```
ygrex@ygrex:~$ mkdir sed-book
ygrex@ygrex:~$ cd !$
cd sed-book
ygrex@ygrex:~/sed-book$ git init
Initialized empty Git repository in ~/sed-book/.git/
ygrex@ygrex:~/sed-book [sed-book:]$ git config user.name Ygrex
ygrex@ygrex:~/sed-book [sed-book:]$ git config user.email ygrex@ygrex.ru
ygrex@ygrex:~/sed-book [sed-book:]$ ed sed-book.tex

sed-book.tex: No such file or directory
a
\documentclass{article}
\title{Sed Book}
\author{Ygrex}
\begin{document}
\maketitle
\end{document}
.
w
99
```

Sed Book

Ygrex

February 19, 2020

Contents

Chapter 1

Forewords

1.1 Introduction

This book is about a streaming editing programming with modern development techniques.

What exactly modern techniques are, will be discussed in details later. Just brief examples here will give a hint for the beginning: structuring code in functions, classes, modules, projects, covering program with unit-tests, versioning and distribution. All techniques are trivial in context of nowadays mainline programming languages, but counter-intuitive otherwise. This book presents few possible approaches to those trivialities with certainly “otherwise” language — GNU Sed.

Chapter 2

Simple Sed

2.1 What is Sed

Sed in this book refers to GNU Sed most of the time. GNU Sed is a POSIX compliant version of Sed with GNU specific extensions distributed under conditions of GPL.

GNU extensions are used extensively across the book, so that other versions of Sed unlikely will give similar result.

Simple and intended way of using Sed is processing text line-by-line in a command line or in shell scripts. Quick example is extracting top-level domains from a list of FQDNs:

```
$ printf '%s\n' host.com host.org localhost.com |
        sed 's/^.*\.//'
com
org
com
```

Next example features filtering out wrong domain names. For simplicity wrong domain names are those words that contain no dot:

Listing 2.1: Simple filtering

```
$ printf '%s\n' not-a-domain host.com host.org localhost.com |
        sed -n 's/^.*\.//p'
com
org
com
```

Here `not-a-domain` is the wrong domain which gets filtered out.

These two examples are quite simple and probably familiar to the reader, but here are a few Sed concepts worth highlighting very the more involved examples are introduced.

2.1.1 One-line greediness

Pattern from the first example has assumption about the greediness: `/^.*\./`

It is important for multi-level domain names, for example `www.example.com`. It contains tow substrings matching the pattern (i.e. a substring spanning from the beginning of a line till a dot character): `www.` and `www.example.`, which one matches from the whole string depends on the greediness. Sed has only greedy matching, in other words the longest possible substring selected and filtered out after being substituted with an empty string:

Listing 2.2: Simple greedy matching

```
$ echo www.example.com | sed 's/^.*\.//'
com
```

Let's assume for a moment that a non-greedy matching desired, for the above examples it would mean matching on the left-most domain name segment. The common approach for single line patterns is to replace "everything matcher" `/.*/` with "matching everything but a separator" `/[^.]*/`. It might be confusing to see a dot character in two meanings here: as a literal dot, that separates domain name segments, and a special pattern matching against any character. Both meanings must be clearly understood differently as having no common semantics. The resulting non-greedy pattern now looks like:

Listing 2.3: Simple non-greedy matching

```
$ echo www.example.com | sed 's/^[^.]*\.//'
example.com
```

This approach is the most simple one. But for sure it has its own limitations and not universally applicable. In particular, meaning of `[^.]*` changes in multi-line mode (controlled by `m` or `M` modifiers).

Just one more example to achieve a non-greedy matching from the left will start with a greedy matching from the right:

```
$ echo www.example.com | sed 's/\..*$/\n&/'
www
.example.com
```

Here the input string transformed into two lines, first one contains exactly the same substring as matched above with a non-greedy pattern `/^[^.]*/`. The later is to filter it out:

```
$ echo www.example.com | sed 's/\..*$/\n&/ ; s/^.*\n//'
.example.com
```

and to get rid of the leading separator (dot character):

```
$ echo www.example.com | sed 's/\..*$/\n&/ ; s/^.*\n\.//'
example.com
```

Now the result is the same as in the simple example above 2.3.

One more non-greedy example, which will be useful going forward, is to replace first separator by a special character which is not expected to be used anywhere else in the input:

```
$ echo www.example.com | sed 's/\./\x01/'
wwwexample.com
```

Here `\x01` is a non-printable character, which used for that purpose. The next is to run exactly the same 2.2 substitution command but with this special separator:

```
$ echo www.example.com | sed 's/\./\x01/ ; s/^.*\x01//'
example.com
```

Note the result is identical to the previous one 2.3.

2.1.2 Pattern space

As it happens all the time when working in a world of regular expressions, there is one more overloaded term. The pattern space is a special text buffer in Sed. All text transformation can only happen in this buffer.

Rules for the pattern space are:

- Every Sed script has an input data. On the first execution cycle the pattern space contains first line of the input excluding newline (newline means a line separator which is parameterized).

  ```
  $ printf '%s\n' first second | sed 's/^first$/&/p ; Q'
  first
  ```

- Input might have no trailing newline, it is indistinguishable from Sed script:

  ```
  $ printf 'abc' | sed -n 's/^// ; l ; Q'
  abc$

  $ printf 'abc\n' | sed -n 's/^// ; l ; Q'
  abc$
  ```

  with one exception, when input is empty:

  ```
  $ printf '' | sed -n 's/^// ; l ; Q'
  ```

  in this case, the script does not execute at all.

- Every subsequent cycle receives next input line in the pattern space. Any data from the previous cycle left in the pattern space gets replaced entirely, if not explicitly instructed otherwise.

- If instructed explicitly, data from the pattern space might be partially passed down to the next cycle and optionally mixed with next input line.
- It is also allowed to append next input line into the pattern space and proceed execution of the current cycle.

Last two points are of limited use for there are conditions depending on the data itself. Trying to pass down part of the pattern space might occasionally result in reading input, while reading input might occasionally interrupt the execution of the whole script. For starts it is reasonable to focus on first three rules and note that pattern space usually does not keep data between execution cycles.

That is how data arrives to the pattern space. The next aspect is sending this data to the output stream. It is not worth mentioning unless an auto-printing mode is enabled. Since it is enabled by default and the output stream is the only valuable result of the Sed script execution most of the time, it is important to track state of the pattern space at the time it gets sent to the output stream. In auto-printing mode it happens at the end of each cycle and implicitly with some commands (quitting command `q` is one notable example). Things are more simple if printing pattern space is controlled explicitly, all but very simple examples in this book will have the auto-printing off.

One more note to the pattern space and output stream interaction will be important if one expects Sed to be a tool for binary data processing (which is not). As stated above, it cannot be known from inside the script itself, if the input line has a trailing newline or not. That is true, but output will have the trailing newline depending on its presence in the input:

```
$ echo 'sed' | sed -n 's/^...$/success/p ; Q'
success

$ printf 'sed' | sed -n 's/^...$/success/p ; Q' | wc -c
7

$ printf 'sed\n' | sed -n 's/^...$/success/p ; Q' | wc -c
8
```

Any output command behaves the same way: `p`, `P`, `w`, `W` and implicit printing. Unambiguous printing `l` ignores the distinction consistently though, it always appends `$` and newline, but it is hardly useful for anything but debugging.

With that all said about the pattern space and printing, let's have a closer look to the above filtering example 2.1. First of all, it runs in a mode with auto-printing off. The result is only assembled by calling printing command `p` explicitly. In this case the command is called indirectly by substitution `s` whenever it succeeds. On its turn it succeeds whenever the substitution pattern `/^.*\./` matches against the text in the pattern space. The pattern space in this particular example on each execution cycle receives one word from list: `not-a-domain`, `host.com`, `host.org` or `localhost.com`. Those words, which do not match the substitution pattern, are skipped by the substitution command and the next execution cycle starts with the next input word placed in the pattern space. This is the filtering part.

Those words, which happen to match the substitution pattern, get transformed (the greedily matching substring removed by this). Transformation alters the content of the pattern space, data gets entirely replaced by the substitution result. After that, substitution continues to printing because of `p` flag (it is not a command but a flag in this context) and follows to the next execution cycle then.