screen or into a file; and Sed is damn fast. I know of no other editor that can compete with Sed. Sed allows for very advanced editing commands. However, Sed is only comfortable to use with small editing tasks. For more advanced tasks one might prefer to use AWK or Perl.

What are typical tasks for Sed? Text substitutions! The most important thing one can do with Sed is the substitution of defined text patterns with other text. Here are some examples: removing *HTML tags* from files, changing the *decimal markers* from commas to points, changing words like *colour* to *color* throughout a text document, removing *comment lines* from shell scripts, or *formatting* text files are only a small number of examples of what can be done with Sed.

For repeatedly occurring tasks you can write *sed scripts* or include *sed* in shell scripts (see Program 26 on p. 155 in Sect. 10.11.4 on p. 155).

12.2 Getting Started

How is Sed used? The basic syntax is quite straightforward. What makes Sed scripts sometimes look complicated are the regular expressions which find the text to edit and the shortcuts used for the commands.

Let us start with a small Sed editing command.

Terminal 78: The Substitute Command s/.../.../

```
1 $ cat>test.file.txt
2 I think most spring flowers bloom white.
3 Is that caused by gene regulation?
4 $ sed 's/most/few/' test.file.txt
5 I think few spring flowers bloom white.
6 Is that caused by gene regulation?
7 $  sed -n 's/most/few/p' test.file.txt
8 I think few spring flowers bloom white.
9 $
```

In Terminal 78 we first create a short text file using `cat` Sect. 7.1.1 on p. 78). In line 4 you find our first Sed editing command. The command itself is enclosed in single quotes: `s/most/few/`. It tells Sed to substitute the first occurrence of *most* with *few*. The slash functions as a delimiter. Altogether, there are four parts to this substitution command:

`s`	Substitute command
`/../../`	Slashes as delimiter
`most`	Regular expression pattern string
`few`	Replacement string

We will look at the substitution command in more detail in Sect. 12.5.1 on p. 182. The editing command is enclosed in single quotes in order to avoid that the shell interprets meta characters. After executing the Sed command in line 4 of Terminal 78 on the facing page the output is immediately printed onto the screen. Note: The input file *test.file.txt* remains unchanged.

Now you know 90 % of Sed. This is no joke. Most people use Sed for substituting one pattern by another; but be aware that the last 10 % are more difficult to achieve than the initial 90 %—always! In line 7 of Terminal 78 on the preceding page Sed is started with the option `-n`. In this case Sed works silently and prints out text only if we tell it to do so. We do that with the command `p` (print). Each line that matches the pattern *most* is edited and printed, all other lines are neither edited nor printed onto the screen. The next terminal shows you that the single character `p` is a command.

Terminal 79: Print Command

```
$ sed -n 'p' test.file.txt
I think most spring flowers bloom white.
Is that caused by gene regulation?
$
```

In the example in Terminal 79 the option `-n` tells Sed not to print any line from the input file unless it is explicitly stated by the command `p`.

Sometimes one likes to apply more than one editing command. That can be achieved with the option `-e`.

Terminal 80: Combining Editing Commands

```
$ sed -e 's/most/few/' -e 's/few/most/' -e 's/white/red/'
      test.file.txt
I think most spring flowers bloom red.
Is that caused by gene regulation?
$
```

All commands must be preceded by the option `-e`. The example in Terminal 80 shows how Sed works through the editing commands. After the first line of text has been read by Sed, the first editing command (the one most to the left) is executed. Then the other editing commands are executed one by one. This is the reason why the word *most* remains unchanged: first it is substituted by *few*, then *few* is substituted by *most*. With more and longer editing commands that looks quite ugly. You are better off writing the commands into a file and telling Sed with the option `-f` that the editing commands are in a file.

Terminal 81: Editing Command File

```
$ cat>edit.sed
s/most/few/
s/few/most/
s/white/red/
$ sed -f edit.sed test.file.txt
I think most spring flowers bloom red.
Is that caused by gene regulation?
$
```

In Terminal 81 we create a file named *edit.sed* and enter the desired commands. Note: Now there are no single quotes or `e`-options! However, each command must reside in its own line.

A good way to test small Sed editing commands is piping a line of text with `echo` to Sed.

Terminal 82: Bash Command echo and Sed

```
$ echo "Darwin meets Newton" | sed 's/meets/never met/'
Darwin never met Newton
$
```

In Terminal 82 the text "*Darwin meets Newton*" is piped to and edited by Sed. With this simple construction one can nicely test small Sed editing scripts.

Now you know 95 % of Sed. Not too bad, is it? Well, the last 5 % are usually even worse to grep than the last 10 %. Joking apart, after this first introduction let us now take a closer look at Sed.

12.3 How Sed Works

In order to develop Sed scripts, it is important to understand how Sed works internally. The main thing one must consider is: what does Sed do to a line of text? Well, first the line is copied from the input file and saved in the *pattern space*. This is Sed's working memory. Then editing commands are executed on the text in the pattern space. Finally, the edited pattern space is sent to the standard output, usually the screen. Furthermore, it is possible to save a line in the *hold space*. Okay, let us examine these "spaces" a bit more closely.

12.3.1 Pattern Space

The pattern space is a kind of working memory where a single text line (also called record) is held, while the editing commands are applied. Initially, the pattern space contains a copy of the first line from the input file. If several editing commands are given, they will be applied *one after another* to the text in the pattern space. This is why we can change back a previous change as shown in Terminal 80 on p. 177 (most → few → most). When all the instructions have been applied, the current line is moved to the standard output and the next line from the input file is copied into the pattern space. Then all editing commands are applied to that line and so forth.

12.3.2 Hold Space

While the pattern space is a memory that contains the current input line, the hold space is a temporary memory. In fact, you can regard it as the memory key of a calculator. You can put a copy from the pattern space into the hold space and recall it later. A group of commands allow you to move text lines between the pattern and the hold space:

`h`	*hold-O*	Overwrites the hold space with the contents of the pattern space.
`H`	*hold-A*	Appends a newline character followed by the pattern space to the hold space.
`g`	*get-O*	Overwrites the pattern space with contents of hold space.
`G`	*get-A*	Appends a newline character followed by the hold space to the pattern space.
`x`	*exchange*	Swaps the contents of the hold space and the pattern space.

You might like to play around with the hold space once you have gotten accustomed to Sed. In the example in Sect. 12.6.3 on p. 192, space is used in order to reverse the lines of a file.

12.4 Sed Syntax

The basic way to initiate Sed is

`sed'EditCommand'Input.File`

This line causes Sed to edit the *Input.File* line by line according to the job specified by *EditCommand*. As shown before, if you want to apply many edit commands sequentially, you have to apply the option `-e`:

`sed -e'Edit.Command.1' -e'Edit.Command.2'Input.File`

Sometimes you wish to save your edit commands in a file. If you do this, Sed can execute the commands from the file (which we would call a script file). Assume the script file's name is *EditCommand.File*. Then you call Sed with

`sed -f EditCommand.File Input.File`

The option `-f` tells Sed to read the editing commands from the file stated next. Instead of reading an input file, Sed can also read the standard input. This can be useful if you want to pipe the output of one command to Sed. We have already seen one example in conjunction with the `echo` command in Terminal 82 on p. 178. A more applied example with `ls` is shown in Terminal 85 on p. 184.

12.4.1 Addresses

As I said above, Sed applies its editing commands to each line of the input file. With addresses you can restrict the range to which editing commands are applied. The complete syntax for Sed becomes

`sed'AddressEditCommand'Input.File`

Note that there is no space between the address and the editing command. There are several ways to describe an address or even a range:

`a`	*a* must be an integer describing a line number.
`$`	The dollar character symbolizes the last line.
`/re/`	*re* stands for a regular expression, which must be enclosed in slashes.
	Each line containing a match for *re* is chosen.
`a,b`	Describes the range of lines from line *a* to line *b*. Note that *a* and *b*
	could also be regular expressions or $. All kinds of combinations are possible.
`x!`	The exclamation mark negates the address range.
	x can be any of the addresses above.

Let us take a look at some examples. We do not edit anything, but just print the text lines to which the addresses match. Therefore, we run Sed with the option `-n`. Sed's command to print a line of text onto the screen is `p` (see Sect. 12.5.6 on p. 189). As an example file let us use the protein structure file *structure.pdb* (File 4 on p. 161). In order to display the whole file content we use

`sed -n'p' structure.pdb`

Do you recognize that this command resembles "`cat structure.pdb`"? What would happen if we omitted the option `-n`? Then all lines would be printed twice! Without option `-n` Sed prints every line by default *and* additionally prints all lines it is told to by the print command `p`—ergo, each line is printed twice.

Terminal 83: Addresses

```
1  $ sed -n '1p' structure.pdb
2  HEADER  Hydrogenase                          23-Mar-99   1CFZ
3  $ sed -n '2p' structure.pdb
4  COMPND  Hydrogenase Maturating Endopeptidase Hybd From
5  $ sed -n '1,3p' structure.pdb
6  HEADER  Hydrogenase                          23-Mar-99   1CFZ
7  COMPND  Hydrogenase Maturating Endopeptidase Hybd From
8  SOURCE  ORGANISM_SCIENTIFIC: Escherichia coli
9  $ sed -n '/ATOM 7/,$p' structure.pdb
10 ATOM 7  O   MET A 1 47.875  25.011  52.020  1.00 54.99 O
11 ATOM 8  SD  MET A 1 49.731  23.948  47.163  1.00 77.15 S
12
13 $ sed -n '/HELIX/p' structure.pdb
14 HELIX  1 hel ILE A   18  GLN A   28
15 HELIX  2 hel PRO A   92  THR A  107
16 HELIX  3 hel ILE A  138  SER A  152
17 $ sed -n '/HEADER/,/AUTHOR/p' structure.pdb
18 HEADER  Hydrogenase                          23-Mar-99   1CFZ
19 COMPND  Hydrogenase Maturating Endopeptidase Hybd From
20 SOURCE  ORGANISM_SCIENTIFIC: Escherichia coli
21 AUTHOR  Fritsche, Paschos, Beisel, Boeck & Huber
22 $ sed -n '/HEADER\|AUTHOR/p' structure.pdb
23 HEADER  Hydrogenase                          23-Mar-99   1CFZ
24 AUTHOR  Fritsche, Paschos, Beisel, Boeck & Huber
25 \$
```

Terminal 83 gives you a couple of examples how to extract certain lines from File 4 on p. 161. Lines 1–8 show examples how to select and display certain lines or line ranges. The address used in line 9 makes Sed print all lines from the first occurrence of "*ATOM 7*" to the last line (which is, in our case, an empty line). The address `/HELIX/` in line 13 instructs Sed to print all lines containing the word *HELIX*. In line 22 Sed selects all lines containing either *HEADER* or *AUTHOR*. Note that the logical *or* represented by the vertical bar needs to be escaped (`\|`).

It is important that you understand that the address *selects* the lines of text that are treated by Sed. In Terminal 83 the treatment consisted simply of printing. The treatment could as well have been an editing statement. For example,

`sed'/^ATOM/s/.*/---deleted---/'structure.pdb`

would select all lines starting with *ATOM* (`/^ATOM/`) for editing. The editing instruction (`s/.*/---deleted---/`) selects all the content of the selected lines (`.*`) and substitutes it with the text "*--- deleted ---*". Try it out!

12.4.2 Sed and Regular Expressions

Of course, Sed works fine with regular expressions. However, in contrast to the meta characters we used with `egrep` in Chap. 11 on p. 157, there are some syntactical differences: Parentheses "( )", braces "{ }" and the vertical bar "|" must be escaped with a backslash! You remember that parentheses group commands (see Sect. 11.4.3 on p. 165) and save the target pattern for back referencing (see Sect. 11.4.7 on p. 165), whereas braces belong to the quantifiers (see Sect. 11.4.2 on p. 164).

12.5 Commands

Up to now, we have already encountered some vital Sed commands. We came across substitutions with `'s/.../.../'` and know how to explicitly print lines by the combination of the option `-n` and the command `p`. We also learned that the commands `g`, `G`, `h`, `H` and `x` manipulate the pattern space and hold space. Let us now take a tour through the most important Sed commands and their application. For some of the following examples we are going to use a new text file, called *GeneList.txt* (File 6).

File 6: GeneList.txt

```
1  Energy metabolism
2     Glycolysis
3        slr0884: glyceraldehyde 3-phosphate dehydrogenase (gap1)
4        Init: 1147034 Term: 1148098 Length (aa): 354
5        slr0952: fructose-1,6-bisphosphatase (fbpII)
6        Init: 2022028 Term: 2023071 Length (aa): 347
7  Photosynthesis and respiration
8     CO2 fixation
9        slr0009: ribulose bisphosphate carboxylase large (rbcL)
10       Init: 2478414 Term: 2479826 Length (aa): 470
11       slr0012: ribulose bisphosphate carboxylase small (rbcS)
12       Init: 2480477 Term: 2480818 Length (aa): 113
13    Photosystem I
14       slr0737: photosystem I subunit II (psaD)
15       Init: 126639 Term: 127064 Length (aa): 141
16       ssr0390: photosystem I subunit X (psaK)
17       Init: 156391 Term: 156651 Length (aa): 86
18       ssr2831: photosystem I subunit IV (psaE)
19       Init: 1982049 Term: 1982273 Length (aa): 74
20    Soluble electron carriers
21       sll0199: plastocyanin (petE)
22       Init: 2526207 Term: 2525827 Length (aa): 126
23       sll0248: flavodoxin (isiB)
24       Init: 1517171 Term: 1516659 Length (aa): 170
25       sll1796: cytochrome c553 (petJ)
26       Init: 846328 Term: 845966 Length (aa): 120
27       ssl0020: ferredoxin I (petF)
28       Init: 2485183 Term: 2484890 Length (aa): 97
```

This file contains information about the genome of the cyanobacterium *Synechocystis* PCC 6803. It was sequenced in 1996. The file is a very, very small exemplified assembly of annotated genes (gene names are given in the unambiguous *Cyanobase Format*, like *slr0884*, and, in parentheses, with their scientific specification), their position on the chromosome (*Init* and *Term*), the predicted length of the gene products (aa = amino acids) and their function. The file contains two major functional classes, *energy metabolism* and *photosynthesis and respiration*, respectively. These major classes are divided into subclasses, such as *glycolysis*, CO_2 *fixation*, *photosystem I* and *soluble electron carriers*.

12.5.1 Substitutions

Substitutions are by far the most common, and probably most useful, editing actions Sed is used for. An example of a simple substitution is the transformation of all decimal markers from commas to points. The basic syntax of the substitution command is

`s/pattern/replacement/flags`

The *pattern* is a regular expression describing the target that is to be substituted by *replacement*. The *flags* can be used to modify the action of the substitution command in one or the other way. Important flags are:

`g`	Make the changes globally on all occurrences of the pattern. Normally only the first occurrence is edited.
`n`	The replacement should be made only for the *n*th occurrence of the target pattern.
`p`	In conjunction with the option `-n`, only matched and edited lines (pattern space) are printed.
`w` *file*	Same as `p`, but writes the pattern space into a file named *file*.

The flags can be used in combinations. For example, the flag `gp` would make the substitution globally on the line and print the line.

The *replacement* can make use of *back referencing* (see Sect. 11.4.7 on p. 168). Furthermore, the ampersand (`&`) stands for (and is replaced by) the string that matched the complete regular expression.

Terminal 84: Back References

```
1  $ sed -n 's/^   [A-Z].*$/>>>&/p' GeneList.txt
2  >>>   Glycolysis
3  >>>   CO2 fixation
4  >>>   Photosystem I
5  >>>   Soluble electron carriers
6  $ sed -n 's/^   \([A-Z].*$\)/--- \1 ---/p' GeneList.txt
7  --- Glycolysis ---
8  --- CO2 fixation ---
9  --- Photosystem I ---
10 --- Soluble electron carriers ---
11 $
```

In Terminal 84 we see two examples of back references. In both cases we use the combination of the option `-n` and the flag `p` to display modified lines only. The target match for the search pattern "`^␣␣␣[A-Z].*$`" is any line that begins with three space characters (here symbolized by "␣"), continues with one uppercase alphanumeric character and continues with any or no character up to the end of the line. This search pattern exactly matches the subclasses in *GeneList.txt* (File 6 on p. 182). In line 1 of Terminal 84 the matching pattern is substituted by "`>>>`" plus the whole matching pattern, which is represented by the ampersand character (`&`). In line 6 we embedded only the last part of the search pattern in parentheses (note that the parentheses need to be escaped) and excluded the three spaces. The matching pattern can then be recalled with "`\1`".

In order to restrict the range of the editing action to the first 6 lines of File 6 on p. 182, line 6 of Terminal 84 could be modified to

sed − n′1, 6s/ ^\textvisiblespace\textvisiblespace\textvisiblespace\([A − Z]. ∗ $\)/ − \textvisiblespace\1\textvisiblespace − /p′ GeneList. txt

Now let us assume we want to delete the text parts describing the position of genes on the chromosome. The correct command would be

sed′s/Init. ∗ Term: \textvisiblespace[0 − 9] ∗ \textvisiblespace/ /′GeneList. txt

Here, we substitute the target string with an empty string. Try it out in order to see the result.

The next example is a more practical one. In order to see the effect of the Sed command in Terminal 85 you must have subdirectories in your home directory.

Terminal 85: Piping to Sed

```
1 $ ls -l ~ | sed 's/^d/DIR: /' | sed 's/^[^Dt]/     /'
2 total 37472
3      rw-rw-r-- 1 Freddy Freddy     30 May  5 16:29 Datum2
4      rw-rw-r-- 1 Freddy Freddy     57 May 11 19:00 amino2s
5 DIR: rwxrwxr-x 3 Freddy Freddy   4096 May  1 20:04 blast
6 DIR: rwxrwxr-x 3 Freddy Freddy   4096 May  1 20:34 clustal
7      rw-rw-r-- 1 Freddy Freddy 102400 Apr 19 15:55 dat.tar
8 $
```

Terminal 85 shows how to use Sed in order to format the output of the `ls` command. By applying the editing commands given in line 1, the presence of directories immediately jumps to the eye. With "`ls -l ~`" we display the content of the home directory [Remember that the tilde character (`~`) is the shell shortcut for the path of your home directory]. In the first Sed editing command (`s/^d/DIR: /`) we substitute all occurrences of `d` at the beginning of a line (`^`) with "`DIR: `". The result is piped to a second instance of Sed which introduces spaces at the beginning of each line *not* starting with a "`D`" (as in "DIR") or "`t`" (as in "total..."). With our knowledge about writing shell scripts and the `alias` function (see Sect. 8.8 on p. 104), we could now create a new standard output for the `ls` command.

Program 27: list-dir.sh - Format ls output

```
$ cat list-dir.sh
#!/bin/bash
# save as list-DIR.sh
# reformats the output of ls -l
echo "'ls -l $1|sed 's/^d/DIR: /'|sed 's/^[^Dt]/     /'"
```

An appropriate shell script is shown in Program 27. The shell script is executed with

`./list-DIR.sh DirName`

from the directory where you saved it. The parameter *DirName* specifies the directory of which you wish to list the content. How does the script work? We basically put the command line from Terminal 85 into a shell script and execute the command from within the program by enclosing it in graves (see Sect. 10.5.2 on p. 136). The parameter *DirName* can be accessed via the variable *$1*.

12.5.2 Transliterations

You have got a nice sequence file from a colleague. It contains important sequence data, which you want to process with a special program. The stupid thing is that the sequences are lowercase RNA sequences. However, you require uppercase DNA sequences. Hmmm—what you need is Sed's transliteration command "`y`".

Terminal 86: Transliterations with y/.../.../

```
1  $ cat>rna.seq
2  >seq-a
3  acgcguauuuagcgcaugcgaauaucgcuauuacg
4  >seq-b
5  uagcgcuauuagcgcgcuagcuaggaucgaucgcg
6  $ sed '/>/!y/acgu/ACGT/' rna.seq
7  >seq-a
8  ACGCGTATTTAGCGCATGCGAATATCGCTATTACG
9  >seq-b
10 TAGCGCTATTAGCGCGCTAGCTAGGATCGATCGCG
11 $
```

Terminal 86 shows a quick solution to the problem. In lines 1–5 we create our example RNA sequence file in *FASTA format*. The magic command follows in line 6. First we define the address (`/>/!`): all lines *not* containing the greater character are to be edited. The editing command invoked by "`y/acgu/ACGT/`" transliterates all "a"'s to "A"'s and so on. The transliteration command "`y`" does not understand words. The first character on the left side (a) is transliterated into the first character on the right side (A) and so on. It always affects the whole line.

Now let us generate the complement of the sequences in the file *rna.seq*. The correct comment is

`sed'/>/!y/acgu/ugca/'rna.seq`

Could you achieve the same result with the substitution command from Sect. 12.5.1 on p. 182? If you think so and found a solution, please email it to me.

12.5.3 Deletions

We have used already the substitution command to delete some text. Now we will delete a whole line. Assume you want to keep only the classes and subclasses but not the gene information in File 6 on p. 182 (*GeneList.txt*). This means we need to delete these lines. The corresponding command is `d` (delete), the syntax is

`addressd`

The `d` immediately follows the address. The address can be specified as described in Sect. 12.4.1 on p. 180. If no address is supplied, all lines will be deleted—that sounds not too clever, doesn't it? The important thing to remember is that the whole line matching the address is deleted, not just the match itself. Let us come back to our task: extracting the classes and subclasses of *GeneList.txt* (File 6 on p. 182).

Terminal 87: Deletions

```
$ sed '/     /d' GeneList.txt
Energy metabolism
   Glycolysis
Photosynthesis and respiration
   CO2 fixation
   Photosystem I
   Soluble electron carriers
$
```

Since all gene information lines are indented by 5 space characters, these 5 spaces form a nice address pattern. Thus, the task is solved easily, as shown in Terminal 87.

12.5.4 Insertions and Changes

Another common editing demand is to insert, append, change or delete text lines. Note that only whole lines and no words or other fragments can be inserted, appended or changed. These commands require an address. The corresponding commands are

a	*append*	Append a text after the matching line.
i	*insert*	Insert a text before the matching line.
c	*change*	Change the matching line.

Append and *insert* can only deal with a single line address. However, for *change*, the address can define a range. In that case, the whole range is changed. In contrast to all other commands we have encountered up to now, `a`, `i` and `c` are multiple-line commands. What does this mean? Well, they require a line break. For example, the syntax for the append command `a` is

```
addressa\
text
```

As you can see, the command is followed by a backslash without any space character between the address and the command! That is true for all three commands. Then follows the text that is to be appended after all lines matching the address. To insert multiple lines of text, each successive line must end with a backslash, except the very last text line.

Terminal 88: Insertions

```
1 $ sed '/>/i\
2 > ---------' rna.seq
3 ---------
4 >seq-1
5 acgcguauuuagcgcaugcgaauaucgcuauuacg
6 ---------
7 >seq-2
8 uagcgcuauuagcgcgcuagcuaggaucgaucgcg
9 $
```

Terminal 88 gives an example of an insertion. The trick with multiple-line commands is that the shell recognizes unclosed quotes. Thus, when you press Enter after entering line 1 of Terminal 88 the greater character (>) appears and you are supposed to continue your input. In fact, you can enter more lines, until you type the single quote again. This we do in line 2: after our insertion text (a number of dashes) we close Sed's command with the closing single quote followed by the filename. In this example we use the file we created in Terminal 86 on p. 185. What does the command spanning from line 1 to line 2 do? Before every line beginning with a greater character some dashes are inserted.

We can do fancy things with multiple-line commands.

Terminal 89: Deletions and Insertions

```
1  $ sed '/     /d
2  > /^[A-Z]/a\
3  > ========================
4  > /   /a\
5  > ------------------------
6  > ' GeneList.txt
7  Energy metabolism
8  ========================
9     Glycolysis
10 ------------------------
11 Photosynthesis and respiration
12 ========================
13    CO2 fixation
14 ------------------------
15    Photosystem I
16 ------------------------
17    Soluble electron carriers
18 ------------------------
19 $
```

Terminal 89 gives you an idea of how several commands can be used with one call of Sed. Line 1 contains the editing commands to delete all lines containing five successive space characters. This matches the gene-information lines of our example File 6 on p. 182 (*GeneList.txt*). The address in line 2 matches all main classes. A line containing equal characters (=) is appended to this match. Then, to all lines containing three consecutive space characters, a line with dashes is appended. The editing action is executed after hitting Enter in line 6. What follows in Terminal 89 is the resulting output.

Let us finally take a quick look at the *change* command.

Terminal 90: Changing Text Blocks

```
$ sed '/Photo/,$c\
> stuff deleted' GeneList.txt
Energy metabolism
   Glycolysis
      slr0884: glyceraldehyde 3-P dehydrogenase (gap1)
      Init: 1147034 Term: 1148098 Length (aa): 354
      slr0952: fructose-1,6-bisphosphatase (fbpII)
      Init: 2022028 Term: 2023071 Length (aa): 347
stuff deleted
$
```

In the example shown in Terminal 90 all lines between the first occurrence of "Photo" and the last line is changed to (substituted by) the text "stuff deleted".

12.5.5 Sed Script Files

If you need to apply a number of editing commands several times, you might prefer to save them in a file.

Program 28: script1.sed - Sed File

```
1 # save as script1.sed
2 # Formatting of GeneList.txt
3 /     /d
4 /^[A-Z]/a\
5 ========================
6 /   /a\
7 ------------------------
```

The script file containing the commands used in Terminal 89 on the preceding page is shown in Program 28. Lines 1 and 2 contain remarks that are ignored by Sed. You would call this script by

`sed -f script1.sed GeneList.txt`

Maybe you are using the script very, very often. Then it makes sense to create an executable script.

Program 29: script2.sed - Sed Executable

```
sed '
# save as script2.sed
# Provide filename at the command line
/     /d /^[A-Z]/a\
========================
/   /a\
------------------------
' $*
```

Program 29 would be the result of this approach. Do not forget to make the file executable with "`chmod u+x script2.sed`". Note that all commands are enclosed in single quotes. Do you recognize the variable "*$**"? Take a look at Sect. 10.4.4 on p. 133. – It contains all command line parameters. In our case these would be the files that are to be edited—yes, you can edit several files at once. In order to execute the same task as shown in Terminal 89 you type

`./script2.sed GeneList.txt`

This requires much less typing, especially if you create an *alias* for the program (see Sect. 8.8 on p. 104)!

12.5.6 Printing

We have already used the printing command `p` a couple of times. Still, I will shortly mention it here. Unless the default output of Sed is suppressed (`-n`), the print command will cause duplicate copies of the line to be printed. The print command can be very useful for debugging purposes.

Terminal 91: Debugging

```
1 $  sed '/>/p
2 > s/>/</' rna.seq
3 >seq-1
4 <seq-1
5 acgcguauuuagcgcaugcgaauaucgcuauuacg
6 >seq-2
7 <seq-2
8 uagcgcuauuagcgcgcuagcuaggaucgaucgcg
9 $
```

Terminal 91 shows how to use `p` for debugging. In line 1 the current line in the pattern space is printed. Line 2 modifies the pattern space and prints it out again. Thus, we see the line before and after editing and might detect editing errors.

A special case of printing provides the equal character (`=`). It prints the line number of the matching line.

Terminal 92: Printing Line

```
$ sed -n '/^[A-Z]/=' GeneList.txt
1
7
$
```

The editing command in Terminal 92 prints the line number of lines containing major classes in *GeneList.txt* (File 6 on p. 182).

12.5.7 Reading and Writing Files

The read (`r`) and write (`w`) commands allow you to work directly with files. This can be very comfortable when working with script files. Both commands need a single argument: the name of the file to read from or write to, respectively. There must be a single-space character between the command and the filename. At the end of the filename, either a new line must start (in script files) or the script must end with the single quote (as in Terminal 93).

Terminal 93: Writing into File

```
$ sed -n '/^   [A-Z].*/w output.txt' GeneList.txt
$ cat output.txt
   Glycolysis
   CO2 fixation
   Photosystem I
   Soluble electron carriers
$
```

The editing command in Terminal 93 writes the subclasses from File 6 on p. 182 into the file *output.txt*.

Terminal 94: Reading a File

```
1 $ cat>input.txt
2    sub classes...
3 $ sed '/^[A-Z]/r input.txt
4 > /   /d' GeneList.txt
5 Energy metabolism
6    sub classes...
7 Photosynthesis and respiration
8    sub classes...
9 $
```

In Terminal 94 we first create an input file called *input.txt*. In line 3 the file *input.txt* is inserted after each occurrence of a main class in File 6 on p. 182. Note that there must be exactly one space character between "`r`" and the filename, and that there must not be anything behind the filename except a single quote or a new line. In line 4 all other than the main class lines are deleted.

The read command will not complain if the file it should read does not exist. The write command will create a non-existing file and overwrite an existing file. However, write commands *within* one single script will append to the target file!

12.5.8 Advanced Sed

As you can imagine, there are many more possibilities for what you can do with Sed. There are some more advanced commands that allow you to work with multiple-line pattern spaces, make use of the hold space or introduce conditional branches. However, I will not go into details here. These commands require quite some determination to master and are pretty difficult to learn. In my opinion, what you have learned up to this point is by far enough to have fun with Sed. All the advanced stuff starts to become a pain in the neck. Therefore, we save our energy in order to learn how to use AWK and Perl for these more advanced editing tasks. Anyway, if you feel Sed is just the scripting language you have been waiting for, you should read some special literature in either English (Dougherty and Robbins 1997) or German (Herold 2003).

12.6 Examples

You should keep up the habit of trying out some example scripts. Do some modification and follow the changes in the output.

12.6.1 Gene Tree

The following examples reformats the content of File 6 on p. 182 (*GeneList.txt*) into a tree-like output. This facilitates easy reading of the data.

Terminal 95: Creating a Gene Tree-View

```
1  $ sed -n 's/^\([A-z]..*\)/|--- \1/p
2  s/^   \([A-Z]..*\)/|   |--- \1/p
3  s/^      [A-Za-z]..*(\(....\))/|   |   |--- \1/p
4  ' GeneList.txt
5  |--- Energy metabolism
6  |   |--- Glycolysis
7  |   |   |--- gap1
8  |--- Photosynthesis and respiration
9  |   |--- CO2 fixation
10 |   |   |--- rbcL
11 |   |   |--- rbcS
12 |   |--- Photosystem I
13 |   |   |--- psaD
14 |   |   |--- psaK
15 |   |   |--- psaE
16 |   |--- Soluble electron carriers
17 |   |   |--- petE
18 |   |   |--- isiB
19 |   |   |--- petJ
20 |   |   |--- petF
21 $
```

The Sed editing commands in Terminal 95 largely resemble those in Terminal 84 on p. 183. A speciality resides in line 3 of Terminal 95: here we use the back reference in order to grep the content of the parentheses, that is gene name. You might have realized that the script makes use of the `-n` option and `p` command combination. This makes it easy to exclude the lines starting with "Init...".

12.6.2 File Tree

Since we started with trees, why not writing a scripting in order to print the directory content in a tree-like fashion?

Program 30: tree.sed - Directory Tree

```
1  #!/bin/bash
2  # save as tree.sed
3  # requires path as command line parameter
4  if [ $# -ne 1 ]; then
5    echo "Provide one directory name next time!"
6    echo
7  else
8    find $1 -print 2>/dev/null |
9    sed -e 's/[^\/]*\//|--- /g' -e 's/--- |/     |/g'
10 fi
```

What does Program 30 do? Well, first you should recognize that it is a shell script. In line 4 we check whether exactly one command line parameter has been supplied. If this is not the case, the message "Provide one directory name next time!" is displayed and the program stops. Note: It is always a good idea to make a script *fool-proof* and inform the user about the correct syntax when he applies the wrong syntax. Otherwise, the `find` command is invoked with the directory name provided at the command line (*$1*). The `find`option `-print` prints the full filename, followed by a new line, on the standard output. Error messages are redirected into the nirvana (`2 > /dev/null`, see Sect. 8.5 on p. 101). The standard output, however, is not printed onto the screen but redirected (`|`) to Sed. In order to understand what Sed is doing, I recommend you to look at the output of `find` and apply the search pattern "`[^\/]*\/`" to it. How? Use Vim! In Sect. 7.2.8 on p. 92. we saw that we can read external data into an open editing file. If you type

`:r! find . -print`

in the command modus (and then hit Enter), the output of "`find . print`" is imported into Vim. Then you can play around with regular expressions and see what the search pattern is doing, as described in Sect. 11.6.1 on p. 172.

12.6.3 Reversing Line Order

Assume you have a file with lists of parameters in different lines. For reasons I do not know, you want to reverse the order of the file content. Ahh, I remember the reason: in order to exercise!

Terminal 96: Reversing Lines

```
1  $ cat > parameterfile.txt
2  Parameter 1 = 0.233
3  Parameter 2 = 3.899
4  Parameter 3 = 2.230
5  $ sed '1!G
6  > h
7  > $!d' parameterfile.txt
8  Parameter 3 = 2.230
9  Parameter 2 = 3.899
10 Parameter 1 = 0.233
11 $
```

In lines 1–4 of Terminal 96 we create the assumed parameter file, named *parameterfile.txt*. The Sed script consists of three separate commands:"`1!G`", "`h`" and "`$!d`". You see, herewe make use of the hold space, as explained in Sect. 12.3.2 on p. 179. The first command in line 5 is applied to all but the first line (`1!`), whereas the third command in line 7 is applied to all but the last line (`$!`). The second command is executed for all lines. When we execute the Sed script on the text file *parameterfile.txt*, the first command that is executed is `h`. This tells Sed to copy the contents of the pattern space (the buffer that holds the current line being worked on) to the hold space (the temporary buffer). Then, the `d` command is executed, which deletes the current line from the pattern space. Next, line 2 is read into the pattern space and the command `G` is executed. `G` appends the contents of the hold space (the previous line) to the pattern space (the current line). The `h` command puts the pattern space, now holding the first two lines in reverse order, back to the hold space for safe keeping. Again, `d` deletes the line from the pattern space so that it is not displayed. Finally, for the last line, the same steps are repeated, except that the content of the pattern space is not deleted (due to `$!`)' before the 'd'). Thus, the contents of the pattern space is printed to the standard output.

Note that you could execute the script also with

`sed'1!G;h;$!d'parameterfile.txt`

In this case, the commands are separated by the semicolon instead of the newline character.

12.7 Exercises

Now let us see what you have learned. All exercises are based on the File 4 on p. 161 (*structure.pdb*).

12.1 Change Beisel's name to Weisel.

12.2 Delete the first three lines of the file.

12.3 Print only lines 5 through 10.

12.4 Delete lines containing the word MET.

12.5 Print all lines where the HELIX line contains the word ILE.

12.6 Append three stars to the end of lines starting with "H".

12.7 Replace the line containing SEQRES with SEQ.

12.8 Create a file with text and blank lines. Then delete all blank lines of that file.

Copyright information