

# Advanced `bash` scripting

(block course)



Michael F. Herbst  
`michael.herbst@iwr.uni-heidelberg.de`  
<http://blog.mfhs.eu>

Interdisziplinäres Zentrum für wissenschaftliches Rechnen  
Ruprecht-Karls-Universität Heidelberg

24<sup>th</sup> – 28<sup>th</sup> August 2015

# Contents

<b>Contents</b>	<b>i</b>
<b>List of Tables</b>	<b>iv</b>
<b>Course description</b>	<b>v</b>
Learning targets and objectives . . . . .	v
Prerequisites . . . . .	vi
Compatibility of the exercises . . . . .	vi
<b>Errors and feedback</b>	<b>vi</b>
<b>Licensing and redistribution</b>	<b>vi</b>
<b>1 Introduction to Unix-like operating systems</b>	<b>1</b>
1.1 The Unix philosophy . . . . .	1
1.1.1 Impact for scripting . . . . .	2
1.2 The Unix utilities . . . . .	2
1.2.1 Accessing files or directories . . . . .	3
1.2.2 Modifying files or directories . . . . .	3
1.2.3 Getting or filtering file content . . . . .	3
1.2.4 Other . . . . .	5
1.2.5 Exercises . . . . .	6
1.3 The Unix file and permission system . . . . .	7
1.3.1 What are files? . . . . .	7
1.3.2 Unix paths . . . . .	8
1.3.3 Unix permissions . . . . .	8
<b>2 A first look at the bash shell</b>	<b>10</b>
2.1 Historic overview . . . . .	10
2.1.1 What is a shell? . . . . .	10
2.1.2 The Bourne-again shell . . . . .	10
2.2 Handy features of the <b>bash</b> . . . . .	11
2.2.1 Tab completion . . . . .	11
2.2.2 Accessing the command history . . . . .	11
2.2.3 Running multiple commands on a single line . . . . .	13
2.3 Redirecting command input/output . . . . .	13
2.4 The exit status of a command . . . . .	16
2.4.1 Logic based on exit codes: The operators <b>&amp;&amp;</b> , <b>  </b> , <b>!</b> . . . . .	17
2.5 Tips on getting help . . . . .	19

<b>3</b>	<b>Simple shell scripts</b>	<b>21</b>
3.1	What makes a shell script a shell script? . . . . .	21
3.1.1	Executing scripts . . . . .	21
3.1.2	Scripts and <i>stdin</i> . . . . .	22
3.2	Shell variables . . . . .	22
3.2.1	Special parameters . . . . .	24
3.2.2	Command substitution . . . . .	25
3.3	Escaping strings . . . . .	27
3.4	Word splitting and quoting . . . . .	28
<b>4</b>	<b>Control structures and Input/Output</b>	<b>32</b>
4.1	Printing output with <b>echo</b> . . . . .	32
4.2	The <b>test</b> program . . . . .	32
4.3	Conditionals: <b>if</b> . . . . .	34
4.4	Loops: <b>while</b> . . . . .	37
4.5	Loops: <b>for</b> . . . . .	41
4.5.1	Common “types” of <b>for</b> loops . . . . .	42
4.6	Conditionals: <b>case</b> . . . . .	45
4.7	Parsing input using shell scripts . . . . .	47
4.7.1	The <b>read</b> command . . . . .	47
4.7.2	Scripts have shared <i>stdin</i> , <i>stdout</i> and <i>stderr</i> . . . . .	48
4.7.3	The <b>while read line</b> paradigm . . . . .	50
4.8	Influencing word splitting: The variable <b>IFS</b> . . . . .	53
4.9	Conventions when scripting . . . . .	55
4.9.1	Script structure . . . . .	56
4.9.2	Input and output . . . . .	56
4.9.3	Parsing arguments . . . . .	56
<b>5</b>	<b>Arithmetic expressions and advanced parameter expansions</b>	<b>57</b>
5.1	Arithmetic expansion . . . . .	57
5.2	Non-integer arithmetic . . . . .	62
5.3	A second look at parameter expansion . . . . .	64
<b>6</b>	<b>Subshells and functions</b>	<b>67</b>
6.1	Explicit and implicit subshells . . . . .	67
6.1.1	Grouping commands . . . . .	67
6.1.2	Making use of subshells . . . . .	69
6.1.3	Implicit subshells . . . . .	71
6.2	<b>bash</b> functions . . . . .	74
6.2.1	Overwriting commands . . . . .	82
6.3	Cleanup routines . . . . .	83
6.4	Making script code more reusable . . . . .	85
<b>7</b>	<b>Regular expressions</b>	<b>89</b>
7.1	Regular expression syntax . . . . .	89
7.1.1	Matching regular expressions in plain <b>bash</b> . . . . .	89
7.1.2	Regular expression operators . . . . .	89
7.1.3	A shorthand syntax for bracket expansions . . . . .	91
7.1.4	POSIX character classes . . . . .	92
7.1.5	Getting help with regexes . . . . .	93

7.2	Using regexes with <b>grep</b> . . . . .	93
7.3	Using regexes with <b>sed</b> . . . . .	95
7.3.1	Alternative matching syntax . . . . .	98
<b>8</b>	<b>A concise introduction to awk programming</b>	<b>99</b>
8.1	Structure of an <b>awk</b> program . . . . .	99
8.2	Running <b>awk</b> programs . . . . .	100
8.3	<b>awk</b> programs have an implicit loop . . . . .	101
8.4	<b>awk</b> statements and line breaks . . . . .	103
8.5	Strings in <b>awk</b> . . . . .	104
8.6	Variables and arithmetic in <b>awk</b> . . . . .	104
8.6.1	Some special variables . . . . .	107
8.6.2	Variables in the <b>awk</b> code vs. variables in the shell script .	108
8.6.3	Setting <b>awk</b> variables from the shell . . . . .	110
8.7	<b>awk</b> conditions . . . . .	110
8.8	Important <b>awk</b> action commands . . . . .	114
8.8.1	Conditions inside action blocks: <b>if</b> . . . . .	116
8.9	Further examples . . . . .	116
8.10	<b>awk</b> features not covered . . . . .	118
<b>9</b>	<b>A word about performance</b>	<b>119</b>
9.1	Collection of bad style examples . . . . .	120
9.1.1	Useless use of <b>cat</b> . . . . .	120
9.1.2	Useless use of <b>ls *</b> . . . . .	120
9.1.3	Ignoring the exit code . . . . .	120
9.1.4	Underestimating the powers of <b>grep</b> . . . . .	121
9.1.5	When <b>grep</b> is not enough . . . . .	121
9.1.6	<b>testing</b> for the exit code . . . . .	121
<b>A</b>	<b>Obtaining the files</b>	<b>122</b>
<b>B</b>	<b>Other bash features worth mentioning</b>	<b>123</b>
B.1	<b>bash</b> customisation . . . . .	123
B.1.1	The <b>.bashrc</b> and related configuration files . . . . .	123
B.1.2	Tab completion for script arguments . . . . .	123
B.2	Making scripts locale-aware . . . . .	123
B.3	<b>bash</b> command-line parsing in detail . . . . .	123
B.3.1	Overview of the parsing process . . . . .	123
B.4	Notable <b>bash</b> features not covered . . . . .	124
<b>C</b>	<b>Supplementary information</b>	<b>125</b>
C.1	The <b>mtx</b> file format . . . . .	125
	<b>Bibliography</b>	<b>126</b>
	<b>List of Commands</b>	<b>127</b>

# List of Tables

2.1	List of noteworthy shells. . . . .	11
2.2	Summary of the output redirectors . . . . .	15
2.3	Summary of the types of pipes . . . . .	15
2.4	Summary of available commands to get help . . . . .	19
3.1	Important predefined variables. . . . .	23
4.1	A few special escape sequences for <code>echo -e</code> . . . . .	33
4.2	Overview of the most important <code>test</code> operators . . . . .	34
4.3	The most important options of <code>find</code> . . . . .	52

## Course description

The **bash** shell is the default shell in almost all major Unix and LinuX distributions, which makes learning about the **bash** scripting language pretty much unavoidable if one is working on a Unix-like operating system. On the other hand this also means that writing **bash** scripts is conceptually very simple — essentially like typing commands. When it comes to more involved tasks and more powerful scripts, however, some knowledge of the underlying operating system is certainly required. After all **bash** scripting is all about properly combining the available programs in a clever way.

This idea structures the whole course: In the first part we will revisit some basic concepts of a Unix-like operating system and review the set of Unix core-utils one needs for everyday scripting. Afterwards we will talk about the **bash** shell and its core language features, including

- control statements (**if**, **for**, **while**, ...)
- file or user input/output
- **bash** functions
- features simplifying code reuse and script structure

The final part will be concerned with the extraction of information (e.g. from files) using so-called regular expressions and programs like **awk**, **sed** or **grep**.

## Learning targets and objectives

After the course you will be able to

- apply and utilise the Unix philosophy in the context of scripting
- identify the structure of a **bash** script
- enumerate the core concepts of the **bash** scripting language
- structure a script in a way such that code is reusable in other scripts
- extract information from a file using regular expressions and the standard Unix tools
- name advantages and disadvantages of tools like **awk**, **sed** or **grep**, **cut** ..., and give examples for situations in which one is more suitable than the others.

## Prerequisites

This course assumes some familiarity with a Unix-like operating system like GNU/Linux and the `bash` shell. I.e. you should be able to

- navigate through your files from the terminal.
- create or delete files or folders from the terminal.
- run programs from the terminal (like some “one-liners”).
- edit files using a common graphical (or command-line) text editor like `gedit`, `leafpad`, `vim`, `nano`, ...

Whilst it is not assumed that you have any knowledge of programming or any experience in `bash` scripting, it is, however, highly recommended that at least either is the case.

## Compatibility of the exercises

All exercises and script samples have been tested on Debian 7 “Jessie” with the GNU `bash` 4.3 and GNU `awk` 4.1.1. Everything *should* work on other Unix-like operating systems as well, but I cannot guarantee it. Especially in Mac OS X the syntax of the commands differs in some cases, which is why some examples/exercises might not work properly.

## Errors and feedback

If you spot an error or have any suggestions for the further improvement of the material, please do not hesitate to contact me under `michael.herbst@iwr.uni-heidelberg.de`.

## Licensing and redistribution

### Course Notes

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>.



An electronic version of this document is available from <http://blog.mfhs.eu/teaching/advanced-bash-scripting-2015/>. If you use any part of my work, please include a reference to this URL along with my name and email address.

### Script examples

All example scripts in the repository are published under the CC0 1.0 Universal Licence. See the file `LICENCE` in the root of the repository for more details.

# Chapter 1

## Introduction to Unix-like operating systems

Before we dive into scripting itself, we will take a brief look at the family of operating systems on which the use of scripting is extremely prominent: The Unix-like operating systems.

### 1.1 The Unix philosophy

UNIX itself is quite an old operating system (OS) dating back to the 1970s. It was developed by Dennis Ritchie<sup>1</sup>, Ken Thompson and others at the Bell Labs research centre and was distributed by AT&T — initially in open source form. It included important new concepts, now known as the *Unix philosophy*, which made the OS very flexible and powerful. As a result it became widely used in both business and academia. Nowadays, where AT&T UNIX is pretty much dead, the Unix philosophy still plays a key role in operating system design. One can identify a whole family of OSes — the so-called Unix-like OSes or X-like OSes, which derive from the traditional AT&T UNIX. Two of the most important modern OSes, Mac OS X and GNU/Linux, are included in this family. In other words: Unix’ importance in academia and business has not changed very much over the years.

Many formulations of the Unix philosophy exist. The most well-known is the one given by Doug McIlroy, the inventor of the Unix pipe and head at Bell Labs in the 1970s[1]

Write programs that do one thing and do it well.

For the Unix-like OSes this means that in theory

- The OS is a collection of
  - small helper programs or “utilities“, that only do a simple thing (think about `ls`, `mkdir` ...)

---

<sup>1</sup>Also the creator of the “C” programming language



- programs (“shell scripts”) that combine the utilities to achieve a bigger task
- The OS is extremely modular:
  - All programs have a well-defined interface
  - It is easy to swap one program for a modified/enhanced version without breaking the rest of the OS
- The OS is standardised:
  - The functionality of the programs is (almost) identical for all OSes of the Unix-family.

### 1.1.1 Impact for scripting

On such a platform scripting becomes very helpful since

- all important functionality is available in the OS-provided utilities. So very little actual code has to be written to glue the utilities together.
  - the utilities are not too specific for a particular job and can therefore be used flexibly throughout the script.
  - documentation of their interfaces (commandline arguments) is available.
- ⇒ If one changes from one Unix-like OS to another or from one version of the OS to the next, no change in the functionality of the derived script is to be expected.
- ⇒ Scripts become reusable and portable.

## 1.2 The Unix utilities

Now let us briefly review some of the most important utility programs on a modern Unix-like OS. This list is not at all complete and in fact we will add more and more utilities to our toolbox during the course. See page 127 for a full list of commands introduced in this course.

This section is just to remind you about these commands. If more detailed information is required you should consult the manpage (by typing `man command`) or try the tips in section 2.5 on page 19.

### 1.2.1 Accessing files or directories

<code>cd</code>	Change the current working directory of the shell
<code>ls</code>	List the content of the current working directory. Important options: <ul style="list-style-type: none"><li><code>-l</code> long form: More details</li><li><code>-a</code> all: Also include hidden files</li><li><code>-h</code> human-readable: Output sizes in more readable way</li><li><code>-t</code> time: Sort output by time</li></ul>
<code>pwd</code>	Print the current working directory of the shell

### 1.2.2 Modifying files or directories

<code>touch</code>	Change the modification time if the file exists, else create an empty file, options: <ul style="list-style-type: none"><li><code>-t</code> Change modification time to the one provided</li></ul>
<code>mkdir</code>	Create a directory
<code>rm</code>	Delete files. Important options: <ul style="list-style-type: none"><li><code>-r</code> recursive: Delete all files and directories in a directory</li><li><code>-i</code> Ask before each file deleted</li><li><code>-I</code> Ask only in certain circumstances and only once (mass-delete)</li></ul>
<code>rmdir</code>	Delete empty folders
<code>chown</code>	Change ownership for a file (see section 1.3 on page 7)

### 1.2.3 Getting or filtering file content

<code>cat</code>	Concatenate one or many files together
<code>tac</code>	Concatenate files and print lines in reverse order
<code>tee</code>	Write input to a file and to output as well
<code>cut</code>	Extract columns from input, options <ul style="list-style-type: none"><li><code>-d</code> delimiter: Character to use for the split</li><li><code>-f</code> fields: Which fields(columns) to output</li></ul>
<code>grep</code>	Filter input/ by a pattern <ul style="list-style-type: none"><li><code>-i</code> ignore case</li><li><code>-v</code> invert: only non-matching lines are given</li><li><code>-o</code> only-matching: print only matching content</li><li><code>-C</code> context: print n lines of context as well</li></ul>

`-q` only the return code is determined

`sort` sort input according to some parameters, Options:

`-n` numeric sort

`-u` unique sort: each identical line is only print once

`uniq` Take a sorted input and discard double lines

`-c` count the number of occurrences

**Example 1.1.** In this example we will assume that the current working directory is the top level of the git repository <sup>2</sup>. If we run

```
1 cat resources/matrices/3.mtx
```

we get the content of the file `resources/matrices/3.mtx` (Check with a text editor) If we do the same thing with `tac`, we get the file again, but reversed line by line.

Now, many of you probably know the `<` character can be used to get the input for a command from a file. I.e. the command

```
1 < resources/matrices/3.mtx cut -f 1
```

takes its input from the file we just looked at and passes it onto `cut`. Naively we expect `cut` to print only the first column of this file. This does, however, not occur, because `cut` per default only considers the tabulator character when splitting the data into columns. We can change this behaviour by passing the arguments `-d " "`. This tells `cut` that the space character should be used as the field separator instead. So running

```
1 < resources/matrices/3.mtx cut -f 1 -d " "
```

gives the first column as desired.

**Example 1.2.** In this example we want to find all lines of the Project Gutenberg<sup>3</sup> books `pg74.txt` and `pg76.txt` that contain the word “hunger”. One could run those two commands one after another

```
1 < resources/gutenberg/pg74.txt grep hunger
2 < resources/gutenberg/pg76.txt grep hunger
```

or we can use the pipe “`|`” to connect the `cat` and `grep` commands together like

```
1 cat resources/gutenberg/pg74.txt \
2 resources/gutenberg/pg76.txt | grep hunger
```

Reminder: The pipe connects the output of the first with the input of the second command

---

<sup>2</sup>The top level is the directory in which this pdf is contained

<sup>3</sup><https://www.gutenberg.org/>

**Example 1.3.** There exists a counterpart to “<”, which writes to a file, the “>”. In principle it just takes the output from the last command and writes it to the file specified afterwards. In other words the effect of the two commands

```
1 < infile cat > outfile
2 cp infile outfile
```

is absolutely equivalent.

Note that there are many cases where the precise place where one puts the < and > is not important. For example the commands

```
1 < infile > outfile cat
2 cat <infile > outfile
```

all work equally well. The space after the “arrows” is also optional.

**Example 1.4.** Since `uniq` can only operate on sorted data, it is very common to see e.g.

```
1 < resources/testfile sort | uniq
```

This can of course be replaced by the shorter (and quicker)

```
1 < resources/testfile sort -u
```

One really might wonder at first sight why the `sort` command has the `-u` flag, since somewhat violates the Unix philosophy. Most Unix-like OS have this flag nevertheless, since sorting algorithms become more efficient if we already know that we only want to keep a single occurrence of each line.

Note, that in many cases a construct like `< file command` can actually be replaced by `command file`. Most commands are built to do the “right thing” in such a case and will still read the file. For example for `sort` this is equivalent to the above:

```
1 sort -u resources/testfile
```

In some cases the latter command tends to perform somewhat better. Nevertheless I personally prefer the version `< resources/testfile sort -u` since this has a very suggestive syntax: The data flows from the producers (`< file`) on the RHS to the consumers on the LHS and on the way passes through all commands.

#### 1.2.4 Other

<code>less</code>	View input or a file in a convenient way
<code>wc</code>	Count characters, lines or words on input
	<code>-l</code> count number of lines
	<code>-w</code> count number of words
<code>echo</code>	Print something to output
<code>man</code>	Open manual page for a command
<code>whatis</code>	Print a short summary describing a command

**Example 1.5.** If we want to find out how the commands `tail` and `head` work we could use the manpage

```
1 man tail
2 man head
```

The same works with `man` itself, try e.g.

```
1 man man
```

Problems arise with so-called shell builtins. We will talk about this in the next chapter (see section 2.5 on page 19).

### 1.2.5 Exercises

**Exercise 1.6.** Exploring the `man` program:

- Run the commands `man -Lde tail` and `man -LC tail`. What does the `-L` flag do to `man`?
- Find out about the different sections about the Unix manual (read line 21 till 41 of `man man`).
- Which section number is the most important for us?
- Find out how one can enforce an article to be from an appropriate section.

**Exercise 1.7.** A first look at Project Gutenberg books in `resources/gutenberg`

- Find out how many lines of the book `pg74.txt` actually contain “hunger”. Do this in two possible ways, both times using `grep` at least once.
  - Once use at least one pipe
  - Once use no pipe at all.
- Find out what the `grep` options `-A -B -n -H -w` do
- *optional* `pg74.txt` contains two lines that directly follow another in which the first line contains the word “hunger” and the second line contains the word “soon”. Find out the line numbers of these two lines.

**Exercise 1.8.** Looking at some matrices:

- Read the manpages of `head` and `tail`. Rebuild the effect of the `tail` command using `head`. I.e. give a commandline that achieves the same effect as `< resources/testfile tail`, but that does not contain `tail` at all.
- Find out (using the manpage) how one could print all lines but the first of a file. You can either use the commands from your answer to 1. or use `tail`, both is possible. Try your suggested command sequence on `resources/matrices/3.mtx` to see that it works.
- You might have noticed that the `mtx` files contain a few lines in the beginning that start with the special comment character “%”. Suggest another way to suppress comment lines in the file `3.mtx`.

- Provide a sequence of commands using `cut` and `sort` which prints *how many distinct* values there are in the third column. I.e. if this column contains 3 fours, 2 threes and 1 zero, the answer should be 3. Note that the columns are not separated by tabs, so you will need to play with the flag `-d` of `cut`. Again use your idea from the previous answer to ignore the comment line. Once you get an answer look at the file yourself and compare the values.
- Provide a sequence of commands that prints *the smallest* value in the third column of `3.mtx`. Again make your commands ignore the first comment line.
- Do the same thing with `resources/matrices/bcsstm01.mtx`. Be very careful and check the result properly. Here you will need the right options for `sort` for this to give the correct answer.
- Run the same sequence of commands as in the previous part on `resources/matrices/lund_b.mtx`. The result should surprise you. What goes wrong here?
- Another tool that can be used to print certain columns in files is `awk`. The syntax is `awk '{print $n}'` to print the `n`th column. Use it instead of `cut` for the file `lund_b.mtx`. How does it perform?

## 1.3 The Unix file and permission system

To conclude this chapter we want to spend some time discussing the way Unix-like operating systems organise files.

### 1.3.1 What are files?

- Convenience feature for programmers or users of the computer
- File: Virtual chunk of data.
- File path: Virtual location where user expects the file.
- File System: Provides lookup feature to translate file path to hard drive location
- Lookup mechanism incorporates extra information about the file:
  - Owner (Person who created the file)
  - Group (Group of people file is attributed to)
  - Permissions for file access
  - Time when time was created/accessed/modified
- All this information can be obtained using the `ls -l` command

- Some files are “special”, e.g.
  - soft links: Files that point to a different file path  
⇒ OS performs look-up at the other file path
  - hard links: Duplicated entries in the lookup mechanism  
⇒ Two paths point to the same hard drive location

### 1.3.2 Unix paths

Paths are a structured syntax that allow the user to tell the operating system which file he or she is referring to. In Unix these paths are characterised as follows:

- Entities on the path are separated by “/”
- The last entity may be a file or directory, all the others are directories<sup>4</sup>
- *Absolute path*: Path starting at the root directory, i.e. who has “/” as the first character
- *Relative path*: Gives a location relative to the current directory. May contain “..” to denote the parent directory relative or “.” to denote the identical directory to the entity on the left. E.g. the paths

```
1 foo/bar/baz
2 foo/./bar/./bar/./baz
```

are all relative paths to exactly the same location.

### 1.3.3 Unix permissions

Consider the following output of the command `ls -l`

```
1 drwxr-xr-x 4 mfh agdreuw 4096 Aug 15 19:07 resources
2 -rw-r--r-- 1 mfh agdreuw 4115 Aug 15 20:18 file
3 -r----- 1 mfh agdreuw 4096 Aug 15 00:00 secret
```

The output means from left to right:

- Permissions (10 chars)
  - 1 char (here `d` or `-`): Indicates the file type
  - 3 chars: Access rights for the owner
  - 3 chars: Access rights for the group
  - 3 chars: Access rights for the world (anyone else on the machine)
  - `r` means read, `w` means write, `x` means execute
- Number of hard links to this hard drive location
- Owner
- Group

---

<sup>4</sup>Which are actually just some special kind of files

- Size ( in bytes)
- Last modification time
- File name

A file is (readable/writeable/executable) for a specific user if at least one of the following is true

- He is the owner and the (r/w/x)-bit set (i.e. `ls` shows the respective letter in the listing)
- He is in the group the file belongs to and the group has the (r/w/x)-bit set
- The (r/w/x)-bit is set for the world

The permissions can be changed using the command `chmod` and the owner and group information can be changed using `chown`.

**Example 1.9.** After a run of `chmod +x secret` the `ls -l` would show

```
1 drwxr-xr-x 4 mfh agdreuw 4096 Aug 15 19:07 resources
2 -rw-r--r-- 1 mfh agdreuw 4115 Aug 15 20:18 file
3 -r-x--x--x 1 mfh agdreuw 4096 Aug 15 00:00 secret
```

Further running `chmod g-r` gave the result

```
1 drwxr-xr-x 4 mfh agdreuw 4096 Aug 15 19:07 resources
2 -rw----r-- 1 mfh agdreuw 4115 Aug 15 20:18 file
3 -r-x--x--x 1 mfh agdreuw 4096 Aug 15 00:00 secret
```



## Chapter 2

# A first look at the bash shell

In this chapter we will take a first look at the **bash** shell itself. We will discuss some very handy features to save oneself from typing too much and we will have a closer look at elementary features of the shell like pipes and redirects.

### 2.1 Historic overview

#### 2.1.1 What is a shell?

Back in the days:

- Terminal: Place where commands can be keyed in in order to do work on a computer
- Shell: Interface the OS provides to the user on a terminal

In this definition a graphical user interface is a shell as well!

Nowadays:

- Hardly any work done inside terminals any more
- Programs to start a virtual terminal: “Terminal emulator”
- Shell: Default program started by the terminal emulator

#### 2.1.2 The Bourne-again shell

- **bash** is short for Bourne-again shell
- derived and improved version of the Bourne shell **sh**
- Pretty much the default shell on all Unix-like OS
- Other important shells see table 2.1 on the next page

<b>sh</b>	Bourne shell	1977	first Unix shell
<b>csh</b>	C shell	1978	syntax more like C
<b>ash</b>	Almquist shell	1980s	lightweight shell
<b>ksh</b>	Korn shell	1983	<b>sh</b> improved by user requests at Bell Labs
<b>bash</b>	Bourne-again shell	1987	the default shell
<b>zsh</b>	Z shell	1990	massive and feature-rich, compatible to <b>bash</b>




Table 2.1: List of noteworthy shells. For more information see [https://en.wikipedia.org/wiki/Comparison\\_of\\_command\\_shells](https://en.wikipedia.org/wiki/Comparison_of_command_shells)

## 2.2 Handy features of the bash

### 2.2.1 Tab completion

- Can save you from a lot of typing
- Needs to be loaded by running





```
1 . /etc/bash_completion
```

- Press  once to complete a command
- Press   to get list of possible completions
- Works on files and options

### 2.2.2 Accessing the command history

Consider a sequence of commands


















```
1 ls resources/
2 cd resources/
3 ls -al
4 ls matrices
5 cd matrices
6 ls -al
7 ls -al
```

- It would be nice to do as little typing as possible
- Fortunately the **bash** remembers what was most recently typed
- Navigation through history using  and 
- The last line can also be executed by  



Another way of accessing the history is given by the so-called *history expansion*, e.g.











```
!!      run the most recent command again
!$      the last argument of the previous command line
!^      the first argument of the previous command line
!:n     the n-th word of the previous command line
!:n-m   words n till m of the previous command line
```

So if we assume the working directory is the top level directory of the git repository, we could just type

```
ls r  
cd !$  
ls -al  
ls m  
cd !$  
   
  
```

to achieve the same thing as above.

Another thing worth mentioning here is *reverse-i-search*. In order to transform the shell in this mode type  + .

- Now start typing
- The shell will automatically display the most recent command matching command line
- type  to execute
- type more chars to continue searching
- use , , , , ... to edit the current match, then  to run the edited version
- type  +  to go to the next match further back in the history
- type  +  to abort





Note that both tab completion as well as the `bash`s history features do only work in an interactive environment and not when writing scripts.

**Exercise 2.1.** What is the smallest number of keystrokes you need to achieve the execution of the following command sequences.



```
1 cd resources
2 ls images | grep blue          #no file blue exists
3 ls|grep blue
4 mkdir grep_red grep_blue
```


Assume as usually that the current working is the top level of the repository. Assume further that the command history is filled exactly with these entries (from oldest to newest):

```
1 ls images | grep red
2 ls tables
3 ls resources
```



Note: Count special symbols like “\_” or “|” or combined strokes like  +  as one keystroke. Also count all  s or  s required.

### 2.2.3 Running multiple commands on a single line

The **bash** offers quite a few ways to separate subsequent commands from one another. The simplest one, which everyone has used already multiple times just for this course, is the newline character (as produced by the  key). The character `;` is entirely synonymous to . So typing

```
cd -; ls 
```

or

```
cd -   
ls 
```

is equivalent.

In contrast the character `&` tells the **bash** to send the program on its left to background and immediately proceed with the execution of the next command. This is extremely helpful for running long jobs without blocking the shell, e.g.

```
1 cp BigFile /media/usbstick/ & ls resources
```

would start copying the big file **BigFile** to the usbstick and immediately display the content of **resources**, not waiting for the copying to be finished. During the execution of the background job `cp BigFile /media/usbstick/`, output from *both* jobs will be displayed on the terminal.

If more than one command is specified on a single commandline, the compound is also called a “command list”, so `cd -; ls` and `cp BigFile /media/usbstick/ & ls resources` are examples of command lists.

## 2.3 Redirecting command input/output

Each command which is run on the terminal per default opens 3 connections to the shell environment:

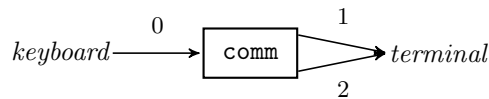
- *stdin* or file descriptor (fd) 0: The command reads all input from here
- *stdout* or fd 1: All normal output is printed here
- *stderr* or fd 2: All output concerning errors is printed here

Especially the distinction what is printed to *stdout* and what is printed to *stderr* is not clear and programs can sometimes give rise to rather unexpected behaviour. Usually one can expect error messages on *stderr*, everything else on *stdout*. There are a few good reasons to distinguish *stdout* and *stderr*:

1. In many cases one is only interested in part of the output of a program
  - ⇒ One pipes the program into **grep**
  - ⇒ Only a small portion of the output produced reaches the eye of the user
    - But: We still want to see all the errors

2. Scripts often capture the output of a program for later use.
  - ⇒ Programmer only expects normal output in the capture, no error messages
  - ⇒ Can capture *stdout* but not *stderr*
3. Usually one can safely discard the output on *stdout* whereas *stderr* is usually important.
  - ⇒ Output implicitly split into two categories for logging.

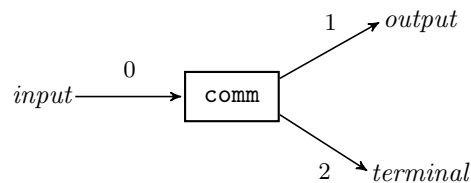
By default *stdin* is connected to the keyboard and both *stdout* and *stderr* are connected to the terminal. Running a `comm` in the shell hence gives a “redirection diagram” like



As we already know the characters `<` and `>` can be used to read/write from/to a file, so the commandline

```
1 < input comm >output
```

can be visualised as



If we want to prevent the content of the file `output` to be overwritten, we can use the syntax

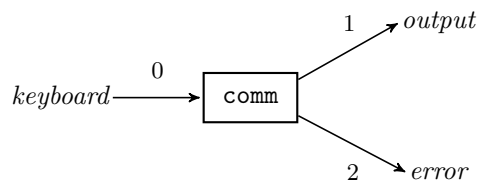
```
1 < input comm >>output
```

This does exactly the same thing as above, just it *appends* *stdout* to the file `output` instead of deleting the previous content and replacing it by the output of `comm`.

If one wants to redirect the output on *stderr* to a file called `error` as well, we can use the commandline

```
1 comm >output 2>error
```

or pictorially



syntax	Comment
>	print <i>stdout</i> to file
>>	append <i>stdout</i> to file
2>	print <i>stderr</i> to file
2>>	append <i>stderr</i> to file
&>	print <i>stdout</i> and <i>stderr</i> to file
&>>	append <i>stdout</i> and <i>stderr</i> to file

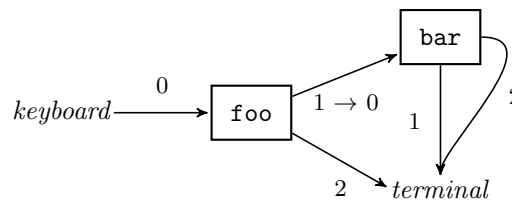
Table 2.2: Summary of the output redirectors of the **bash** shell. The versions with a single > always substitute the content of the file entirely, whereas the >> redirectors append to a file.

syntax	Comment
	connect <i>stdout</i> $\rightarrow$ <i>stdin</i>
&	connect <i>stdout</i> and <i>stderr</i> $\rightarrow$ <i>stdin</i>

Table 2.3: Summary of the types of pipes

Many more output redirectors exist. They all differ only slightly depending on what file descriptor is redirected and whether the data is appended or not. See table 2.2 for an overview.

Similar to output redirection >, a pipe between commands **foo** | **bar** only connects *stdout* to the next command and not *stderr*, i.e.



Again there is also a version that pipes both *stdout* and *stderr* to the next command, see table 2.3.

One very common paradigm in scripting is output redirection to the special device files **/dev/null** or **/dev/zero**. These devices have the property, that they discard everything which gets written to them. Therefore all unwanted output may be discarded by writing it to e.g. **/dev/null**. For example, consider the script **2\_intro\_bash/stdout\_stderr.sh** and say we really wanted to get all errors but we are not very much interested in *stdout*, then running

```
1 2_intro_bash/stdout_stderr.sh > /dev/null
```

achieves exactly this task. If we want it to be entirely quiet, we could execute

```
1 2_intro_bash/stdout_stderr.sh &> /dev/null
```

**Exercise 2.2.** Visualise the following command line as a redirection diagram

```
1 ls |& grep test | grep blub | awk '{print $2}' &> outfile
```

**Exercise 2.3.** `tee` is a very handy tool if one wants to log the output of a long-running command. We will explore it a little in this exercise.


- Imagine you run a program called `some_program` which does a lengthy calculation. You want to log all the output the program produces (on either `stdout` or `stderr`) to a file `log.full` and all output that contains the keyword “error” to `log.summary`. Someone proposes the commandline

```
1 some_program | tee log.full |& grep error &> log.✓
    ↪ summary
```

Draw the redirection diagram. Does it work as intended? If not propose a commandline that does achieve the desired goal making sure that only output from `some_program` actually reaches the log files.

- What happens if you run the command multiple times regarding the log files? Take a look at the manpage of `tee` and propose an alternative command line that makes sure that no logging data is lost between subsequent runs of `some_program`.

**Exercise 2.4.** • Create a file called `in` and write some random text to it.

- Run `< in cat > out`. What happens?
- Run `< in cat > in`. What happens here?
- Draw a redirection diagram for running plain `cat`. How can you explain that the terminal seems to “hang” if just `cat` is executed on the commandline.  
(Hint: Run `cat`, type something to the terminal and press )

## 2.4 The exit status of a command

Apart from writing messages to `stdout` or `stderr`, there is yet another channel to inform the user how the execution of a program went:

- Each command running on the shell returns an integer value between 0 and 255 on termination, the so-called “exit status” or “return code”.
- By convention 0 means “no errors”, anything else implies that something went wrong.
- The meaning of a specific can be checked from the program’s documentation (at least in theory)
- The return code is usually not printed to the user, just implicitly stored by the shell.
- In order to get the exit code of the most recently terminated command one may execute `echo $?`
- Note that this is in turn a command and hence alters the value printed by the next execution of `echo $?`.

### 2.4.1 Logic based on exit codes: The operators `&&`, `||`, `!`

We already looked at the `&` and `;` operators to separate commands in a command list, e.g.

```
1 foo ; bar
2 foo & bar
```

In both syntax there is no control about the execution of `bar`: Irrespective whether `foo` is successful or not, `bar` is executed. If we want execution of the `bar` command only if `foo` succeeds or fails, we need the operators `&&` or `||`, respectively:

```
1 foo || bar # bar only executed if foo fails
2 foo && bar # bar only executed if foo successful
```

- Conditional `cd`:

```
1 cd blub || cd matrices
```

Goes into directory `matrices` if `blub` does not exist.

- If the annoying error message should be filtered in case `blub` does not exist, one could run

```
1 cd blub &> /dev/null || cd matrices
```

- Very common when developing code:

```
1 make && ./a.out
```

The compiled program `./a.out` is only executed if compiling it using `make` succeeds.

- A list of commands connected by `&&` is called an “AND list” and a list connected by `||` an “OR list”.
- AND lists or OR lists may consist of more than one command

```
1 ./configure && make && make install && echo Successful
```

- This works as expected since the return code of such an AND/OR lists is given by the last command in the sequence
- One can also intermix `&&` and `||`

```
1 cd blub &> /dev/null || cd matrices && vim 3.mtx
```

although this can lead to very hard-to-read code (see exercise below) and is therefore discouraged.

Finally there also exist the operator `!` that inverts the return code of the following program. So running

```
1 ! ls
```

returns the exit code 1 if `ls` has been successful and 0 on error.



**Exercise 2.5.** Go to the directory `resources/directories`. Explain the output of the following commands

- Run

```
1 cd 3/3 || cd 4/2 && cd ../4 || cd ../3 && cat file
```

Note, that this changes the working directory on the shell, so in order to run it again, you need to `cd` back to `resources/directories` beforehand.

- Suggest the places at which we need to insert a `2>/dev/null` in order to suppress the error messages from `cd`. Try to insert as little code as possible
- Go back to the directory `resources/directories`. Now run

```
1 mkdir -p 3/3; cd 3/3 || cd 4/2 && cd ../4 || cd ../3 ↵
   ↪&& pwd
```

**Exercise 2.6.** Find out what the programs `true` and `false` do. Look at the following expressions and try to determine the exit code without executing them. Then check yourself by running them on the shell. Remember that you can access the exit code of the most recent command via `echo $?`

```
1 false || true
2 true && false || true
3 false && false && true
4 false || true || false
```

Run the following commands on the shell

```
1 false | true
2 true | true
3 true | false
4 false | false
5 false |& true
```

What does the pipe do wrt. to the return code?

**Exercise 2.7.** We already talked about the `grep` command in order to search for strings. One extremely handy feature of `grep` is that it returns 0 if it found a match and 1 otherwise. Change to the directory `resources/gutenberg`. Propose `bash` one-liners for each of the following problems.

- Print “success” if the file `pg1661.txt` contains the word “the” (there is a special `grep` flag for word matching), else it should print “error”.
- Do the same thing, but use a special flag of `grep` in order to suppress all output except the “success” or “error” in the end. Apart from there being less amount of output, what is different?
- Now print “no matches” if `pg1661.txt` does not contain the word “Heidelberg”, else print the number of times the word is contained in the file.
- Try a few other words like “Holmes”, “a”, “Baker”, “it”, “room” as well.
- Count the number of words in the file `pg1661.txt`

program	description
<code>man</code>	Accessing the manual pages
<code>info</code>	Accessing the Texinfo manual
<code>whatis</code>	Print a short summary describing a command
<code>apropos</code>	Search in manpage summaries for keyword
<code>help</code>	Access help for <code>bash</code> builtin commands

Table 2.4: Summary of available commands to get help

**Exercise 2.8.** Code `echo` is a command which just prints all of its arguments to *stdout*. As usually we can use output redirection to write this to a file or use a pipe to pipe it to a different program.

Keeping this in mind take a look at the following commands, which are all valid `bash` shell syntax. What do the commandlines mean? How are *stdin*, *stdout* and *stderr* of `grep` connected? What is the exit code?

- `echo test | grep test`
- `echo test & grep test`
- `echo test |& grep test`
- `echo test && grep test`
- `echo test || grep test`

## 2.5 Tips on getting help

It is not always clear how to get help when writing a script or using the commandline. Many commands exist that should provide one with this answers. Table 2.4 gives an overview.

If one knows the name of a command usually a good procedure is:

1. Try to execute `command --help` or `command -h`. Many commands provide a good summary of their features when executed with these arguments.
2. Try to find help in the manpage `man command`
3. If the manpage did not answer your problem or says something about a Texinfo manual, try accessing the latter using `info command`
4. If both is unsuccessful the command is probably not provided by the system, but by the `bash` shell instead – a so-called *shell builtin*. In this case try finding help via `help command`

If the precise command name, however is not known, try to find it first using `apropos keyword`.

A word of warning about shell builtin commands:

- It is intentional that shell builtin commands act extremely alike external commands

- Examples for perhaps surprising shell builtins are `cd`, `test` or `echo`
  - Some of these commands — like `test` or `echo` — are provided by the OS as well.
  - The builtins get preference by the `bash` for performance reasons
- ⇒ The manpage for some commands (describing the OS version of it) do not always agree with the functionality provided by the `bash` builtin.
- Usually the `bash` has more features
- ⇒ Bottom line: Sometimes you should check `help command` even though you found something in the manpages.

**Exercise 2.9.** By crawling through the help provided by the `help` and the `man` commands, find out which of these commands are shell builtins:

```
man kill time fg touch info history rm pwd ls exit
```

## Chapter 3

# Simple shell scripts

In this chapter we will dive into proper scripting and discuss the basic `bash` scripting syntax.

### 3.1 What makes a shell script a shell script?

The simplest script one can think of just consists of the so-called *shebang*

```
1 #!/bin/bash
```

This line, starting with a hash(`#`) and a bang(`!`) — hence the name — tells the OS which program should be used to interpret the following commands. If a file with executable rights is encountered that begins with a shebang, the OS starts up the specified program (in this case `/bin/bash`). Then the remaining content of the file is fed into this program’s *stdin*<sup>1</sup>. In order to compose a shell script we hence need two steps

- Create a file containing a shebang like `#!/bin/bash`
- Give the file executable rights by calling `chmod +x` on it.

#### 3.1.1 Executing scripts

Once script files are made executable using `chmod +x` we can execute it on the shell like any other command. Consider the simple script

```
1 #!/bin/bash
2 echo Hello world!
```

3.simple\_scripts/hello.sh

which just issues a “Hello world.” If the current working directory of the shell is exactly the directory in which `hello.sh` has been created, we can just run it by executing

```
1 ./hello.sh
```

---

<sup>1</sup>Strictly speaking the shebang is not required, since a missing shebang causes the default shell to be used — which works well for many cases. It is nevertheless good practice to include the shebang as it makes the scripts more portable

Otherwise we need to call it by either the full or the relative path of the script file<sup>2</sup>. E.g. if we are in the top directory of the course git repository, we need to execute

```
1 3_simple_scripts/hello.sh
```

instead.

### 3.1.2 Scripts and *stdin*

Similar to other commands, scripts can also process data provided on their *stdin*. E.g. consider the script

```
1 #!/bin/bash
2 cat
```

3\_simple\_scripts/cat.sh

which just contains a `cat`. On call we can redirect input to it

```
1 < resources/testfile 3_simple_scripts/cat.sh
```

or pipe to it

```
1 echo "data" | 3_simple_scripts/cat.sh
```

both is valid syntax. As you probably noticed in both cases the effect is exactly identical to

```
1 < resources/testfile cat
```

or

```
1 echo "data" | cat
```

This is because everything that is input on the script's *stdin* is available for the programs inside the script to process. In other words the *stdin* of the programs inside the script is fed by the *stdin* of the whole script. We will discuss this in more detail in section 4.7.2 on page 48.

## 3.2 Shell variables

Shell variables are defined using the syntax

```
1 VAR=value
```

and are accessed by invoking the so-called *parameter expansion*, e.g.

```
1 echo $VAR
```

- The name of the variable, i.e. `VAR` has to start with a letter and can only consist of alphanumeric characters and underscores.
- The convention is to use all-upper-case names in shell scripts.

```
1 123=4    #wrong
2 VA3=a    #ok
3 V_F=2    #ok
```

<sup>2</sup>This can be changed by altering the `PATH` variable. See section 6.4 on page 85

name	value
USER	name of the user running the shell
HOSTNAME	name of the host on which the shell runs
PWD	The current working directory
RANDOM	Random value between 0 and 32767
HOME	The user's home directory
PATH	Search path for commands
SHELL	Full path of the shell currently running

Table 3.1: Important predefined variables in the `bash` shell. See [2] for details.

- The value does not need to be a plain string but may contain requests to expand other variables, command substitutions (see section 3.2.2 on page 25), arithmetic expansion (see section 5.1 on page 57 and many more (see manual [2]))

```
1 VAR=a${OTHER}34
```

- value may be empty

```
1 VAR=
```

- When expanding a parameter the braces `{}` are only required if the character which follows can be misinterpreted as part of the variable name

```
1 VAR=123
2 VAR2=${VAR}23      #fails
3 VAR2=${VAR}23      #correct
```

- Undefined variables expand to an empty string
- All `bash` variables are stored as plain strings<sup>3</sup>, but they can be interpreted as integers if a builtin command requires this (e.g. `test` — see section 4.2 on page 32)
- Variables can also be deleted<sup>4</sup> using

```
1 unset VAR
```

- A wide range of predefined variables exist (see table 3.1)

<sup>3</sup>This can be changed, however, see the `declare` command in the manual [2]

<sup>4</sup>Note: Not the same thing as setting the variable to the empty string.

### 3.2.1 Special parameters

Apart from the variables we mentioned above, the shell also has a few special parameters. Their expansion works exactly like for other variables, but unlike their counterparts above, their values cannot be changed.

- positional parameters 1, 2, ...; expand to the respective argument passed to the shell script. E.g. if the simple script

```
1 #!/bin/bash
2
3 echo The first: $1
4 echo The second: $2
```

3.simple\_scripts/first\_script.sh

is executed like

```
1 3_simple_scripts/first_script.sh first second
```

we get

```
1 The first: first
2 The second: second
```

- parameter @, which expands to the list of all positional parameters
- parameter #, expands to the number of positional parameters, that are non-zero
- parameter ?, expands to the return code of the most recently executed list of commands
- parameter 0, expands to name of the shell or the shell script

**Example 3.1.** If the script

```
1 #!/bin/bash
2 echo 0: $0
3 echo 1: $1
4 echo 2: $2
5 echo 3: $3
6 echo 4: $4
7 echo @: $@
8 echo ?: $?
9 echo "#: $#"
```

3.simple\_scripts/special\_parameters.sh

is executed like

```
1 3_simple_scripts/special_parameters.sh 1 2 3 4 5 6 7 8 9
```

we get

```
1 0: 3_simple_scripts/special_parameters.sh
2 1: 1
3 2: 2
4 3: 3
```

```

5 4: 4
6 @: 1 2 3 4 5 6 7 8 9
7 ?: 0
8 #: 9

```

For more details about the parameter expansion see chapter 5 on page 57.

### 3.2.2 Command substitution

In order to store the output of a command in a variable, we need a feature called *command substitution*. The basic syntax is

```
1 VAR=$(command_list)
```

- Command substitution only catches output produced on *stdout*, e.g. running the code

```
1 VAR=$(ls /nonexistent)
```

would still result in the “File not found” error message being printed on the terminal, since `ls` prints this message to *stderr*.

- Inside the `$()` we have a so-called subshell (see also section 6.1 on page 67), where output redirection is possible. We could hence suppress the error message by running

```
1 VAR=$(ls /nonexistent 2> /dev/null)
```

- Another consequence of the subshell is, that output of all commands within the `$()` is combined:

```
1 VAR=$(echo one;echo two)
2 echo "$VAR"
```

gives

```

1 one
2 two

```

- The return code of a command substitution is the return code of the command list provided, i.e. the code of the last command executed. So we could use

```
1 VAR=$(ls /nonexistent 2> /dev/null) || echo something ✓
    ↪ wrong here
```

in order to inform the user that something went wrong with the `ls` command.

- Command substitution may be used as an argument for another command:

```
1 ls $(echo chem_output)
```



- Command substitutions may be nested:

```
1 VAR=$(echo $(echo $(echo value)))
2 # VAR now contains "value"
```

**Exercise 3.2.** *optional* Write a **bash** quine<sup>5</sup>, i.e. a script that produces its source code as output when executed. Hint: The solution has less than 20 characters.

**Exercise 3.3.** This exercise is again considered with the matrices in **resources/matrices**.

- Write a script that copies all data from **resources/matrices/3.mtx** to **output.mtx** with the exception that the first (comment) line should appear at the very end of the file **output.mtx**
- In other words the net effect should be that the script moves the comment line to the end of **output.mtx**

Now generalise the script: Make use of the positional parameters in order to:

- Write a script that takes two arguments: The first should be a matrix file, the second should be an output file, to which the script will write all data.
- The script should again copy all data over from the matrix file to the output file, with the exception that the comment line appears at the end of the output file.

**Exercise 3.4.** Write a script that parses input on *stdin* and takes a pattern as first arg.

- The input should be cached in a variable.  
Hint: For shell scripts the *stdin* of individual commands is connected to the *stdin* of the whole script. You also know a way to transfer data from *stdin* to *stdout* without doing anything with it.
- **grep** for the pattern in the cached input and count the number of matches.
- Then print the number of words in the data.

Input on *stdin* is very volatile, once you used it in a script it is gone forever (see section 4.7.2 on page 48 for more details on this). If we need to use it multiple times, we therefore need a temporary cache, like in this example.

---

<sup>5</sup>[https://en.wikipedia.org/wiki/Quine\\_%28computing%29](https://en.wikipedia.org/wiki/Quine_%28computing%29)

### 3.3 Escaping strings

Some characters are special to the `bash` shell:

- “\$”: Initiates parameter substitution
- “#”: Starts a comment
- “;”, “&”, “&&”, “||”: Separate commands in a command list
- “\”: Starts an escape (see below)
- A few more [2]

It happens many times that one needs to use these characters not by their special, but by their *literal* meaning. Examples are:

- Printing data with `echo`
- Defining variables

In such a case we need to *escape* them, i.e. precede them by a `\` character, e.g.

```
1 blubber=foo
2 echo \ $blubber \# \; \\\
```

produces

```
1 $blubber_#;\
```

whereas

```
1 blubber=foo
2 echo $blubber #;\
```

gives rise to

```
1 foo
```

We can even escape a line break by using a `\` as the very last character on a commandline

```
1 echo some very \
2   long line of code \
3   | grep line
```

```
1 some_very_long_line_of_code
```

As a rule of thumb the escape `\` causes the next character to lose its special meaning and be interpreted like any other character.

### 3.4 Word splitting and quoting

Right before the execution of a commandline<sup>6</sup>, i.e. after all variables, parameters and commands have been substituted, the shell performs an operation called *word splitting*:

- The whole commandline is expected and split into smaller strings at each `<newline>`, `<tab>` or `<space>` character. These smaller strings are called *words*.
- Each word is now considered a separate entity: The first word is the program to be executed and all following words are considered to be arguments to this command<sup>7</sup>.

**Example 3.5.** When the shell encounters the command line

```
1 grep ${KEYWORD} $4 $(echo test blubber blub)
```

it first substitutes the commands and parameters:

```
1 # assume KEYWORD=search and 4=3:
2 grep search 3 test blubber blub
```

So the command executed is `grep` and it will be passed the five arguments `search`, `3`, `test`, `blubber`, `blub`.

If we want to prevent word splitting at certain parts of the commandline we need to *quote*. This means that we surround these respective parts by either the single quote `"` or the double quote `"`, e.g.

```
1 echo "This whole thing is a single word"
2 echo 'This guy as well'
```

Similar to escaping, quoting also causes some special characters to lose their meaning inside the quotation:

- single quote `"`: No special characters, but `"` survive
  - ⇒ `"`, `$`, `#` are all non-special
  - ⇒ No parameter expansion or command substitution
  - ⇒ No word splitting
- double quote `"`: Only `"`, `$` and `\` remain special
  - ⇒ We can use parameter expansion, command substitution and escaping
  - ⇒ No word splitting

<sup>6</sup>See appendix B.3.1 on page 123 for more details how a commandline is parsed

<sup>7</sup>With command lists the shell obviously interprets the first word of each “instruction” as the command to be executed and the remaining ones as corresponding arguments.

**Example 3.6.** We consider the output of the script

```

1 #!/bin/bash
2
3 ABC=abcdef
4 NUM=123
5 EXAMPLE="$ABC$NUM$(date)_next"
6 EXAMPLE2=' $ABC$NUM$(data) '
7 echo "$EXAMPLE"
8 echo "\"some other example: \"_$EXAMPLE2
9
10 CODE="echo"
11 CODE="$CODE 'test'"
12 $CODE
13
14 #_we_can_quote_inside_command_substitutions:
15 TEST="$(echo "some_words")"
16 echo_"$TEST"

```

3\_simple\_scripts/quoting\_example.sh

which is

```

1 abcdef123Mo_24._Aug_21:07:23_CEST_2015_next
2 "some_other_example: _$ABC$NUM$(data)
3 'test'
4 some_words

```

**Example 3.7.** The only way to represent an empty string or pass an empty argument to a function is by quoting it, e.g. calling

```

1 VAR=
2 3_simple_scripts/first_script.sh $VAR -h

```

gives

```

1 The_first:_-h
2 The_second:

```

Whilst

```

1 3_simple_scripts/first_script.sh "$VAR" -h

```

gives

```

1 The_first:
2 The_second:_-h

```

Forgotten quoting or escaping is a very common source of error — some hints:

- When passing arguments to commands *always* quote them using double quotes (unless you have a reason not to)
  - ⇒ This avoids problems when variables are empty
  - ⇒ It does not hurt anything

- When initialising variables *always* quote the values using double quotes  
⇒ Same reason as above
- When a variable contains a path be extra careful that you use double quotes everywhere you use it  
⇒ Paths or filenames may contain spaces
- Use syntax highlighting in your editor<sup>8</sup>  
⇒ You will discover missing escapes or closing quotes much more quickly

**Exercise 3.8.** The following script is supposed to extract some information from a few files in different directories. Identify possible problems.

```

1 #!/bin/bash
2 # script to extract some information from directories
3 # $1: additional keyword to search for
4 #
5 cd Top Dir
6 ADDITIONAL=$(<output grep $1)
7 IMPORTANT=$(<output grep -i important)
8 cd Lower
9 FILE=$(<out1 grep -H $1; <out2 grep -H $2)
10 COUNT=$(echo '$FILE' | wc -l)
11
12 echo results:
13 echo "    important messages:" $IMPORTANT
14 echo '    other messages: $ADDITIONAL '
15 echo we found $COUNT more findings in
16 echo $FILE

```

3.simple-scripts/ex\_quoting.sh

**Exercise 3.9.** It is very common to see the paradigm

```
1 echo "$VAR" | wc -l
```

in order to count the number of lines in the variable `VAR`. Try this for the following values of `VAR`:

- `VAR=$(echo line1; echo line2)`, i.e. two lines of data
- `VAR=$(echo line1)`, i.e. one line of data
- `VAR=""`, i.e. no data at all

Can you describe the problem? There exists an alternative method to count the number of lines, which is more reliable

```
1 echo -n "$VAR" | grep -c ^
```

<sup>8</sup>vi: syntax on, Emacs: font-lock-mode

You will learn in the next chapter that the `-n` flag prevents `echo` from printing an extra trailing `<newline>` character after the content of `VAR` has been printed. The parameter `^` which is passed to `grep` is a so-called *regular expression*, which we will discuss in more detail in chapter 7 on page 89. For now it is sufficient to know that `^` is a “special” kind of keyword that matches all beginnings of all lines.

- Try this command on the three examples above to verify that it works.

**Exercise 3.10.** *optional* Write a script that

- takes a pattern (which may contain spaces) as an argument.
- uses recursive `ls` (`manpage`) to find all directories below the current working directory, which have a relative path, that matches the pattern.
- prints the relative paths of these matching directories.

For example: If the current working directory contains the directory `resources/matrices` as well as the directory `resources/gutenberg`, and the pattern is “gut”, the script should print `resources/gutenberg` but not the other path. A few hints:

- First run `ls --recursive` once and try to understand the output
- What distinguishing feature do directory paths have compared to the other output printed?
- Everything can be achieved in a single line of `bash` using only 3 different programs (`ls`, `grep` and one more).
- You might need to make the assumption that none of the files or directories below the working directory contains a “:” character in their name in order to achieve the functionality.

**Exercise 3.11.** Write a script that takes a filename and 3 keywords. It should `grep` in the file for all 3 keywords and display for each keyword the number of matches followed by the line numbers where the matches did occur.

- No other output on *stdout* should be produced by the script
- If the file cannot be read the script should exit with a return code 1, else with code 0 (see `help exit` if you do not know the exit command)
- Count the number of characters excluding comments (use the script `resources/charcount.sh` for this task). The shortest shell script (using only what we have covered so far) wins :)

## Chapter 4

# Control structures and Input/Output

This chapter we will jump from simple scripts where instructions are just executed line-by-line to more complicated scripts that contain conditions or loops. We will also discuss some of the available options to read or write data from scripts.

### 4.1 Printing output with echo

The most basic output mechanism in shell scripts is the `echo` command. It just takes all its arguments and prints them to *stdout* separated by a `<space>` character. A few notes:

- For printing to *stderr* one can use a special kind of redirector, namely `>&2`<sup>1</sup>

```
1 echo "This goes to stdout"
2 echo "This goes to stderr" >&2
```

This is needed for error messages, which should by convention be printed on *stderr*.

- The argument `-n` suppresses the final newline (see exercise 3.9 on page 30)
- The argument `-e` enables the interpretation of a few special escapes (see `help echo` and table 4.1 on the next page)

### 4.2 The test program

`test` is a very important program that is used all the time in scripting. Its main purpose is to compare numbers or strings or to check certain properties about files. `test` is extremely feature-rich and this section can only cover the

---

<sup>1</sup>This redirector is general: It works also in command substitution expressions or anywhere else on the shell

escape	meaning
<code>\t</code>	<tab> char
<code>\\</code>	literal \
<code>\n</code>	<newline> char

Table 4.1: A few special escape sequences for `echo -e`

most important options. For more detailed information about `test`, consider `help test` and the `bash` manual [2].

Most checks the `test` program can perform follow the syntax

```
1 test <operator> <argument>
```

or

```
1 test <argument1> <operator> <argument2>
```

e.g.

```
1 test -z "$VAR" # Test if a string is empty
2 test "a" == "b" # Test if two strings are equal
3 test 9 -lt 3    # Test if the first number is less than the
  ↪ second
4 test -f "file" # Test if a file exists and is a regular
  ↪ file
```

An overview of important test operators gives table 4.2 on the following page. In fact `test` is so important that a second shorthand notation using rectangular brackets exists. In this equivalent form the above commands may be written as

```
1 [ -z "$VAR" ]
2 [ "a" == "b" ]
3 [ 9 -lt 3 ]
4 [ -f "file" ]
```

There are a few things to note

- The space before the closing “]” is important, else the command fails.
- `bash` can only deal with integer comparison and arithmetic. Floating point values cannot be compared on the shell (but there are other tools like `bc` to do this, see 5.2 on page 62)
- The `test` command does not produce any output, it only returns 0 for successful tests or 1 for failing tests.
- Therefore we can use the `test` command and the `&&` or `||` operators to guard other commands. E.g.

```
1 [ -f "file" ] && < "file" grep "key"
```

makes sure that `grep` is only executed if the file “file” does exist.

- There also exists the command `[[` in the `bash` shell, which is more powerful. We will talk about this command briefly when we introduce regular expressions in section 7.1.1 on page 89.



operator	description
<code>-e FILE</code>	True if file exists.
<code>-f FILE</code>	True if file exists and is a regular file.
<code>-d FILE</code>	True if file exists and is a directory.
<code>-x FILE</code>	True if file exists and is executable.
<code>-z STRING</code>	True if string is empty
<code>-n STRING</code>	True if string is not empty
<code>STRING = STRING</code>	True if strings are identical
<code>STRING != STRING</code>	True if strings are different
<code>! EXPR</code>	True if EXPR is false
<code>EXPR1 -o EXPR2</code>	True if EXPR1 or EXPR2 are true
<code>EXPR1 -a EXPR2</code>	True if EXPR1 and EXPR2 are true
<code>( )</code>	grouping expressions
<code>NUM1 -eq NUM2</code>	True if number NUM1 equals NUM2
<code>NUM1 -ne NUM2</code>	True if NUM1 is not equal to NUM2
<code>NUM1 -lt NUM2</code>	True if NUM1 is less than NUM2
<code>NUM1 -le NUM2</code>	True if NUM1 is less or equal NUM2
<code>NUM1 -gt NUM2</code>	True if NUM1 is greater NUM2
<code>NUM1 -ge NUM2</code>	True if NUM1 is greater or equal NUM2

Table 4.2: Overview of the most important `test` operators

**Exercise 4.1.** Write a shell script that takes 3 arguments and prints them in reverse order. If `-h` is entered anywhere a short description should be printed as well.

**Exercise 4.2.** *optional* Write a shell script that does the following when given a path as first arg:

- If the path is a file, print whether it is executable and print the file size
- If the path is a directory `cd` to it

### 4.3 Conditionals: `if`

The simplest syntax of the `if` command is

```
1 if list; then list; fi
```

It has the effect:

- All the commands in the `list` are executed.
- If the return code of the `list` is 0, the `then-list` is also executed.

for example

```
1 #!/bin/bash
2 if [ 1 -gt 2 ]; then echo "Cannot happen"; fi
3 if [ 1 -gt 2 ]; VAR=4; then echo "VAR=$VAR"; fi
4 if ! cd ..; then echo "Could not change directory" >&2 ; fi
5 echo $PWD
```

4\_control.io/ifexamples.sh

gives output

```
1 VAR=4
2 /export/home/abs/abs001/bash-course
```

An extended syntax with optional **else** and **elif** (else-if) blocks is also available:

```
1 if list; then
2     list
3 elif list; then
4     list
5 ...
6 else list
7 fi
```

- Again first the **if-list** is executed
- If the return code is 0 (the condition is true) the first **then-list** is executed
- Otherwise the **elif-lists** are executed in turn. Once such an **elif-list** has exit code zero, the corresponding **then-list** is executed and the whole **if-command** completes.
- Otherwise, the **else-list** is executed.
- The exit status of the whole **if-command** is the exit status of the last command executed, or zero if no condition tested true.

**Example 4.3.** The script

```
1 #!/bin/bash
2 USERARG=0 # bash does not know boolean
3     # convention is to use 0/1
4     # or y/n for this purpose
5
6 # [ "$1" ] is the same as ! [ -z "$1" ]
7 if [ "$1" ]; then
8     USERARG=1
9     echo "Dear user: Thanks for feeding me input"
10 fi
11
12 if [ $USERARG -ne 1 ]; then
13     echo "Nothing to do"
14     exit 0
15 fi
16
17 if [ "$1" == "status" ]; then
18     echo "I am very happy"
19 elif [ "$1" == "weather" ]; then
20     echo "No clue"
21 elif [ "$1" == "date" ]; then
22     date
23 elif [ -f "$1" ]; then
24     if ! < "$1" grep "robot"; then
25         echo "Could not find keyword" >&2
```

```

26     exit 1
27 fi
28 else
29     echo "Unknown␣command:␣$1" >&2
30     exit 1
31 fi

```

4\_control\_io/more\_ifexamples.sh

when run with arg "date" produces the output

```

1 Dear␣user:␣Thanks␣for␣feeding␣me␣input
2 Di␣18.␣Aug␣16:38:47␣CEST␣2015

```

when run with arg "4\_control\_io/more\_ifexamples.sh"

```

1 Dear␣user:␣Thanks␣for␣feeding␣me␣input
2 if␣!␣<␣"$1"␣grep␣"robot";␣then

```

when run with arg "/nonexistent"

```

1 Dear␣user:␣Thanks␣for␣feeding␣me␣input
2 Unknown␣command:␣/nonexistent

```

A general convention is to have tests in the if-list and actions in the then-list for clarity. Compare

```

1 if [ -f "file" ] && [ -d "dir" ] ; then
2     mv "$file" "dir" || exit 1
3     echo "Moved␣file␣successfully"
4 fi

```

and

```

1 if [ -f "file" ] && [ -d "dir" ] && mv "$file" "dir" || ↵
2     ↵exit 1; then
3     echo "Moved␣file␣successfully"
4 fi

```

It is easy to overlook the mv or the exit commands in such scripts.

## 4.4 Loops: while

while syntax:

```
1 while list1; do list2; done
```

- `list1` and `list2` are executed in turn as long as the last command in `list1` gives a zero return code.

```
1 #!/bin/bash
2
3 C=0
4 while echo "while:␣$C"; [ $C -lt 3 ]; do
5     ((C++)) #increase C by 1
6     echo $C
7 done
8
9 # a nested loop
10 N=5
11 while [ $N -gt 2 ]; do
12     ((N--)) #decrease N by 1
13     echo "N␣is␣now␣$N"
14     M=2
15     while [ $M -lt 4 ]; do
16         echo "␣␣␣␣M␣is␣now␣$M"
17         ((M++))
18     done
19 done
20
21 # more generally the statement
22 #     ((I++))
23 # increases the value of the variable I
24 # by one. Analogously
25 #     ((I--))
26 # decreases it by one.
```

4\_control\_io/whileloop.sh

produces the output

```
1 while:␣0
2 1
3 while:␣1
4 2
5 while:␣2
6 3
7 while:␣3
8 N␣is␣now␣4
9 ␣␣␣␣M␣is␣now␣2
10 ␣␣␣␣M␣is␣now␣3
11 N␣is␣now␣3
12 ␣␣␣␣M␣is␣now␣2
13 ␣␣␣␣M␣is␣now␣3
14 N␣is␣now␣2
15 ␣␣␣␣M␣is␣now␣2
16 ␣␣␣␣M␣is␣now␣3
```

We can stop the execution of a loop using the `break` command. This will only exit the innermost loop.

```

1 #!/bin/bash
2
3 C=0
4 while echo "while:␣$C"; [ $C -lt 3 ]; do
5     ((C++)) #increase C by 1
6     echo $C
7     [ $C -eq 2 ] && break
8 done
9
10 # a nested loop
11 N=5
12 while [ $N -gt 2 ]; do
13     ((N--)) #decrease N by 1
14     echo "N␣is␣now␣$N"
15     M=2
16     while [ $M -lt 4 ]; do
17         echo "␣␣␣␣M␣is␣now␣$M"
18         ((M++))
19         [ $M -eq 3 -a $N -eq 3 ] && break
20     done
21 done

```

4\_control\_io/whilebreak.sh

produces the output

```

1 while:␣0
2 1
3 while:␣1
4 2
5 N␣is␣now␣4
6 ␣␣␣␣M␣is␣now␣2
7 ␣M␣is␣now␣3
8 N␣is␣now␣3
9 ␣M␣is␣now␣2
10 N␣is␣now␣2
11 ␣M␣is␣now␣2
12 ␣M␣is␣now␣3

```

There also exists the command `continue` which jumps straight to the beginning of the next iteration, i.e. `list1` is evaluated once again and if it is true, `list2` and so fourth. The `continue` command allows to skip some instructions in a loop.

```

1 #!/bin/bash
2
3 C=0
4 while echo "while:␣$C"; [ $C -lt 3 ]; do
5     ((C++)) #increase C by 1
6     [ $C -eq 2 ] && continue
7     echo $C
8 done
9

```

```

10 # a nested loop
11 N=5
12 while [ $N -gt 2 ]; do
13     ((N--)) #decrease N by 1
14     echo "N_is_now_$N"
15     M=2
16     while [ $M -lt 4 ]; do
17         ((M++))
18         [ $M -eq 3 -a $N -eq 3 ] && continue
19         echo "____M_is_now_$M"
20     done
21 done

```

4\_control\_io/whilecontinue.sh

produces the output

```

1 while: 0
2 1
3 while: 1
4 while: 2
5 3
6 while: 3
7 N_is_now_4
8 ____M_is_now_3
9 __M_is_now_4
10 N_is_now_3
11 __M_is_now_4
12 N_is_now_2
13 __M_is_now_3
14 __M_is_now_4

```

**Exercise 4.4.** *optional* Write a script that takes two integer values as args, I and J. The script should:

- create directories named 1, 2, ..., I
- Use `touch` to put empty files named 1 till J in each of these directories
- Print an error if a negative value is provided for I or J
- If any of the files exist, the script should exit with an error.
- Provide help if one of the args is `-h`, then exit the script.
- If the third argument is a file, the script should copy this file to all locations instead of creating empty files with `touch`.

**Exercise 4.5.** Implement the `seq` command in `bash`:

- If called with a single argument, print all integers from 1 to this value, i.e.

```
1 seq 5
```

should give

```
1 1
2 2
3 3
4 4
5 5
```

- If called with two arguments, print from the first arg to the second arg, e.g. `seq 3 5`:

```
1 3
2 4
3 5
```

Assume that the first number is always going to be smaller or equal to the second number.

- *optional* If called with three arguments, print from the first arg to the third in steps of the second, in other words

```
1 seq 1 4 13
```

gives

```
1 1
2 5
3 9
4 13
```

Again assume that the first number is smaller or equal to the third one.

- Your script should print help if the first arguments is `-h`, and then exit.
- *optional* Your script should print an error if any of the assumptions is violated and exit.

## 4.5 Loops: for

Basic syntax:

```
1 for name in word ...; do list; done
```

- The variable `name` is subsequently set to all `words` following `in` and the `list` executed:

```
1 #!/bin/bash
2
3 for word in 1 2 dadongs blubber; do
4     echo $word
5 done
6
7 for row in 1 2 3 4 5; do
8     for col in 1 2 3 4 5; do
9         echo -n "$row.$col_"
10    done
11    echo
12 done
```

4\_control\_io/forbasic.sh

which gives the output

```
1 1
2 2
3 dadongs
4 blubber
5 1.1_1.2_1.3_1.4_1.5
6 2.1_2.2_2.3_2.4_2.5
7 3.1_3.2_3.3_3.4_3.5
8 4.1_4.2_4.3_4.4_4.5
9 5.1_5.2_5.3_5.4_5.5
```

- We can again use `break` or `continue` in order to skip some executions of the loops:

```
1 #!/bin/bash
2
3 for word in 1 2 dadongs blubber; do
4     echo "$word" | grep -q da && continue
5     echo $word
6 done
7
8 for row in 1 2 3 4 5; do
9     for col in 1 2 3 4 5; do
10        [ $col -gt $row ] && break
11        echo -n "$row.$col_"
12    done
13    echo
14 done
```

4\_control\_io/forbreakcontinue.sh



with output

```

1 1
2 2
3 blubber
4 1.1
5 2.1_2.2
6 3.1_3.2_3.3
7 4.1_4.2_4.3_4.4
8 5.1_5.2_5.3_5.4_5.5

```

### 4.5.1 Common “types” of for loops

As we said in the previous chapter, word splitting occurs right before the execution, i.e. basically after everything else. Therefore there is quite a large variety of expressions one could use after the `in` in `for` loops. This section gives an overview.

- Explicitly provided words: What we did in the examples above
- Parameter expansion

```

1 #!/bin/bash
2 VAR="a_b_c_d"
3 VAR2=$(< resources/matrices/3.mtx grep 1)
4 for i in $VAR $VAR2; do
5     echo $i    #note: all spaces become line breaks
6 done | head

```

4\_control\_io/forparameter.sh

```

1 a
2 b
3 c
4 d
5 1
6 1
7 1
8 1
9 2
10 1

```

- Command substitution

```

1 #!/bin/bash
2 N=10
3 for i in $(seq $N); do
4     echo $i
5 done

```

4\_control\_io/forcommandsubst.sh

```

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10

```

- The characters `*` and `?` are the special pattern characters. If replacement of `*` by zero or more arbitrary characters gives the name of an existing file, this replacement is done before execution. Similarly for `?`: This character is replaced by exactly one arbitrary character if this leads to the name of a file<sup>2</sup>. In the context of `for` loops this is usually encountered like this

```

1 #!/bin/bash
2 cd resources/matrices/
3 for i in *.mtx; do
4     echo $i
5 done
6
7 # there is no need for a file to be in pwd
8 for i in ../matrices/?a.mtx; do
9     echo $i
10 done
11
12 #NOTE: Non-matching strings still contain * or ?
13 for i in /non?exist*ant; do
14     echo $i
15 done

```

4\_control\_io/forwildcard.sh

```

1 3a.mtx
2 3_b.mtx
3 3.mtx
4 bcsstm01.mtx
5 lund_b.mtx
6 ../matrices/3a.mtx
7 /non?exist*ant

```

- Combinations of all of these

A word of warning: The paradigm

```

1 for file in $(ls); do
2     # some stuff with $file
3 done

```

<sup>2</sup>This process is called *pathname expansion* and a few other patterns exist as well. See [2] for details.

is extremely problematic, since files with spaces are not properly accounted for<sup>3</sup>. Compare the following results with the last example we had above

```
1 #!/bin/bash
2 for i in $(ls resources/matrices/*.mtx); do
3     echo $i
4 done
```

4\_control\_io/forlscommandsubst.sh

```
1 resources/matrices/3a.mtx
2 resources/matrices/3
3 b.mtx
4 resources/matrices/3.mtx
5 resources/matrices/bcsstm01.mtx
6 resources/matrices/lund_b.mtx
```

**Exercise 4.6.** With this exercise we start a small project trying to recommend a book from Project Gutenberg based on keywords the user provides.

- Write a script that **greps** for a pattern (provided as an argument) in all books of **resources/gutenberg**
  - Make sure that your script keeps working properly if spaces in the pattern or in the files are encountered
  - Ignore case when **grepping** in the files
  - You may assume all books of Project Gutenberg to be **.txt** files
  - *optional* Provide help if the argument is **-h**
  - *optional* Use proper error statements if something goes wrong or is not sensible.
- Change your script such that it prints the number of matches and the number of actual lines next to the script name. The fields of the table should be separated by tabs (use **echo -e**). A possible output could be

```
1 pg74.txt__45__1045
2 pg345.txt__60__965
```

- *optional* Suppress the output of books without any match

**Exercise 4.7.** *optional* With your current knowledge of **bash**, propose two one liners that

- substitute all **<tab>** or **<space>** of a string in a variable **VAR** by **<newline>** characters
- substitute all **<newline>** or **<tab>** characters by **<space>** characters

Hint: Both expressions have less than 30 characters.

<sup>3</sup>The reason is that command substitution happens earlier than pathname expansion: The results of the command substitution **\$(ls)** go through word splitting before being executed, whereas the results of **\*-** and **?-** expressions are still seen as single words at the execution stage. See appendix B.3.1 on page 123 for more details.

## 4.6 Conditionals: case

The `case` command has the following basic syntax:

```

1 case word in
2   pattern) list ;;
3   [ pattern) list ;; ]
4   ...
5 esac

```

- The command tries to match `word` against one of the `patterns` provided
- If a match occurs the respective `list` block is executed
- Both the `word` as well as the inspected `patterns` are subject to parameter expansion, command substitution, arithmetic expansion and a few others [2]

⇒ We may have variables and commands in both `word` and `pattern`.

Usually in `case` statements we have a string containing a variable and we want to distinguish a few cases, e.g.

```

1 #!/bin/bash
2 VAR=$@      # VAR assigned to all arguments
3 case $VAR in
4   a)  echo "VAR_is_\`a\`"
5       ;; #<- do not omit these
6   l*) echo "VAR_starts_with_l"
7       ;;
8   l?) echo "VAR_is_l_and_something"
9       echo "Never_matched"
10      # because it is more specific
11      # than pattern l* above
12      ;;
13   $1) echo "VAR_is_\`$1\`"
14       ;;
15   *)  echo "VAR_is_something_else"
16       ;;
17 esac

```

4\_control\_io/caseexample.sh

The output is

- 4\_control\_io/caseexample.sh lo

```
1 VAR_starts_with_l
```

- 4\_control\_io/caseexample.sh

```
1 VAR_is_$1
```

- 4\_control\_io/caseexample.sh "bash\_is"so cool

```
1 VAR_is_something_else
```

- 4\_control\_io/caseexample.sh unihd

```
1 VAR_is_$1
```

The `case` command is extremely well-suited in the context of parsing commandline arguments. A very common paradigm is `while-case-shift`<sup>4</sup>

```
1 #!/bin/bash
2 # assume we allow the arguments -h, -f and --show
3 # assume further that after -f there needs to be a
4 # filename following
5 #
6 FILE=default_file # default if -f is not given
7 while [ "$1" ]; do # are there commandline arguments left?
8     case "$1" in # deal with current argument
9         -h|--help) echo "-h encountered"
10                ;;
11         # it is common to have "long" and "short" options
12         -f|--file) shift # access filename on $1
13                   echo "-f encountered, file: $1"
14                   FILE=$1
15                   ;;
16         --show)   echo "--show encountered"
17                ;;
18         *)        echo "Unknown argument: $1" >&2
19                   exit 1
20     esac
21     shift # discard current argument
22 done
```

4\_control\_io/argparsing.sh

- The `shift` command shifts the positional parameters one place forward. After the execution: `$1` contains the value `$2` had beforehand, equally `3→2`, `4→3`, ...
- The `while` loop runs over all arguments in turn, `$1` always contains the argument we currently deal with.
- `case` checks the current argument and takes appropriate action.
- If a flag (like `-f` in this case) requires a value afterwards, we can access this value by issuing another `shift` in the code executed for `-f` in `case`.

Example output

- 4\_control\_io/argparsing.sh -h --show

```
1 -h encountered
2 --show encountered
```

---

<sup>4</sup>no official name, but my own creation :)

```
• 4_control_io/argparsing.sh -f file --sho
```

```
1 -f encountered, file: file
2 Unknown argument: --sho
```

**Exercise 4.8.** Write a script that takes the following arguments:

- -h, -q
- --help, --quiet
- -f followed by a filename
- anything else should cause an error message

Once the arguments are parsed the script should do the following

- Print help if -h or --help are present, then exit
- Check that the filename provided is a valid file, else throw an error and exit
- Print a nice welcome message, unless --quiet or -q are given

## 4.7 Parsing input using shell scripts

### 4.7.1 The read command

The syntax to call `read` is

```
1 read <Options> NAME1 NAME2 NAME3 ... NAME_LAST
```

- `read` reads a single line from *stdin* and performs word splitting on it. The first word is assigned to the variable `NAME1`, the second to `NAME2`, the third to `NAME3` and so on. All remaining words are assigned to the last variable as a single unchanged word.

**Example 4.9.** The first line of `resources/matrices/3.mtx` is

```
1 %%MatrixMarket matrix coordinate real symmetric
```

So if we execute

```
1 #!/bin/bash
2 < resources/matrices/3.mtx read COMMENT MTX FLAGS
3 echo "com: $$$COMMENT"
4 echo "mtx: $$$MTX"
5 echo "flags: _$FLAGS"
```

4\_control\_io/readexample.sh

we obtain

```
1 com: $$$%%MatrixMarket
2 mtx: $$$matrix
3 flags: _coordinate_real_symmetric
```

- Two options worth mentioning:
  - `-p STRING`: Print `STRING` before waiting for input — like a command prompt.
  - `-e`: Enable support for navigation through the input terminal and some other very comfortable things.
- The return code of `read` is 0 unless it encounters an EOF (end of file), i.e. unless the stream contains no more data.

By means of the return code of `read` we can check easily whether we were able to obtain *any* data from the user or not. We cannot check with the return code, however, whether all fields are filled or not.

```

1 #!/bin/bash
2 while true; do      #infinite loop
3     # the next command breaks the loop if it was successful
4     read -p "Please type 3 numbers >" N1 N2 N3 && break
5     # if we get here read was not successful
6     echo "Did not understand your results, please try again"
7 done
8 echo "You entered \"$N1\", \"$N2\", \"$N3\""
```

4\_control\_io/readerror.sh

- Running `echo 1 2 3 | 4_control_io/readerror.sh`

```
1 You entered "1", "2", "3"
```

- `echo | 4_control_io/readerror.sh`, i.e. send only a <newline>.

```
1 You entered "", "", ""
```

- `echo -n | 4_control_io/readerror.sh`, i.e. send absolutely nothing

```

1 Did not understand your results, please try again
2 Did not understand your results, please try again
3 ...
4 Did not understand your results, please try again
```

### 4.7.2 Scripts have shared *stdin*, *stdout* and *stderr*

Compared to writing simple one-liners there is a fundamental difference when writing a script: All commands of the script share the same *stdin*, *stdout* and *stderr* (if their input/output is not redirected). Especially when it comes to parsing *stdin*, this has a few consequences, which are best described by examples.

**Example 4.10.** Consider the script

```

1 #!/bin/bash
2 cat
3 cat
```

4\_control\_io/cat\_script.sh

If we run it like so

```
1 < resources/matrices/3.mtx 4_control_io/cat_script.sh
```

we might expect the output to show the content of the input file twice. This is not what happens. We only get the content of `resources/matrices/3.mtx` once, i.e. exactly what would have happened if only a single `cat` was contained in `4_control_io/cat_script.sh`. This is due to the fact that `cat` reads *stdin* until nothing is left (i.e. until EOF is reached). So when the next `cat` starts its execution, it encounters the EOF character straight away and stops reading. Hence no extra output is produced.

The same thing occurs if we use two other commands that keep reading until the EOF, like two consecutive `greps`:

```
1 grep match
2 grep "i_will_never_match_anything"
```

the second `grep` is pointless. If subsequent `greps` on *stdin* are desired, one usually employs a temporary caching variable in order to circumvent these problems:

```
1 CACHE=$(cat)
2 echo "$CACHE" | grep match
3 echo "$CACHE" | grep "i_have_a_chance_to_match_sth."
```

**Example 4.11.** In contrast to `cat` the `read` only reads a single line. Therefore a script may swap the first two lines of *stdin* like this

```
1 #!/bin/bash
2 read OLINE      # read the first line
3 read LINE       # read the second line
4 echo "$OLINE"   # print second line
5 echo "$LINE"    # print first line
6 cat
```

4\_control\_io/swaplines.sh

where the last `cat` just print whatever is left of the file.

**Exercise 4.12.** Write a simple script `read_third.sh` that outputs the third line provided on *stdin* to *stdout* and the fourth line to *stderr*. When you call it like

```
1 < resources/testfile ./read_third.sh
```

it should provide the output

```
1 some
2 other
```

and when called like

```
1 < resources/testfile ./read_third.sh >/dev/null
```

it should only print

```
1 other
```



**Exercise 4.13.** Extend the script from the previous exercise:

- Use `read` to ask the user for two line numbers, `N` and `M`.
- Print the `N`th line of the script's *stdin* to *stdout* and the `M`th line to *stderr*
- Call your script from the shell and use input redirection `<` in order to pass some data from a file to the script's *stdin*.
- Does the script work as expected? Why not?

### 4.7.3 The while read line paradigm

Probably the most important application of the `read` command is the `while read line` paradigm<sup>5</sup>. It can be used to read data from *stdin* line by line:

```
1 #!/bin/bash
2 while read line; do
3     echo $line
4 done
```

4\_control\_io/whilereadline.sh

This works because

- `read` tries to read the current line from *stdin* and stores it in the variable `line`.
- The `line` variable is then available for the loop body to do something with it.
- If all data has been read, `read` will exit with a return code 1, causing the loop to be exited.

Since a loop is considered as a single command by the `bash` shell it has its own *stdin* (and *stdout*), meaning that

- we can redirect its *stdin* to read from a file

```
1 #!/bin/bash
2
3 if [ "$1" == "-h" ]; then
4     echo "Script adds line numbers to a file on \$1"
5     exit 1
6 fi
7
8 if [ ! -f "$1" ]; then
9     echo "File $1 not found" >&2
10    exit 1
11 fi
12
13 C=0
14 while read line; do
15     echo "$C: $line"
16     (( C++ ))
17 done < "$1"
```

4\_control\_io/addlinenumbers.sh

---

<sup>5</sup>Again not an official name

Note: The < input arrow has to be added *after* the **done** — otherwise an error results.

- we can pipe the output of a command to it

```

1  #!/bin/bash
2  if [ "$1" == "-h" ]; then
3      echo "Script sorts lines of file \"$1\" and adds ↵
      ↵ indentation"
4      echo "Sorted file is written to \"$1.sorted"
5      exit 1
6  fi
7
8  if [ ! -f "$1" ]; then
9      echo "File \"$1\" not found" >&2
10     exit 1
11 fi
12
13 echo "Writing sorted data to \"$1.sorted\"
14 < "$1" sort | while read line; do
15     echo "    $line"
16 done > "$1.sorted"

```

4\_control.io/sort\_and\_indent.sh

- we can dump the loop's output in a file by adding > file after the **done** (see previous example)

**Exercise 4.14.** *optional* We want to write a more general version of exercise 3.3 on page 26.

- Write a script takes the arguments **--help**, **--from** (followed by a line number) and parses them. Deal with **--help** and detect unknown arguments.
- The default for **--from** should be the first line.
- Move the line of *stdin* given by **--from** to the last line on *stdout*, copy all other lines.
- You may assume that the users of your script are nice and only pass integer values after **--to** or **--from**.
- If an error occurs, e.g. if the **--to** line number is larger than the number of lines on *stdin*, inform the user.
- Now add an argument **--to**, which is followed by a number. It should have the default setting of **"end"** (symbolising the last line on *stdin*)
- Assume (and check the input accordingly) that the value given to **--to** is larger than the value to **--from**
- Change your code such that the line **--from** is moved to the line **--to**.

option	description
<code>-name "STRING"</code>	The name of the file is string
<code>-name "*STRING*"</code>	The name of the file contains string
<code>-iname "*STRING*"</code>	Same as above, but ignore case
<code>-type f</code>	file is a normal file
<code>-type d</code>	file is actually a directory

Table 4.3: The most important options of find

- Be careful when comparing line numbers to variables that may contain a string:

```
1 [ "end" -eq 4 ]
```

gives an error. This can be circumvented by guarding the `[` with another `[`, e.g.

```
1 VAR="end"
2 [ "$VAR" != "end" ] && [ $VAR -eq 4 ]
```

**Exercise 4.15.** Recall that command substitution expressions combine the output of all internal commands. Therefore we can accumulate lines in a variable using the syntax

```
1 CACHE=$(echo "$CACHE"; echo "next_line")
```

Use this fact and the `while read line` paradigm to build a simple version of the `tac` command, where all input on *stdin* is printed to *stdout* in reverse line order

**Exercise 4.16.** Recall that `read` can take more than one argument.

- Assume you will get some data on *stdin*, which consists of a few columns separated by one or more `<space>` or `<tab>` characters. Write a script `mtx_third.sh` that prints the third column of everything you get on *stdin*.
- Try your script on some of the files in `resources/matrices`. E.g.

```
1 < resources/matrices/lund_b.mtx ./mtx_third.sh
```

- How does it perform compared to `cut`?

**Exercise 4.17.** *optional* `find` is a really handy program to search for files and directories with uncountable options (see `man find`). You can find the most important options in table 4.3. `find` per default searches through all directories and subdirectories and prints the relative paths of all files satisfying the conditions to *stdout*. All options you provide are connected using a logical *and*. This can of course all be changed (see documentation). If you have never used `find` before, try the following:

- `find -name "*.sh"`
- `find -type f -name "*.sh"`
- `find $HOME -type d -name "*bash*"`

In this exercise you should build a `grep_all` script:

- The script should search for all files in or below the working directory (using `find`)
- In all files found, the script should `grep` for the pattern provided on `$1` and it should print to `stdout` in which files and on which line the match occurred.
- The simplest way to achieve this is to pipe the output of `find` to `while read line`

## 4.8 Influencing word splitting: The variable IFS

In table 3.1 on page 23 we already mentioned the variable `IFS`.

- `IFS` is short for “internal field separator”
- This variable is considered in the word splitting step after parameter and command substitution
- Its value gives exactly the characters at which commandline is split into individual words
- Default value: `<space><tab><newline>`

Two important use cases, which alter the `IFS` variable temporarily:

- Manipulation of the way `for` loops iterate:

```
1 #!/bin/bash
2 OIFS=$IFS
3 IFS="+"
4 VAR="4+5+6+7"
5
6 # before the for loop runs the value after the "in"
7 # is subject to word splitting
8 echo first loop
9 for number in $VAR; do
10     echo $number
11 done
12 echo
13
14 # it is good practice to change IFS back to the
15 # original after you used the trick, otherwise
16 # all sorts of crazy errors can occur
17 IFS=$OIFS
18
19 echo second loop
20 for i in 1 2 3 4; do
21     # this works now as intuitively expected:
22     echo $i
23 done
```

4\_controlIo/IFS\_for.sh

```
1 first_loop
2 4
3 5
4 6
5 7
6
7 second_loop
8 1
9 2
10 3
11 4
```

- Influencing `read`:

```
1 #!/bin/bash
2
3 ARG="foo"
4 VAL="bar"
5 COMMENT="Some_crazy_comment"
6
7 # here we run code to determine the values of
8 # ARG, VAL, COMMENT
9
10 # store it for later usage in a more compact form
11 STORAGE="$ARG+$VAL+$COMMENT"
12
13 # ...
14
15 # unpack it again
16 OIFS=$IFS
17 IFS="+"
18 echo "$STORAGE" | {
19     read ARG VAL COMMENT
20     echo "The_argument_was_$ARG"
21     echo "The_value_was_$VAL"
22     echo "The_comment_was_$COMMENT"
23 }
24 # see next chapter why we need the { ... }
25 # ignore it for now
26 IFS=$OIFS
```

4\_control\_io/IFSread.sh

```
1 The_argument_was_foo
2 The_value_was_bar
3 The_comment_was_Some_crazy_comment
```

**Exercise 4.18.** The shell uses the following procedure to lookup the path of the commands to be executed<sup>6</sup>:

- In a commandline the first word is always considered to be the command.
- If this word is a path (contains a “/”) execute this very file.
- Else go through all existing directories in the variable `PATH`. The directories are separated using the character “:”. If there exists a file named like the command in a directory, which is executable as well, execute this file.
- Else keep searching in the next directory in `PATH`

Example: The commandline

```
1 vim testfile
```

has the first word/command `vim`. Consider

```
1 PATH="/usr/local/bin:/usr/bin:/bin"
```

a lookup reveals that the file `/usr/bin/vim` exists and is executable. So this file is executed.

There exists a tool, called `which`, that does exactly this lookup when provided with a command as its first argument. See `man which` for more details. We want to rebuild the `which` command as a script.

- Take the name of a command on `$1`
- Go through all existing directories in `PATH` and try to find an executable file called `$1` in these.
- If it exists print the full path and return 0
- Else return 1

Hints:

- Try to go through all directories in `PATH` first. There is an easy way to do this with one of the loops we discussed and `IFS`-manipulation
- Read the documentation of `test` in order to find out how to test if a file is executable.

## 4.9 Conventions when scripting

To conclude this chapter I have collected a few notes about conventions that I use when writing shell scripts. Some rules are loosely based on the Unix philosophy [1], but most of it comes from my personal experience. Some things I mention here seem tedious, but I can assure you these things pay back at some point. Either because you need less time to look stuff up or because you spot errors more quickly or because they make it easier to reuse scripts at a later point in time.

There are as usually many exceptions to each of the guidelines below. In practice try to follow each guideline, unless you have a good reason not to.

<sup>6</sup>This is a slight simplification since e.g. commandlines can be far more complex.

### 4.9.1 Script structure

- Have a **shebang**. Dot.
- A block of code doing a task should have a comment explaining what happens, what goes in and what comes out. This is especially true for functions (see section 6.2 on page 74).
- Whenever funny **bashisms** are used that could make code unclear, explain what happens.
- One script should only do one job only. Split complicated tasks into many scripts. This makes it easier to code and easier to reuse.
- Use shell functions (see section 6.2 on page 74) to structure your script. Have a comment what each function does.

### 4.9.2 Input and output

- Reserve *stdin* for data: Do not use the **read** command to ask the user for data or parameters, much rather use argument parsing for this. This makes the scripts more flexible.
- Use helpful error messages with as much info as possible. Print them to *stderr*
- Reserve *stderr* for errors, *stdout* for regular output. If you need to output two separate things, have the more important one printed to *stdout*, the other into a file. Even better: Allow the user to choose what goes into the file and what to *stdout*.

⇒ Can be summarised as “Design each script as a filter”

- Use **mktemp** for temporary files and clean the mess up afterwards (see section 6.3 on page 83)

### 4.9.3 Parsing arguments

- Each script should support the arguments **-h** or **--help**. If these arguments are provided, explain what the script does and explain at least the most important commandline arguments it supports.
- For each argument there should be a descriptive “long option” preceded by two “--”. There may be short options (preceded by one “-”).
- Do not worry about the long argument names. You can code tab completion (see section B.1.2 on page 123) for your script.

## Chapter 5

# Arithmetic expressions and advanced parameter expansions

In this chapter we will expand on two topics we already briefly touched: Arithmetic expansion and parameter expansion (in section 3.2 on page 22).

### 5.1 Arithmetic expansion

The arithmetic expansion is a simple, yet extremely convenient way to perform calculations directly in the **bash**. Arithmetic expressions have the syntax

```
1 ((expression))
```

Everything within the brackets is subject to *arithmetic evaluation*<sup>1</sup>:

- The expression may be split into subexpressions using the comma ,

```
1 ((1+2,4-4))
```

- The full range of parameter expansion expressions is available (see section 5.3 on page 64). One may, however, also access or assign variables without the leading \$

```
1 VAR=4
2 OTHER=3
3 LAST=2
4 (( LAST=VAR+$OTHER ))
5 echo $LAST
```

```
1 7
```

- Note: Positional parameters are *not* available

---

<sup>1</sup>The precise rules are more or less identical to the rules of the C programming language



- All common operators are available:
  - + - addition, subtraction
  - \* / % multiplication, (integer) division, remainder
  - \*\* exponentiation
  - name++ ++name name-- --name increment and decrement operators
  - += -= \*= /= %= Infix assignment

```

1 #!/bin/bash
2 ((
3     C=1,
4     D=2,
5
6     SUM=C+D,
7     DIV=C/D,
8     MOD=C%D,
9     EXP=D**4
10 ))
11 echo "C: $\C"
12 echo "D: $\D"
13 echo
14 echo "SUM=C+D: $\SUM"
15 echo "DIV=C/D: $\DIV"
16 echo "MOD=C%D: $\MOD"
17 echo "EXP=D**4: $\EXP"
18
19 ((
20     CAFTER=C++,
21     DAFTER=--D
22 ))
23 echo "C: $\C"
24 echo "D: $\D"
25 echo "CAFTER: $\CAFTER"
26 echo "DAFTER: $\DAFTER"

```

5\_variables/arith\_operator\_ex.sh

```

1 C: 1
2 D: 2
3
4 SUM=C+D: 3
5 DIV=C/D: 0
6 MOD=C%D: 1
7 EXP=D**4: 16
8 C: 2
9 D: 1
10 CAFTER: 1
11 DAFTER: 1

```

- Brackets ( and ) can be used with their usual meaning

- Comparison and logic operators are available as well:

- `==` `!=` equality, inequality
- `<=` `>=` `<` `>` se, ge, smaller, greater
- `&&` `||` logical AND and logical OR

Internally “true” is represented by 1 and “false” by 0 (like in C)

```
1 #!/bin/bash
2 ((4==4)); echo $?
3 ((4!=4)); echo $?
4 ((3<4 && 4!=4)); echo $?
5 ((A= 4==4+4)); echo $A
```

5\_variables/arith\_logic.ex.sh

```
1 0
2 1
3 1
4 0
```

- Expressions evaluating to 0 are considered to be false, i.e. their return code is 1.

```
1 (( 0 )) ; echo $?
```

```
1 1
```

- Expressions evaluating to another value are true, i.e. return with 0.

```
1 (( -15 )) ; echo $?
```

```
1 0
```

Especially the last two point seem a little strange at first, but they assure that arithmetic expressions can be used as a replacement for `test` in `while` or `if` constructs

```
1 #!/bin/bash
2
3 C=1
4 while ((++C<40)); do
5     if ((C%3 == 0)); then
6         echo "I can be divided by 3: $C"
7     fi
8 done
```

5\_variables/arith\_replacement.sh

```
1 I can be divided by 3: 3
2 I can be divided by 3: 6
3 I can be divided by 3: 9
4 I can be divided by 3: 12
5 I can be divided by 3: 15
```

```

6 I can be divided by 3: 18
7 I can be divided by 3: 21
8 I can be divided by 3: 24
9 I can be divided by 3: 27
10 I can be divided by 3: 30
11 I can be divided by 3: 33
12 I can be divided by 3: 36
13 I can be divided by 3: 39

```

By the means of the arithmetic evaluation the `bash` also supports a C-like `for` loop with the syntax

```
1 for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

- `expr1`, `expr2` and `expr3` all have to be arithmetic expressions.
- First `expr1` is evaluated
- Then `expr2` is repeatedly evaluated until it gives zero (“C-false”)
- For each successful evaluation both the `list` is executed as well as `expr3`.

```

1 #!/bin/bash
2 MAX=4
3 for((I=0; I<MAX; ++I)); do
4     echo $I
5 done
6 echo
7 for((I=MAX-1; I>=0; --I));do
8     echo $I
9 done

```

5\_variables/arith\_for\_cloop.sh

```

1 0
2 1
3 2
4 3
5
6 3
7 2
8 1
9 0

```

Finally *arithmetic expansion* is invoked by a syntax like

```
1 $((expression))
```

- `expression` is subject to arithmetic evaluation as described above
- The whole construct is replaced by the final value the `expression` results in.
- The return code of `(( ))` is not available.
- The expression may be used just like an parameter expansion `${VAR}`

```

1 #!/bin/bash
2 N=$1
3 echo "You kindly supplied: $N"
4 echo "The square is: $((N*N))"
5 echo "I can add some stuff: $((1+1,2+N,N+3))"

```

5\_variables/arith\_expansion.sh

```

1 You kindly supplied: 5
2 The square is: 25
3 I can add some stuff: 8

```

A big drawback on all these paradigms is that the **bash** only supports integer arithmetic. Even for intermediates there is only integer precision available, e.g.

```

1 #!/bin/bash
2 echo $((100*13/50))
3 echo $((13/50*100))

```

5\_variables/arith\_intermediate\_floats.sh

```

1 26
2 0

```

Hence the order in which expressions are entered can sometimes become very important.

Whenever floating point arithmetic is needed one needs to use one of the tricks discussed in section 5.2 on the next page.

**Exercise 5.1.** What is the return code of the following expressions and why?

```

1 %TODO have a few easier ones like (( 3-4 )), (( 0*4, 3 ))
2 ((B=0))
3 echo $((B=0))
4 echo $((B=0)) | grep 0
5 for((C=100,A=99 ; C%A-3 ; C++,A-- )); do ((B=(B+1)%2)) ;
  ↪ done; ((B))
6 ((B=1001%10)) | grep 4 || (( C=$(echo "0"|grep 2)+4, 2%3 ))
  ↪ && echo $((4-5 && C-3+B)) | grep 2

```

Last two are *optional*.

**Exercise 5.2.** For the arithmetic expansion an empty variable or a string that cannot be converted to an integer counts as zero (“C-false”)

- Try this in a shell or in a script, e.g. execute the following:

```

1 A="string"
2 echo $((A+0))
3 A="4"
4 echo $((A+0))

```

contrast this with

```

1 A="string"
2 echo $A
3 A="4"
4 echo $A

```

- How could this behaviour (together with the `[]` program) be exploited to test whether an input parameter can be properly converted to an integer?
- Write a script that calculates the cube of `N`, where `N` is an integer supplied as the first argument to your script. Of course you should check that `N` is a sensible integer before entering the routine.

**Exercise 5.3.** *optional* Use `bash` arithmetic expressions to calculate all primes between 1 and `N`, where `N` is a number supplied as the first argument to your script.

## 5.2 Non-integer arithmetic

Non-integer arithmetic, i.e floating point computations, cannot be done in plain `bash`. The most common method is to use the `bc` terminal calculator, like so

```
1 # echo expression | bc -l
2 echo "13/50*100" | bc -l
```

```
1 26.00000000000000000000000000
```

The syntax is more or less identical to the arithmetic expansion, including the C-like interpretation of true and false

```
1 echo "3<4" | bc -l # gives true
2 echo "1_==_42" | bc -l # gives false
```

```
1 1
2 0
```

A minor difference is that `^` is used instead of `**` in order to denote exponentiation.

```
1 echo "3^3" | bc -l
```

```
1 27
```

The format of the output can be changed using a few flags (see manpage of `bc`).

- For example one can influence the base (2,8,10 and 16 are supported)

```
1 echo "obase=2; 2+4" | bc -l
```

```
1 110
```

- or the number of decimal figures

```
1 echo "scale=4; 5/6" | bc -l
```

```
1 .8333
```

Next to `bc` one can in principle also use any other floating-point aware program like `awk` (see chapter 8 on page 99) or `python`. Most of the time it is, however, still sensible to use `bc`, since it is extremely, i.e. quick to start up.

**Exercise 5.4.** Now we want to extend our project to recommend books from Project Gutenberg. Recall that your script from exercise 4.6 on page 44 gives output of the form

```
1 pg74.txt__45__1045
2 pg345.txt__60__965
```

where the columns were separated by tabs. The second column was the number of matches and the third column was the number of actual lines in the file. Write a script that

- takes one pattern as an argument, which is then used to call the script from exercise 4.6 on page 44
- parses the respective script output
- calculates for each book the relative importance given as

$$\xi = \frac{\text{Number of matching lines}}{\text{Number of actual lines}}$$

and writes this  $\xi$ -value and the book name to a temporary file. To make the next steps easier you should separate the value and the book name by a `<tab>` and have the  $\xi$ -value in the first and the book name in the second column.

- *optional* sorts the temporary file according to the relative importance
- *optional* suggests the 3 best-scoring books for the user and gives their score.
- *optional* One can entirely omit writing to a temporary file. Try this in your script.

Try a few patterns, e.g. “Baker”, “wonder”, “the”, “virgin”, “Missouri, Kentucky”. Any observations?

**Exercise 5.5.** Write a script that takes either the argument `-m` or `-s`, followed by as many numbers as the user wishes. The script should

- Calculate the sum of all numbers if `-s` is provided
- *optional* The mean if `-m` is provided
- *optional* Give an error if neither `-m` nor `-s` are given.

Some ideas:

- In both cases you will need to calculate the sum, so try to get that working first.
- As you know `bc` evaluates expressions given to it on *stdin*, so try to build an appropriate sum expression from all commandline arguments using a loop. This you `echo` to `bc` in order to get the sum.
- You may assume that users are nice and will only provide valid strings as the number arguments to your script.

**Exercise 5.6.** *optional* Read about the `mtx` format in appendix C.1 on page 125.

- Write a script that takes a `mtx` file on *stdin* and a number on `$1`.
- The output should be again a valid `mtx` file where all entries are multiplied with said number.
- The comment in the first line (but not necessarily any other) should be preserved
- You can assume that both the data you get on *stdin* as well as the number on `$1` are sensible.

Try your script on `resources/matrices/3.mtx` and `resources/matrices/3b.mtx`, since unfortunately not all `mtx` files will work with this method.

### 5.3 A second look at parameter expansion

Parameter expansion is much more powerful than just returning the value of a parameter. An overview:

- *assign-default*

```
1 ${parameter:=word}
```

If `parameter` is unset or null, set `parameter` to `word`. Then substitute the value of `parameter`. Does not work with positional parameters

- *use-default*

```
1 ${parameter:-word}
```

If `parameter` is unset or null, substitute `word`, else the value of `parameter`

- *use-alternate*

```
1 ${parameter:+word}
```

If `parameter` is unset or null, nothing is substituted, else `word` is substituted.

```
1 #!/bin/bash
2
3 A=
4 B=3
5
6 echo ${B:+ "B works "}
7 echo ${A:+ "A works "}
8 echo ${A:- "notA: " $B}
9
10 echo ${A:= "defined "}
11 echo ${A:+ "A works "}
12 echo ${A:- "notA: " $B}
```

5\_variables/pexp\_use.sh

```

1 B_works
2
3 notA:_3
4 defined
5 A_works
6 defined

```

- *substring expansion*

```

1 ${parameter:offset}
2 ${parameter:offset:length}

```

Expands into up to `length` characters from `parameter`, starting from character number `offset` (0-based). If `length` is omitted, all characters starting from `offset` are printed. Both `length` and `offset` are arithmetic expressions

- *parameter length*

```

1 ${#parameter}

```

Expands into the number of characters `parameter` currently has.

```

1 #!/bin/bash
2
3 VAR="some_super_long_string"
4 LEN=${#VAR}
5 echo $LEN
6
7 # remove first and last word:
8 echo ${VAR:4:LEN-10}
9
10 # since parameter expansion is allowed
11 # in arithmetic expressions
12 echo ${VAR:2+2:${#VAR}-10}

```

5\_variables/pexp\_length.sh

```

1 22
2 super_long
3 super_long

```

- *pattern substitution*

```

1 ${parameter/pattern/string} # one occurrence
2 ${parameter//pattern/string} # global

```

`parameter` is expanded and the *longest* match of `pattern` is replaced by `string`. Normally only the first match is replaced. If the second — global — version is used, however, all occurrences of `pattern` are replaced by `string`.



```

1 #!/bin/bash
2 VAR="some_super_long_string"
3 PATTERN="s*e"
4 PATTERN2="?r"
5 REPLACEMENT="F0000"
6
7 # the longest match is replaced:
8 echo ${VAR/$PATTERN/$REPLACEMENT}
9 echo ${VAR/$PATTERN2/$REPLACEMENT}
10
11 # all matches are replaced
12 echo ${VAR//$PATTERN2/$REPLACEMENT}

```

5\_variables/pexp\_subst.sh

```

1 F0000r_long_string
2 some_supF0000_long_string
3 some_supF0000_long_sF0000ing

```

**Exercise 5.7.** Implement the `rev` command in `bash`:

- Read input provided on *stdin* line by line.
- For each line reverse the characters, i.e.

test    →    tset    abcdef    →    fedcba

- Print the reversed string to *stdout*

Hints:

- The string reversal can be easily achieved using the substring expansion: By using a length of 1 we can design an inner loop to extract one character after another from the string.
- The new reverted string can then be built from these characters.

## Chapter 6

# Subshells and functions

This chapter is concerned with useful features the `bash` provides in order to give scripts a better structure and make code more reusable.

### 6.1 Explicit and implicit subshells

#### 6.1.1 Grouping commands

Multiple commands can be grouped using the syntax

```
1 { list; }
```

- A line break or ; in the end is crucial
- All commands in the list share the same *stdin*, *stdout* and *stderr*.
- The return code is the return code of the last command in list.

The syntax is e.g. useful for

- Unpacking data

```
1 #!/bin/bash
2 < resources/matrices/3.mtx grep -v "%" | {
3   read ROW COL ENTRIES
4     echo "Number_of_rows:        $ROW"
5     echo "Number_of_cols:        $COL"
6   echo "Number_of_entries:    $ENTRIES"
7   echo "List_of_all_entries:"
8   while read ROW COL VAL; do
9     echo "    M($ROW,$COL)=$VAL"
10  done
11 }
```

6.functions\_subshells/group\_unpack.sh

```

1 Number_of_rows:uuuuuu3
2 Number_of_cols:uuuuuu3
3 Number_of_entries:uuu9
4 List_of_all_entries:
5 uuuM(1,1)u=u1
6 uuuM(1,2)u=u1
7 uuuM(1,3)u=u1
8 uuuM(2,1)u=u2
9 uuuM(2,2)u=u2
10 uuuM(2,3)u=u2
11 uuuM(3,1)u=u3
12 uuuM(3,2)u=u3
13 uuuM(3,3)u=u3

```

- Sending data to a file

```

1 #!/bin/bash
2
3 {
4     echo "Crazy_header"
5     echo
6     echo "A_first_message_to_stderr" >&2
7     echo "I_want_fish" | grep -w fish
8     echo "lorem_ipsum_dolor_sit_amet"
9     echo "This_goes_to_the_stderr" >&2
10 } > /tmp/some-file-here 2> /tmp/file-stderr
11
12 # print content
13 echo Everything on the first file:
14 echo -----
15 cat /tmp/some-file-here
16 echo -----
17 echo
18 echo "Everything_on_the_second_file:"
19 echo -----
20 cat /tmp/file-stderr
21 echo -----
22
23 # cleanup
24 rm /tmp/some-file-here /tmp/file-stderr

```

6\_functions\_subshells/group\_write\_file.sh

```

1 Everything_on_the_first_file:
2 -----
3 Crazy_header
4
5 I_want_fish
6 lorem_ipsum_dolor_sit_amet
7 -----
8
9 Everything_on_the_second_file:
10 -----
11 A_first_message_to_stderr
12 This_goes_to_the_stderr
13 -----

```

- There surely exist alternatives we could use in order to write many lines of data to a file, e.g. instead of

```

1 {
2     echo line1
3     echo line2
4     echo line3
5 } > /tmp/file

```

we could also use

```

1 echo line1 > /tmp/file
2 echo line2 >> /tmp/file
3 echo line3 >> /tmp/file

```

The latter method has a few disadvantages, however:

- One easily forgets one of the >> or > operators at the end
- One easily mixes up > and >> when typing the code. So some of the stuff gets overwritten.
- If we want to rearrange the order in which the data gets written at a later point we need to be careful to change the > and >> as well in the appropriate lines.

### 6.1.2 Making use of subshells

Subshells are special environments within the current executing shell, which work very similar to command grouping. Their special property is that all changes to the so-called *execution environment* are only temporary. The execution environment includes

- The current working directory
- The list of defined variables and their values

Once the subshell exits all these changes are undone, i.e. the main shell's *execution environment* is restored. Invocation syntax:

```

1 ( list )

```

- All commands in the `list` share the same `stdin`, `stdout` and `stderr`.
- The return code is the return code of the last command in `list`.
- All changes the subshell makes to the execution environment are only temporary and are discarded once the subshell exits.

An example

```

1 #!/bin/bash
2 A=3
3 B=6
4 pwd
5 (
6     A=5    #locally change variable
7     echo "Hello_from_subshell: A: $A B: $B"
8     cd ..  #locally change directory
9     pwd
10 )
11 echo "Hello_from_main_shell: A: $A B: $B"
12 pwd

```

6\_functions\_subshells/subshell\_example.sh

```

1 /export/home/abs/abs001/bash-course
2 Hello_from_subshell: A: 5 B: 6
3 /export/home/abs/abs001
4 Hello_from_main_shell: A: 3 B: 6
5 /export/home/abs/abs001/bash-course

```

Subshells are particularly useful whenever one wants to change the environment and knows *per se* that this change is only intended to last for a small part of a script. This way a cleanup cannot be forgotten.

```

1 #!/bin/bash
2
3 #Here want to do some stuff in the PWD
4 echo "The_list_of_files_in_the_PWD:"
5 ls | head -n 4
6 (
7     # do stuff in a different directory
8     cd resources/matrices
9
10    # and using a different IFS
11    IFS=":"
12
13    echo
14    echo "The_list_of_files_in_resources/matrices"
15    ls | head -n4
16
17    echo
18    echo "Some_paths:"
19    for path in $PATH; do
20        echo $path
21    done | head -n4
22 )

```

```

23
24 # and we are back to the original
25 echo
26 for i in word1:word2; do
27     echo $i
28 done

```

6.functions\_subshells/subshell\_cdifs.sh

```

1 The_list_of_files_in_the_PWD:
2 1_intro_Unix
3 2_intro_bash
4 3_simple_scripts
5 4_control_io
6
7 The_list_of_files_in_resources/matrices
8 3a.mtx
9 3b.mtx
10 3.mtx
11 bcsstm01.mtx
12
13 Some_paths:
14 /usr/local/bin
15 /usr/bin
16 /bin
17 /usr/local/games
18
19 word1:word2

```

### 6.1.3 Implicit subshells

Apart from the explicit syntax discussed above, the following commands also start a subshell implicitly

- Pipes: This is done for performance reasons by the `bash`. Forgetting about this is a very common mistake:

```

1 #!/bin/bash
2 C=0 # initialise counter
3 < resources/testfile grep "e" | while read line; do
4     # subshell here!
5     ((C++))
6 done
7 #not in subshell any more:
8 echo "We_found_$C_matches_for_e\"."

```

6.functions\_subshells/subshell\_pipes.sh

```

1 We_found_0_matches_for_e".

```

A workaround for this problem is to run everything that needs to access the variable `C` as a group and cache the output using a command substitution:

```

1 #!/bin/bash
2 COUNT=$(  
3   C=0  
4   while read line; do  
5       ((C++))  
6   done  
7   echo $C  
8 })  
9 echo "We found $COUNT matches for \"e\"."

```

6\_functions\_subshells/subshell\_pipes\_correct.sh

```

1 We found 4 matches for "e".

```

- Command substitutions: Usually less of a problem

```

1 #!/bin/bash
2 A=-1
3 # everything between $( and ) in the next
4 # line is a subshell. The increment is lost.
5 echo $( ((A++)); echo $A )
6 echo $A

```

6\_functions\_subshells/subshell\_commandsubst.sh

```

1 0
2 -1

```

- If command substitutions start a subshell one might wonder how we could extract multiple results calculated in a single command substitution. Unfortunately there is no simple way to do this, since all changes we make to variables inside the `$( \dots )` are lost. We only have *stdout*, which we can cache in another variable in order to pass data back to the main shell. The solution to this problem is to pack the data inside the subshell and to unpack it later, e.g.

```

1 #!/bin/bash
2
3 # some input from the main shell
4 N=15
5
6 RES=$(
7     # do calculations in the subshell
8     SUM=$((N+13))
9     SQUARE=$((N*N))
10
11     # pack the results with a :
12     # i.e. echo them separated by a :
13     echo "$SUM:$SQUARE"
14 )
15
16 # now use cut to unpack them and recover
17 # the individual values
18 SUM=$(echo "$RES" | cut -d: -f1)

```

```

19 SQUARE=$(echo "$RES" | cut -d: -f2)
20
21
22 # echo them:
23 echo "$SUM"
24 echo "$SQUARE"

```

6.functions\_subshells/subshell\_pack.sh

```

1 28
2 225

```

**Exercise 6.1.** This script does not produce the results the author expected. Spot the errors and correct them.

```

1 #!/bin/bash
2
3 # initial note:
4 #   this script is deliberately made cumbersome
5 #   this script is bad style. DO NOT COPY
6
7 # keyword
8 KEYWORD=${1:-0000}
9
10 ERROR=0
11 [ ! -f "bash_course.pdf" ] && (
12     echo "Please run at the top of the bash_course repository ✓
13     ↪" >&2
14     ERROR=1
15 )
16 # change to the resources directory
17 if ! cd resources/; then
18     echo "Could not change to resources directory" >&2
19     echo "Are we in the right directory?"
20     ERROR=1
21 fi
22
23 [ $ERROR -eq 1 ] && (
24     echo "A fatal error occurred"
25     exit 1
26 )
27
28 # list of all matching files
29 MATCHING=
30
31 # add files to list
32 ls matrices/*.mtx gutenber/*.txt | while read line; do
33     if < "$line" grep -q "$KEYWORD"; then
34         MATCHING=$(
35             echo "$MATCHING"
36             echo $line
37         )
38     fi

```



```

39 done
40
41 # count the number of matches:
42 COUNT=$(echo "$MATCHING" | wc -l)
43
44 if [ $COUNT -gt 0 ]; then
45     echo "We found $COUNT matches!"
46     exit 0
47 else
48     echo "No match" >&2
49     exit 1
50 fi

```

6\_functions\_subshells/subshell\_exercise.sh

```
1 We found 1 matches!
```

**Exercise 6.2.** Rewrite your PATH-lookup script from exercise 4.18 on page 55 using the features from this section wherever it is sensible.

## 6.2 bash functions

The best way to structure shell code by far are **bash** functions. Functions are defined<sup>1</sup> like

```
1 name() { list; }      # list executed in the current shell ✓
    ↪environment
```

or

```
1 name() (list)        # list executed in subshell
```

and essentially define an alias to execute list by the name of name. Basic facts:

- Functions work like user-defined commands. We can redirect and/or pipe stuff from/to them. As with scripts or grouped commands, the whole list shares *stdin*, *stdout* and *stderr*.

```

1 #!/bin/bash
2 testfct() {
3     echo blub    #write to stdout
4     read test    #read from stdin
5     read test2   #also read from stdin
6     echo $test >&2 #write to stderr
7     echo $test2  #write to stout
8 }
9
10
11 {
12     echo line1
13     echo line 2
14 } | testfct | grep 2

```

6\_functions\_subshells/fun\_pipe.sh

<sup>1</sup>There are more ways to define functions. See the **bash** manual [2] for the others

```

1 line1
2 line_2

```

- We can pass arguments to functions, which are available by the positional parameters

```

1 #!/bin/bash
2
3 argument_analysis() {
4     echo $1
5     echo $2
6     echo $#
7     echo $#
8 }
9
10 # call function
11 argument_analysis 1 "2_3" 4 5

```

6\_functions\_subshells/fun\_arguments.sh

```

1 1
2 2_3
3 1_2_3_4_5
4 4

```

- Inside a function the special `return` command exists, which allows to exit a function prematurely and provide an exit code to the caller.
- If no `return` is called, the last command in list determines the exit code.

```

1 #!/bin/bash
2
3 return_test() {
4     if [ "$1" == "a" ]; then
5         echo "No_thanks"
6         return 1
7     fi
8
9     echo "Thank_you"
10 }
11
12 other_test() {
13     [ "$1" == "b" ]
14 }
15
16 VAR=b
17 if other_test "$VAR"; then
18     return_test "$VAR"
19     echo $?
20 fi
21
22 return_test "a"
23 echo $?

```

6\_functions\_subshells/fun\_return.sh

```

1 Thank_you
2 0
3 1

```

- All variables of the calling shell are available and may be modified
- Variables inside a function may be defined with the prefix `local`. In this case they are forgotten once the function returns from the `list`. In other words this variable is only available for the function itself and all its children<sup>2</sup>.

```

1 #!/bin/bash
2 # Global variables:
3 VAR1=vvv
4 VAR3=lll
5
6 variable_test() {
7     local FOO=bar
8     echo $VAR1
9     VAR3=$FOO
10 }
11
12 echo "--$VAR1--$FOO--$VAR3--"
13 variable_test
14 echo "--$VAR1--$FOO--$VAR3--"

```

6\_functions\_subshells/fun\_vars.sh

```

1 --vvv----lll--
2 vvv
3 --vvv----bar--

```

⇒ One can think of functions as small scripts within scripts.

---

<sup>2</sup>Functions directly or indirectly called by the function, i.e. called functions, functions called from called functions, ...

Good practice when using functions:

- Give functions a sensible and descriptive name.
- Put a comment right at the top of the function definition, describing:
  - what the function does
  - what the expected argument are
  - what the return code is
- Do not trust the caller: Check similar to a script that the parameters have the expected values
- Do not modify global variables unless you absolutely have to. This greatly improves the readability of your code.
- Use local variables by default inside functions.
- Have functions first, then “global code”
- Try to define functions in an abstract way. This makes is easier to reuse and expand them later.
- It usually is a good idea to have functions only return error codes and print error messages somewhere else depending on the context.

Compare the two code snippets and decide for yourself what is more readable<sup>3</sup>

```

1 #!/bin/bash
2 # a bad example
3
4 if [ "$1" == "-h" -o "$1" == "--help" ];then
5     echo "Script to display basic information in an mtx file"
6     exit 0
7 fi
8
9 foo() {
10     echo $NONZERO
11 }
12
13 DATA=""
14
15 check2() {
16     if [ -z "$DATA" ]; then
17         echo "Can't read file" >&2
18         return 1
19     fi
20     return 0
21 }
22
23 blubb() {
24     echo $ROW
25 }
```

<sup>3</sup>By the way: `6_functions_subshells/fun_bad.sh` contains an error. Good luck finding it.

```

26
27 check1() {
28     if [ ! -r "$1" ]; then
29         echo "Can't read file" >&2
30         return 1
31     fi
32     return 0
33 }
34
35 check1 "$1" || exit 1
36
37 fun1() {
38     DATA=$(< "$1" grep -v "%" | head -n1)
39 }
40
41 fun1 "$1"
42 check2 || exit 1
43
44 reader() {
45     echo $DATA | {
46         read COL ROW NONZERO
47     }
48 }
49
50 reader
51 echo -n "No rows:UUUUU"; blubb
52
53 tester() {
54     echo $COL
55 }
56 echo -n "No cols:UUUUU"; tester
57 echo -n "No nonzero:UU"; foo
58
59 exit 0

```

6.functions\_subshells/fun\_bad.sh

```

1 #!/bin/bash
2 # a good example
3
4 mtr_read_head() {
5     # $1: file name of mtx file
6     # echos the first content line (including the matrix size ↗
7     ↪) to stdout
8     # returns 0 if all is well
9     # returns 1 if an error occurred (file could not be read)
10
11     # check we can read the file
12     [ ! -r "$1" ] && return 1
13
14     # get the data
15     local DATA=$(< "$1" grep -v "%" | head -n1)

```

```

16 # did we get any data?
17 if [ "$DATA" ]; then
18     echo "$DATA"
19     return 0
20 else
21     return 1
22 fi
23 }
24
25 gcut() {
26     # this a more general version of cut
27     # that can be tuned using the IFS
28     #
29     # $1: n -- the field to get from stdin
30     # return 1 on any error
31
32     local n=$1
33     if ((n<1)); then
34         return 1
35     elif ((n==1)); then
36         local FIELD BIN
37
38         # read two fields and return
39         # the first we care about
40         read FIELD BIN
41         echo "$FIELD"
42     else
43         local FIELD REST
44
45         # discard the first field
46         read FIELD REST
47
48         # and call myself
49         echo "$REST" | gcut $((n-1))
50     fi
51 }
52
53 mtx_get_rows() {
54     # get the number of rows in the matrix from an mtx file
55     # echo the result to stdout
56     # return 1 if there is an error
57
58     local DATA
59
60     # read the data and return when error
61     DATA=$(mtr_read_head "$1") || return $?
62     # parse the data -> row is the first field
63     echo "$DATA" | gcut 1
64
65     # implicit return of return code of gcut
66 }
67
68 mtx_get_cols() {
69     # get the number of columns in the matrix file

```

```

70 # return 1 on any error
71
72 local DATA
73 DATA=$(mtr_read_head "$1") || return $?
74 echo "$DATA" | gcut 2 #cols on field 2
75 }
76
77 mtx_get_nonzero() {
78 # get the number of nonzero entries in the matrix file
79 # return 1 on any error
80
81 local DATA
82 DATA=$(mtr_read_head "$1") || return $?
83 echo "$DATA" | gcut 3 #cols on field 2
84 }
85
86 mtx_get_comment() {
87     mtx_fill_cache "$1" && echo "$_MTX_INFO_CACHE_COMMENT"
88 }
89
90 #####
91 # the main script
92
93 if [ "$1" == "-h" -o "$1" == "--help" ];then
94     echo "Script to display basic information in an mtx file"
95     exit 0
96 fi
97
98 if [ ! -r "$1" ]; then
99     echo "Please specify mtx file as first arg." >&2
100    exit 1
101 fi
102
103 echo "No rows:$(mtx_get_rows "$1")"
104 echo "No cols:$(mtx_get_cols "$1")"
105 echo "No nonzero:$(mtx_get_nonzero "$1")"
106
107 exit 0

```

6\_functions\_subshells/fun\_good.sh

**Exercise 6.3.** *optional* Rebuild the `find -type f` command (see exercise 4.17 on page 52) using the features of the `bash` shell. I.e. your script should list the relative path to all files in all subdirectories of the current working directory. Some hints:

- It is a good idea to define a function that deals with the directories recursively
- Use subshells to keep track of the current directory level you are in.
- The `for file in *; do`-loop is your friend here.

**Exercise 6.4. optional** Take another look at your script from the second Project Gutenberg exercise (exercise 5.4 on page 63). Split the script up into sensible functions. A few ideas:

- One function to parse all output from the ex.-4.6-script and prepare a list of the book names and  $\xi$ -numbers on *stdout*
- One function to read this list and print three recommended books to *stdout*
- The main body should just call the ex.-4.6-script and the functions defined above and print the final messages to the user.

**Exercise 6.5.** In this exercise we will try some abstract **bash** programming using functions. First take a look at the following function:

```

1 map() {
2     COMMAND=$1 # read the command
3     shift      # shift $1 away
4
5     # now for all remaining arguments execute
6     # the command with the argument:
7     for val in $@; do
8         $COMMAND $val
9     done
10 }
```

6\_functions\_subshells/map.lib.sh

It is a so-called mapping function that applies a command or a function name to all arguments provided in turn. Copy the code to a fresh file and add the following lines in order to understand **map** more closely:

```

1 map echo "some" "variables_on_the" "commandline"
2
3 cd ~/bash-course #replace by dir where you downloaded ✓
4 ↪the git into
5 map head "resources/testfile" "resources/matrices/3.mtx" ✓
6 ↪"
```

What happens in each case?

Now try to write the following functions:

- A function **add** that expects 2 arguments. It adds them and echos the result.
- A function **multiply** that also expects 2 arguments. It multiplies them and echos the result.
- A function **operation** that reads a global variable **SEL** and depending on its value calls **add** or **multiply**. It should pass all arguments supplied to **operation** further on to either **add** or **multiply**.
- A function **calculate3** that takes a single argument and calls **operation** passing on this single argument and also the number “3” as the second argument to **operation**.



*optional* Write an encapsulating script that

- uses `map` to apply `calculate3` all arguments on the commandline but the first.
- examines the first argument in order to set the variable `SEL` (e.g. the argument `--add3` selects addition, the argument `--multiply3` multiplication)

How much effort does it take to add a third option that allows to subtracts 3 from all input parameters?

### 6.2.1 Overwriting commands

At the stage of execution the `bash` gives preference to user-defined functions over builtin commands or commands from the operating system. As a result care must be taken when naming your functions, since these can “overwrite” commands<sup>4</sup>:

```

1 #!/bin/bash
2
3 test() {
4     echo "Hi_from_the_test_function"
5 }
6
7 VAR="blubber"
8 test -z "$VAR" && echo "VAR_is_zero"

```

6\_functions\_subshells/overwrite\_fail.sh

```

1 Hi_from_the_test_function
2 VAR_is_zero

```

This is of course also true for commands within the function itself, which can lead to very subtle infinite loops:

```

1 #!/bin/bash
2
3 C=0 # count to break at some point
4
5 [() { # overwrite the [ builtin
6
7     # use test to end at some point
8     if test $((C++)) -gt 100; then
9         echo "$C"
10        exit 0
11    fi
12
13    # this gives an infinite loop:
14    if [ $C -gt 100 ] ; then
15        echo "never_printed"
16        exit 1
17    fi

```

<sup>4</sup>*Overwriting* is a concept from object-oriented programming where functions of the same name are called depending on the context of the call

```

18 }
19
20 if [ "$VAR" ]; then
21     echo "VAR_is_not_empty" #never reached
22 fi

```

6\_functions\_subshells/overwrite\_loop.sh

```

1 102

```

In scripts it is best to avoid this feature since it can make code very counterintuitive and hard to understand. For customising your interactive **bash**, however, this can become very handy (see appendix B.1.1 on page 123).

Also note, that the **bash** only remembers the most recently defined body for a function name. So we could alter a function dynamically during a script.

```

1 #!/bin/bash
2
3 printer() { echo "1"; }
4
5 for((I=0;I<10;++I)); do
6     printer
7     printer() { echo "$I"; }
8 done

```

6\_functions\_subshells/overwrite\_mostrecent.sh

```

1 1
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
10 9

```

Again this feature should be used with care.

### 6.3 Cleanup routines

Using subshells it becomes easy to temporarily alter variables and have them “automatically” change back to their original value — no matter how the subshell exited. For some use cases this is not enough, however. Consider for example the following program

```

1 #!/bin/bash
2 TMP=$(mktemp) # create temporary file
3
4 # add some stuff to it
5 echo "data" >> "$TMP"
6

```

```

7 ##
8 # many lines of code
9 ##
10
11 # and now we forgot about the temporary file
12 if [ "$CONDITION" != "true" ]; then
13     exit 0
14 fi
15
16 ##
17 # many more lines of code
18 ##
19
20 #cleanup
21 rm $TMP

```

6\_functions\_subshells/cleanup\_notrap.sh

Especially when programs get very long (and there are many exit conditions) one easily forgets about a proper cleanup in all cases. For such purposes we can define a routine that gets executed whenever the shell exits, e.g.

```

1 #!/bin/bash
2 TMP=$(mktemp) # create temporary file
3
4 # define the cleanup routine
5 cleanup() {
6     echo cleanup called
7     rm $TMP
8 }
9 # make cleanup be called WHENEVER the shell exits
10 trap cleanup EXIT
11
12 # add some stuff to it
13 echo "data" >> "$TMP"
14
15 ##
16 # many lines of code
17 ##
18
19 # and now we forgot about the temporary file
20 if [ "$CONDITION" != "true" ]; then
21     exit 0
22 fi
23
24 ##
25 # many more lines of code
26 ##
27
28 #no need to do explicit cleanup

```

6\_functions\_subshells/cleanup\_trap.sh

```

1 cleanup_called

```

## 6.4 Making script code more reusable

Ideally one wants to write code once and reuse it as much as possible. This way when new features or a better algorithm is implemented, one needs to change the code at only a single place (see ex. 6.5 on page 81). For this purpose the `bash` provides a feature called “sourcing”. Using the syntax

```
1 . otherscript
```

a file `otherscript` can be executed in the environment of the *current* shell. This means that all variables and functions defined in `otherscript` are also available to the shell afterwards:

```
1 testfunction() {
2     echo "Hey_I_exist"
3 }
4 VAR=foo
```

6\_functions\_subshells/sourcing.lib.sh

```
1 #!/bin/bash
2
3 PATH="$PATH:6_functions_subshells"
4 . sourcing.lib.sh #lookup performed in PATH
5
6 echo $VAR
7 testfunction
```

6\_functions\_subshells/sourcing.script.sh

```
1 foo
2 Hey_I_exist
```

Note: In order to find `otherscript` the `bash` honours the environment variable `PATH`. As the example suggests this way libraries defining common or important functionality may be stored in a central directory and used from many other scripts located in very different places.

There exists a dirty trick to make each script become sourceable by default. It relies on the fact that the `return` statement is not allowed in scripts, which are executed normally, but is a well-allowed command if this file is sourced instead. Therefore one can realise a break between function definitions and “global code” that is only considered when a script is actually executed:

```
1 #!/bin/bash
2
3 mtr_read_head() {
4     # $1: file name of mtx file
5     # echos the first content line (including the matrix size ↗
6     ↪) to stdout
7     # returns 0 if all is well
8     # returns 1 if an error occurred (file could not be read)
9
10    # check we can read the file
11    [ ! -r "$1" ] && return 1
12
13    # get the data
```

```

13  local DATA=$(< "$1" grep -v "%" | head -n1)
14
15  # did we get any data?
16  if [ "$DATA" ]; then
17      echo "$DATA"
18      return 0
19  else
20      return 1
21  fi
22 }
23
24 gcut() {
25     # this a more general version of cut
26     # that can be tuned using the IFS
27     #
28     # $1: n -- the field to get from stdin
29     # return 1 on any error
30
31     local n=$1
32     if ((n<1)); then
33         return 1
34     elif ((n==1)); then
35         local FIELD BIN
36
37         # read two fields and return
38         # the first we care about
39         read FIELD BIN
40         echo "$FIELD"
41     else
42         local FIELD REST
43
44         # discard the first field
45         read FIELD REST
46
47         # and call myself
48         echo "$REST" | gcut $((n-1))
49     fi
50 }
51
52 mtx_get_rows() {
53     # get the number of rows in the matrix from an mtx file
54     # echo the result to stdout
55     # return 1 if there is an error
56
57     local DATA
58
59     # read the data and return when error
60     DATA=$(mtr_read_head "$1") #|| return $?
61     # parse the data -> row is the first field
62     echo "$DATA" | gcut 1
63
64     # implicit return of return code of gcut
65 }
66

```

```

67 mtx_get_cols() {
68     # get the number of columns in the matrix file
69     # return 1 on any error
70
71     local DATA
72     DATA=$(mtr_read_head "$1") || return $?
73     echo "$DATA" | gcut 2 #cols on field 2
74 }
75
76 mtx_get_nonzero() {
77     # get the number of nonzero entries in the matrix file
78     # return 1 on any error
79
80     local DATA
81     DATA=$(mtr_read_head "$1") || return $?
82     echo "$DATA" | gcut 3 #cols on field 2
83 }
84
85 mtx_get_comment() {
86     mtx_fill_cache "$1" && echo "$_MTX_INFO_CACHE_COMMENT"
87 }
88
89 #if we have been sourced this exits execution here:
90 # so by sourcing we can use gcut, mtx_get_rows, ...
91 return 0 &> /dev/null
92
93 #####
94
95 if [ "$1" == "-h" -o "$1" == "--help" ];then
96     echo "Script to display basic information in an mtx file"
97     exit 0
98 fi
99
100 if [ ! -r "$1" ]; then
101     echo "Please specify mtx file as first arg." >&2
102     exit 1
103 fi
104
105 echo "No rows:$(mtx_get_rows "$1")"
106 echo "No cols:$(mtx_get_cols "$1")"
107 echo "No nonzero:$(mtx_get_nonzero "$1")"
108
109 exit 0

```

6\_functions\_subshells/source\_sourcability.sh

**Exercise 6.6.** Make your script from exercise 6.5 on page 81 sourceable and amend the following script in order to get the functionality described in the comments:

```
1 #!/bin/bash
2
3 # do something in order to get the functions
4 # add and multiply from the exercise we had before
5
6 # add 4 and 5 and print result to stdout:
7 add 4 5
8
9 # multiply 6 and 7 and print result to stdout:
10 multiply 6 7
```

6\_functions\_subshells/source\_exercise.sh

## Chapter 7

# Regular expressions

In the previous chapters we have introduced the most important features of the **bash** shell<sup>1</sup>. We will now discuss regular expressions, a syntax that is used by many Unix tools in order to search for or describe textual data.

### 7.1 Regular expression syntax

#### 7.1.1 Matching regular expressions in plain bash

We will introduce regular expressions in a second, but beforehand we need a tool with which we can try them out with. The **bash** already provides us with a syntax which understands regular expressions or *regexes*:

```
1 [[ string =~ regex ]]
```

- This command returns with exit code 0 when there exists a substring in string which can be described by the regular expression regex. Else it returns 1.
- If such a substring exists one calls string a *match* for regex and says that regex *matches* string.

Actually the `[[` command can do a lot more things than just matching regular expressions, which we will not discuss here. Just note that it is an extended version of `[`, so in fact everything you know for `[` can also be done using `[[ ... ]]` in exactly the same syntax.

#### 7.1.2 Regular expression operators

It is best to think of regular expressions as a “search” string where some characters have a special meaning. All non-special characters just stand for themselves, e.g. the regex “a” just matches the string “a”<sup>2</sup>.

Without further ado a non-exhaustive list of *regular expression operators*<sup>3</sup>:

---

<sup>1</sup>A list of things we left out can be found in appendix B.4 on page 124

<sup>2</sup>This is why for **grep** — which in fact also uses substrings by default — we could just **grep** for a word not even knowing anything about regexes

<sup>3</sup>More can be found e.g. in the **awk** manual [3]



`\` The escape character: Disables the special meaning of a character that follows

`^` matches the beginning of a string, ie. “`^word`” matches “`wordblub`” but not “`blubword`”. Note that `^` does not match the beginning of a line:

```
1 [[ $(echo -e "test\nword") =~ ^test ]]; echo $? ✓
   ↪ #0=true
2 [[ $(echo -e "word\ntest") =~ ^test ]]; echo $? ✓
   ↪ #1=false
```

7\_regular\_expressions/regex\_anchor.sh

`$` matches the end of a string in a similar way

```
1 [[ $(echo -e "word\ntest") =~ test$ ]]; echo $? ✓
   ↪ #0=true
2 [[ $(echo -e "test\nword") =~ test$ ]]; echo $? ✓
   ↪ #1=false
```

7\_regular\_expressions/regex\_anchorend.sh

`.` matches any single character, including `<newline>`, e.g. `P.P` matches `PAP` or `PLP` but not `PLLP`

`[...]` *bracket expansion*: Matches one of the characters enclosed in square brackets.

```
1 [[ "o" =~ ^[oale]$ ]]; echo $? #true
2 [[ "a" =~ ^[oale]$ ]]; echo $? #true
3 [[ "oo" =~ ^[oale]$ ]]; echo $? #false
4 [[ "\$" =~ ^[$]$ ]]; echo $? #true
```

7\_regular\_expressions/regex\_bracket.sh

Note: Inside *bracket expansion* only the characters `]`, `-` and `^` are *not* interpreted as literals.

`[^...]` *complemented bracket expansion*: Matches all characters *except* the ones in square brackets

```
1 [[ "o" =~ [^eulr] ]]; echo $? #true
2 [[ "e" =~ [^eulr] ]]; echo $? #false
3
4 #ATTENTION: this is not a cbe
5 [[ "a" =~ [o^ale] ]]; echo $?
```

7\_regular\_expressions/regex\_compbracket.sh

`|` *alternation operator* Specifies alternatives: Either the regex to the right or the one to the left has to match. Note: Alternation applies to the largest possible regexes on either side

```
1 #gives true, since ^wo
2 [[ "word" =~ ^wo|rrd$ ]]; echo $?
```

7\_regular\_expressions/regex\_alternation.sh

(...) Grouping regular expressions, often used in combination with `|`, to make the alternation clear, e.g.

```
1 [[ "word" =~ ^(wo|rrd)$ ]]; echo $? #1=false
7_regular_expressions/regex_grouping.sh
```

**\*** The *preceding* regular expression should be repeated as many times as necessary to find a match, e.g. “`ico*`” matches “`ic`”, “`ico`” or “`icoooo`”, but not “`icco`”. The “`*`” applies to the *smallest* possible expression only.

```
1 [[ "wo_(rd)" =~ wo* \(\) ]]; echo $? #true
2 [[ "woo_(rd)" =~ wo* \(\) ]]; echo $? #true
3 [[ "oo_(rd)" =~ wo* \(\) ]]; echo $? #false
4 [[ "oo_(rd)" =~ (wo)* \(\) ]]; echo $? #true
5 [[ "wowo_(rd)" =~ (wo)* \(\) ]]; echo $? #true
7_regular_expressions/regex_star.sh
```

**+** Similar to “`*`”: The preceding expression must occur at least once

```
1 [[ "woo_(rd)" =~ wo+ \(\) ]]; echo $? #true
2 [[ "oo_(rd)" =~ (wo)+ \(\) ]]; echo $? #false
3 [[ "wo_(rd)" =~ (wo)+ \(\) ]]; echo $? #true
7_regular_expressions/regex_plus.sh
```

**?** Similar to “`*`”: The preceding expression must be matched once or not at all. E.g. “`ca?r`” matches “`car`” or “`cr`”, but nothing else.

There are a few things to note

- Programs will try to match as much as possible.
- Regexes are case-sensitive
- Unless `^` or `$` are specified, the matched substring may start and end anywhere and a single matching substring is enough to fulfil the condition imposed by a regular expression

### 7.1.3 A shorthand syntax for bracket expansions

Both bracket expansion and complemented bracket expansion allow for a shorthand syntax, which can be used for *ranges* of characters or ranges of numbers, e.g

short form	equivalent long form
[a-e]	[abcde]
[aA-F]	[aABCDEF]
[^a-z4-9A-G]	[^abcdefghijklmnopqrstuvwxyz456789ABCDEFGHIJKLMNOPQRSTUVWXYZ]

**Exercise 7.1.** Consider these strings

"ab"	"67"	"7b7"
"g"	"67777"	"o7x7g7"
"77777"	"7777"	" (empty)

For each of the following regexes, decide which of the above strings are matched:

- ..
- ^..\$
- [a-e]
- ^.7\*\$
- ^(.7)\*\$

### 7.1.4 POSIX character classes

There are also some special, named bracket expansions, called POSIX character classes. For example

short form	equivalent long form	description
[ :alnum: ]	a-zA-Z0-9	alphanumeric chars
[ :alpha: ]	A-Za-z	alphabetic chars
[ :blank: ]	␣\t	space and tab
[ :digit: ]	0-9	digits
[ :print: ]		printable characters
[ :punct: ]		punctuation chars
[ :space: ]	␣\t\r\n\v\f	space characters
[ :upper: ]	A-Z	uppercase chars
[ :xdigit: ]	a-fA-F0-9	hexadecimal digits

Note that POSIX character classes can only be used within bracket expansions, e.g.

```

1 if [[ $1 =~ ^[[:space:]]*[0[:alpha:]]+ ]]; then
2     # $1 starts arbitrarily many spaces
3     # following by at least one 0 or letter
4     echo Match
5     exit 0
6 fi
7 echo "No␣match"
8 exit 1

```

7.regular\_expressions/regex\_posixclass.sh

### 7.1.5 Getting help with regexes

Writing regular expressions takes certainly a little practice, but is extremely powerful once mastered.

- <https://www.debuggex.com> is extremely helpful in analysing and understanding regular expressions. The website graphically analyses a regex and tells you why a string does/does not match.
- Practice is everything: See <http://regexcrossword.com/> or try the Android app *ReGeX*.

**Exercise 7.2.** Fill the following regex crossword. The strings you fill in have to match both the pattern in their row as well as the pattern in their column.

	a?[3[:space:]]+b?	b[^21eaf0]
[a-f][0-3]		
[:xdigit:]]b+		

**Exercise 7.3.** Give regular expressions that satisfy the following

	matches	does not match	chars
a)	abbbc, abbc, abc, ac	aba	4
b)	abbbc, abbc, abc	bac, ab	4
c)	ac, abashc, a123c	cbluba, aefg	5
d)	□□qome, □qol, qde	eqo, efeg	4
e)	arrp, whee	bla, kee	4

Note: The art of writing regular expressions is to use the smallest number of characters possible to achieve your goal. The number in the last column gives the number of characters necessary to achieve *a* possible solution.

## 7.2 Using regexes with grep

**grep** uses regular expressions by default, so instead of providing it with a word to search for, we can equally supply it with a regular expression as well. Instead of filtering those lines of input data which contain the word provided, the regular expression will be matched to the *whole line*, i.e. **grep** will only show those lines which are matched by the regex.

Care has to be taken to properly quote or escape those characters in the regex which are special characters to the shell. Otherwise the shell tries to interpret them by itself and they are thus not actually passed on to **grep** at all. In most cases surrounding the search pattern by single quotes deals with this issue well.

```
1 # find lines containing foo!bar:
2 < file grep 'foo!bar'
```

Exceptions to this rule of thumb are

- A literal “!” is needed in the search pattern.
- Building the search pattern requires the expansion of shell variables.

In the latter cases one should use double quotes instead and escape all necessary things manually. Note that this can lead to constructs like

```

1 # find the string \'
2 echo "tet\'ter" | grep "\\\'"

```

where a lot of backslashes are needed.

Especially the `-o`-flag is extremely useful when used together with regular expressions. Its purpose is to have `grep` print only the part of the line, which actually matches the regex. E.g. running

```

1 #!/bin/bash
2
3 echo "Plain_grep_gives:"
4 < resources/testfile grep "[a-f]$"
5
6 echo "grep_o_gives:"
7 < resources/testfile grep -o "[a-f]$"

```

7\_regular\_expressions/grep\_only\_matching.sh

gives

```

1 Plain grep gives:
2 some
3 data
4 some
5 date
6 grep -o gives:
7 me
8 ta
9 me
10 te

```

There are quite a few cases where plainly using `grep` with a regular expression does not lead to the expected result. Examples are when the regex contains the `( ... )`, `|`, `?` or `+` operators. If this happens (or when in doubt) one should pass the additional argument `-E` to `grep`.

The `-E` flag is sometimes necessary since `grep` by default only expects a so-called *basic regular expression* or BRE from the user, whereas the syntax explained in this chapter gives so-called *extended regular expressions* or EREs<sup>4</sup>. As the name suggests EREs are more powerful and can be considered a superset of BREs<sup>5</sup>. Nevertheless it is a good idea to just use plain `grep` wherever this is sufficient since matching strings using EREs is a more demanding process.

---

<sup>4</sup>To make matters worse there are actually even more kinds of regular expressions. The scripting language `perl` has its own dialect, so-called `perl`-compatible regular expressions or PCREs. Often which operators are understood as BRE or ERE — or even understood at all — depends on the program or the implementation (e.g. GNU `grep` is different than traditional Unix `grep` ...)

<sup>5</sup>This is not fully correct, see `grep` manpage for details.

**Exercise 7.4.** This exercise tries to show you how much more powerful `grep` becomes when used with regular expressions:

- Design a regular expression to match a single digit. In other words if the string contains the number “456”, the regex should match “4”, “5” and “6” *separately* and not “456” as a whole.
- Use `grep -o` together with this expression on the file `resources/digitfile`. You should get a list of single digits.
- Look at the file. What does this list have to do with the input?
- Now pipe this result in some appropriate Unix tools in order to find out how many times each digit is contained in the file. The output should be some sort of a table telling you that there are e.g. 2 fours, 3 twos, ...

*optional* Now we try to extract a little more structured information from the file `resources/matrices/bcsstm01.mtx`. More information about the `mtx`-format can be found in appendix C.1 on page 125 if necessary.

- First use `grep -o -E` to verify that the regular expression `-?[0-9]\.[0-9]*e[+-][0-9][0-9]` extracts the 3rd values column from `resources/matrices/bcsstm01.mtx`. Since the regex starts with a `-` itself you will need to call `grep` like this

```
1 grep -o -E -e -?[0-9]\.[0-9]*e[+-][0-9][0-9]
```

- Use this expression to find the largest matrix value of `resources/matrices/bcsstm01.mtx`.

## 7.3 Using regexes with sed

`sed` — the stream editor — is a program program to filter or change textual data. We will not cover the full features of `sed`, but merely introduce a few basic commands which allow to add, delete or change lines on *stdin*. The invocation of `sed` is almost exactly like `grep`. Either one filters a stream:

```
1 echo "data_stream" | sed 'sed_commands'
```

or reads a file, filters it and prints it to *stdout*

```
1 sed 'sed_commands' file
```

Again, if a literal “`'`” or e.g. parameter expansions are needed in `sed_commands`, we are better off using double quotes instead. Be warned, that double quotes can lead to an accumulation of escapes for both `sed` as well as the shell:

```
1 # compare
2 echo '\$a' | sed "s/\\\\\\\$a/bbb/g"
3
4 # with the single-quote example
5 echo '\$a' | sed 's/\\\\\$a/bbb/g'
```

7\_regular\_expressions/sed\_double\_quotes.sh

Overview of basic sed commands<sup>6</sup>:

cmd; cmd2      Run two **sed** commands on the same stream sequentially: First cmd1 is executed and on the resulting line cmd2. Can also be achieved by having the two commands separated by a line break.

/regex/atext      Add a new line containing text *after* each line which is matched by regex.

/regex/itext      Similar to above, but add the line with text *before* the matched lines.

```

1 #!/bin/bash
2
3 {
4     echo blub
5     echo blbl
6 } | sed '/bl/alaber '
7
8 echo -----
9
10 {
11     echo blub
12     echo blbl
13 } | sed '/bl/ilaber '
```

7.regular\_expressions/sed\_insertion.sh

```

1 blub
2 laber
3 blbl
4 laber
5 -----
6 laber
7 blub
8 laber
9 blbl
```

/regex/d      Delete matching lines.

```

1 #!/bin/bash
2 {
3     echo line1
4     echo line2
5     echo line3
6 } | sed '/2$/d '
```

7.regular\_expressions/sed\_delete.sh

```

1 line1
2 line3
```

<sup>6</sup>see e.g. the **sed** manual [4] for more details.

`s/regex/text/` Substitute the first match of `regex` in each line by `text`. We can use the special character `&` in `text` to refer back to the precise part of the current line that was matched by `regex` (so the thing `grep -o` would extract). Note that `text` may contain special escape sequences like “`\n`” or “`\t`”.

`s/regex/text/g` Works like the above command except that it substitutes all matches of `regex` in each line by `text`.

```

1 #!/bin/bash
2
3 generator() {
4     echo "line1"
5     echo "line2"
6     echo "LiNE3"
7     echo
8 }
9
10 generator | sed 's/in/blablabla/'
11 echo "-----"
12 generator | sed 's/i.*[1-3]/...&.../'
13 echo "-----"
14
15 # a very common sequence to normalise input
16 generator | sed '
17     s/[[[:space:]]][[:space:]]*/ /g
18     s/^[[:space:]]//
19     s/[[[:space:]]]$//
20     /^$/d
21 '

```

7\_regular\_expressions/sed\_substitute.sh

```

1 blablablae1
2 line2
3 LiNE3
4
5 -----
6 l...ine1...
7 line2...
8 L...iNE3...
9
10 -----
11 line1
12 line2
13 LiNE3

```

Similar to `grep` it may be necessary to with to extended regular expressions for some things to work. For `sed` this is done by specifying the argument `-r` before passing the `sed` commands.



### 7.3.1 Alternative matching syntax

Sometimes it is desirable to use the `/` character inside a regular expression for a `sed` command as well. E.g. consider replacing specific parts of an absolute path by others. For such cases a more general matching syntax exists:

- In front of a command, `/regex/` can also be expressed as `\c regex c`, where `c` is an arbitrary character.
- For the command `s`: `s c regex c text c` is equivalent to `s/regex/text/`.

```

1 #!/bin/bash
2 VAR="/some"
3 echo "/some/crazy/some/path" | sed "s#$VAR#/m0Re#g"
4 echo "--"
5 echo "/some/crazy/path" | sed "\#crazy#d"
6 echo "--"

```

7.regular\_expressions/sed.altmatch.sh

```

1 /m0Re/crazy/m0Re/path
2 --
3 --

```

**Exercise 7.5.** Consider the first 48 lines of the file `resources/chem_output/qchem.out`.

- First use `head` to only generate a derived file containing just the first 48 lines

Write a `bash` one-liner using `sed` and `grep` that generates a sorted list of the surnames of all *Q-Chem* authors:

- Exclude all lines containing the word `Q-Chem`.
- Remove all initials and bothering “.” or “-” symbols (Do not remove the “-” on compound surnames!)
- Replace all `,` by `\n`, the escape sequence for a line break.
- Do cleanup: Remove unnecessary leading or trailing spaces as well as empty lines
- Pipe the result to `sort`

*optional* This whole exercise can also be done without using `grep`.

## Chapter 8

# A concise introduction to awk programming

In this chapter we will take a brief look at the `awk` programming language designed by Alfred Aho, Peter Weinberger, and Brian Kernighan in order to process text files. Everything we have done in the previous chapters using `grep`, `sed` or any of the other Unix tools can be done in `awk` as well and much much more .... In fact often it only takes a few lines of `awk` to re-code the functionality of one of the aforementioned programs.

### 8.1 Structure of an awk program

All input given to an `awk` program is automatically split up into larger chunks called *records*. Each record is subsequently split up even further into *fields*. By default records are just the individual lines of the input data and fields are the words on each line. In other words records are separated by `<newline>` and fields by any character from `[:space:]`.

`awk` programs are a list of rules given in the following structure

```
1  condition { action }
2  condition { action }
3  ...
```

During execution `awk` goes from record to record and tries to satisfy each condition for it. If the record satisfies the condition the `action` code corresponding to the fulfilled condition is executed.

Both the condition as well as the action block `{ action }` may be missing from an `awk` rule. In the former case the `action` is executed for each input record. In the latter case the whole record is just printed to `stdout` without any change made to it.

Similar to the shell the `#` starts a comment in `awk` programs and `<newline>` and `“;”` may both be used interchangeably. Note that each rule line has to be ended with either `<newline>` or `“;”`.

## 8.2 Running awk programs

There multiple ways to run `awk` programs and provide them with input data. For example we could place all `awk` source code into a file called `name` and then use it like

```
1 awk -f name
```

to parse data from *stdin*. For our use case, where `awk` will just be a helper language to perform small tasks in surrounding `bash` scripts, it is more convenient to use `awk` just inline:

```
1 awk '
2   ...
3   awk_source
4   ...
5 '
```

Note, that once again we could use double quotes here and escape whatever is necessary by hand. As it turns out `awk` has a few very handy features, however, for passing data between the calling script and the inner `awk` program such that we get away with single quotes in almost all cases.

**Example 8.1.** Just to give you an example for what we discussed in this section, this is a shell script which pipes some input to an inline `awk` program, which uses it to print some nice messages<sup>1</sup>. For the printing to *stdout* we make use of the `awk` action command `print` (see 8.8 on page 114 below for details), which works very similar to `echo` in the shell.

```
1 #!/bin/bash
2 {
3   echo "awk_input"
4 } | awk '
5   # missing condition => always done
6   { print "Hi_user.This_is_what_you_gave_me:" }
7
8   # condition which is true and no action
9   # => default print action
10  1 == 1
11
12  # another message which is always printed
13  { print "Thank_you" }
14 '
```

8.awk/basic\_example.sh

```
1 Hi_user.This_is_what_you_gave_me:
2 awk_input
3 Thank_you
```

We observe — as stated in the previous section — that rules without a condition are always executed, and that rules without any action block trigger the default action: Printing the whole record as it is to *stdout*.

<sup>1</sup>I will use syntax highlighting adapted for `awk` code for all example code in this chapter.

### 8.3 awk programs have an implicit loop

As we said in section 8.1 on page 99, all rules of an `awk` program are executed *for each record* of the input data. Usually a record is equal to a line, such that we can consider the whole `awk` program to be enwrapped in an implicit loop over all lines of the input.

Consider the following examples.

```

1 #!/bin/bash
2
3 # function generating the output
4 output() {
5     echo "line_1"
6 }
7
8 echo "Program1:"
9 # a small awk program which just prints the output
10 # line-by-line as it is
11 # we use a condition which is always true and the
12 # default action here (implicit print of the whole
13 # record, i.e. line)
14 output | awk '1==1'
15
16 echo
17 echo "Program2:"
18 # a program with two rules:
19 # one which does the default printing
20 # and a second one which prints an extra line
21 # unconditionally
22 output | awk '
23     1==1 #default print action
24     { print "some_stuff" }
25 '
```

8\_awk/each\_line\_example.sh

Here only a single line of input is specified and hence all rules of the two `awk` programs are run only once: For exactly the single line of input. We get the output

```

1 Program1:
2 line_1
3
4 Program2:
5 line_1
6 some_stuff
```

We note that programs that for programs, which contain multiple rules (like Program2), it may well happen that more than one action gets executed. Here for Program2 both the default action to print the line/record as well as the extra action to print “`extra stuff`” are executed. This is of course since both actions have conditions which are either true or not present and hence implicitly true.

Now let us try the same thing but pass two or three lines of input

```

1 #!/bin/bash
2
3 # function generating the output
4 output() {
5     echo "line_1"
6     echo "line_2"
7 }
8
9 echo "Program1:"
10 output | awk '1==1'
11
12 echo
13 echo "Program2:"
14 output | awk '
15     1==1 #default print action
16     { print "some_stuff" }
17 '

```

8\_awk/each\_line\_example2.sh

```

1 Program1:
2 line_1
3 line_2
4
5 Program2:
6 line_1
7 some_stuff
8 line_2
9 some_stuff

```

and

```

1 #!/bin/bash
2
3 # function generating the output
4 output() {
5     echo "line_1"
6     echo "line_2"
7     echo "line_3"
8 }
9
10 echo "Program1:"
11 output | awk '1==1'
12
13 echo
14 echo "Program2:"
15 output | awk '
16     1==1 #default print action
17     { print "some_stuff" }
18 '

```

8\_awk/each\_line\_example3.sh

```

1 Program1:
2 line_1
3 line_2
4 line_3
5
6 Program2:
7 line_1
8 some_stuff
9 line_2
10 some_stuff
11 line_3
12 some_stuff

```

In these two examples the implicit loop over all records of input shows up. The source code of the **awk** programs has not changed, still we get different output:

- Program1 prints each record/line of input as is, since the default action is executed *for each record* of the input.
- Program2 prints first each record of the input, but then the second rule is also executed for each record as well since the conditions for both rules are missing or true. So overall we get two lines of output for each line of input: First the record itself, then the extra output “extra stuff” from the second rule.

This behaviour is surely a little strange and counterintuitive for people who have experience with other programming languages: The **awk** code is not just executed once, from top to bottom, but in fact *N* times if there are *N* records in the input.

## 8.4 awk statements and line breaks

Not only individual rules but also individual actions within an action block need to be separately by a line break or equivalently a “;”<sup>2</sup>. Other line breaks are (usually) ignored. This means that e.g.<sup>3</sup>

```

1 # the echo is just here to make awk do anything -> see ↙
   ↪ footnote
2 echo | awk '
3 {
4     print "some_message"
5     print "other_message"
6 }
7 {
8     print "third_message"
9 }
10 '

```

<sup>2</sup>This is not entirely correct, see section 1.6 of the **gawk** manual [3] for details

<sup>3</sup>We already said that the **awk** rules are executed *N* times if there are *N* records in the input. This means that they are not touched at all if there is no input. So in many examples in this chapter we will have a leading **echo |** in front of the inline **awk** code, just to have the code execute *once at all*.

and

```
1 echo | awk '{ print "some_message"; print "other_message" }'
2 { print "third_message" }'
```

and

```
1 echo | awk '{ print "some_message"; print "other_message" ↵
↵}; { print "third_message" }'
```

are all equivalent.

## 8.5 Strings in awk

Strings in `awk` all have to be enclosed by double quotes, e.g.<sup>4</sup>

```
1 # inside awk action block -> see footnote
2 print "This_is_a_valid_string"
```

Multiple strings may be concatenated, just by leaving white space between them

```
1 #!/bin/bash
2 echo | awk '{ print "string1" " " "string2" }'
```

8.awk/vars\_stringconcat.sh

```
1 string1_string2
```

`awk` per default honours special sequences like “\t”(Tab) and “\n”(Newline) if used within strings:

```
1 #!/bin/bash
2 echo | awk '
3 { print "test\ttest2\ntest3" }
4 '
```

8.awk/vars\_stringspecial.sh

```
1 test__test2
2 test3
```

## 8.6 Variables and arithmetic in awk

Variables and arithmetic in `awk` are both very similar to the respective constructs in `bash`. A few notes and examples:

- Variables are assigned using a single equals “=”. Note that there can be space between the name and the value.

```
1 var="value"
2 # or
3 var = "value"
```

<sup>4</sup>For some examples in this chapter the enclosing script is left out for clarity. They will just contain plain `awk` code, which could be written inside an `awk` action block. You will recognise these examples by the fact that they don’t start with a shebang.

- Such a statement counts as an action, so we need multiple of these to be separated by a line break or “;”:

```
1 varone="1"; vartwo="2"
```

- In order to use the value of a variable no \$ is required:

```
1 print var          # => will print "value"
```

- awk is aware of floating point numbers and can deal with them properly

```
1 #!/bin/bash
2 echo | awk '{
3     var="4.5"
4     var2=2.4
5     print var "+" var2 "=" var+var2
6 }'
```

8\_awk/vars\_fpaware.sh

```
1 4.5+2.4=6.9
```

- Undefined variables are 0 or the empty string (like in bash)
- Variables are converted between strings and numbers automatically. Strings that cannot be interpreted as a number are considered to be 0.

```
1 #!/bin/bash
2 echo | awk '{
3     floatvar=3.2
4     stringvar="abra"      #cannot be converted to number
5     floatstring="1e-2"    #can be converted to number
6
7     # calculation
8     res1 = floatvar+floatstring
9     res2 = floatvar + stringvar
10
11     print res1 "_" res2
12 }'
```

8\_awk/vars\_fpconvert.sh

```
1 3.21_3.2
```

- All variables are global and can be accessed and modified from all action blocks (or condition statements as we will see later)

```
1 #!/bin/bash
2 echo | awk '
3 { N=4; A="blub" }
4 { print N }
5 { print "String_" A "has_the_length_" length(A) }
6 '
```

8\_awk/vars\_global.sh



```

1 4
2 String blub has the length 4

```

- Arithmetic and comparison operators follow very similar conventions as discussed in the `bash` arithmetic expansion section 5.1 on page 57. This includes the C-like convention of 0 being “false” and 1 being “true”:

```

1 #!/bin/bash
2 echo | awk '{
3     v=3
4     u=4
5
6     print v "-" u "=" v-u
7
8     v+=2
9     u*=0.5
10
11    print v "%" u "=" v%u
12
13
14    # exponentiation is ^
15    print v "^" u "=" v^u
16
17    # need to enforce that comparison operators are
18    # executed before concatenation of the resulting
19    # strings. Not quite sure why.
20    print v "==" u ":" (v==u)
21    print v "!=" u ":" (v!=u)
22    print v "!=" u "||" v "==" u ":" (v!=u || v==u)
23    print v "!=" u "&&" v "==" u ":" (v!=u && v==u)
24 }'

```

8\_awk/vars\_arithlogic.sh

```

1 3-4=-1
2 5%2=1
3 5^2=25
4 0
5 1
6 1
7 0

```

### 8.6.1 Some special variables

Some variables in `awk` have special meaning:

- `$0` contains the content of the current record (i.e. usually the current line). Note that the `$` is part of the name of the variable.
- `$1`, `$2`, ... Variables holding the fields of the current record. `$1` refers to the first field, `$2` to the second and so on. There is no limit on the number of fields, i.e. `$125` refers to the 125th field. If a field does not exist, the variable contains an empty string. Note that these variables may be changed as well!

```

1 #!/bin/bash
2 echo -e "some_7_words\tfor_awk_to_process" | awk '
3 {
4     print "arithmetic:" 2*$2
5     print $4 "_" $1
6 }
7
8 {
9     print "You_gave_me:_" $0
10 }
11 '
```

8\_awk/vars\_fields.sh

```

1 arithmetic: 14
2 for_some
3 You_gave_me: some_7_words_for_awk_to_process
```

This lookup also works indirectly:

```

1 #!/bin/bash
2 echo -e "some_words_for\tawk_to_process" | awk '
3 {
4     v=5
5     print $v
6 }'
```

8\_awk/vars\_fields\_indirect.sh

```

1 to
```

NF contains the number of fields in the current record. So the last field in a record can always be examined using \$NF

```

1 #!/bin/bash
2 echo "some words for awk to process" | awk '
3 {
4     print "There are " NF " fields and the last
      ↪ is " $NF
5 }'
```

8\_awk/vars\_fields\_nf.sh

```

1 There are 6 fields and the last is process
```

FS *field separator*: regular expression giving the characters where the record is split into fields. It can become extremely handy to manipulate this variable. For examples see section 8.9 on page 116.

RS *record separator*: Similar thing to FS: Whenever a match against this regex occurs a new record is started. In practice it is hardly ever needed to modify this.<sup>5</sup>

## 8.6.2 Variables in the awk code vs. variables in the shell script

The inline `awk` code, which we write between the “`'`”, is entirely independent of the surrounding shell script. This implies that all variables which are defined on the shell are *not* available to `awk` and that changes made to the environment within the `awk` program are not known the surrounding shell script either. Consider the example:

```

1 #!/bin/bash
2
3 # define a shell variable:
4 A=laborer
5
6 echo | awk '
7     # define an awk variable and print it:
8     { N=4; print N }
9
10    # print something using the non-present shell variable A:
11    { print "We have no clue about string A: " A " " }
12 '
13
14 # show that the shell knows A, but has no clue about N:
15 echo --$A--$N--
```

8\_awk/awk\_vs\_shell\_vars.sh

```

1 4
2 We have no clue about string A: " "
3 --laborer----
```

<sup>5</sup>Be aware that some `awk` implementations like `mawk` furthermore have no support for changing RS.

So the question arises how we might be able to access computations of the `awk` program from the shell later on. The answer is exactly the same as in section 6.1.3 on page 71, where we wanted to extract multiple results from a single command substitution: We need to pack the results together in the `awk` program and unpack them later in the shell script. For example:

```

1 #!/bin/bash
2
3 # some data we have available on the shell
4 VAR="3.4"
5 OTHER="6.7"
6
7 # do calculation in awk and return packed data
8 RES=$(echo "$VAR_$OTHER" | awk '{
9     sum=$1 + $2
10    product=$1*$2
11    print sum "+" product
12 }')
13
14 # unpack the data on the shell again:
15 SUM=$(echo "$RES" | cut -f1 -d+)
16 PRODUCT=$(echo "$RES" | cut -f2 -d+)
17
18 # use it in an echo
19 echo "The_sum_is:$SUM"
20 echo "The_product_is:$PRODUCT"

```

8\_awk/awk\_vs\_shell.getdata.sh

```

1 The_sum_is:10.1
2 The_product_is:22.78

```

**Exercise 8.2.** Write a script which uses `awk` in order to process some data, which is available to the script on *stdin*:

- Print the second and third column as well as the sum of both for each line of input data. Assume that the columns are separated by one or more characters from the `[:space:]` class.
- You will only need a single line of `awk`.

Try to execute your script, passing it data from `resources/matrices/3.mtx` or `resources/matrices/lund_b.mtx`. Compare the results on the screen with the data in these files. Does your script deal with the multiple column separator characters in the file `resources/matrices/lund_b.mtx` properly?

### 8.6.3 Setting awk variables from the shell

`awk` has a commandline flag `-v` which allows to set variables before the actual inline `awk` program code is touched. A common paradigm is:

```
1  awk -v "name=value" ' awk_source '
```

This is very useful in order to transfer `bash` variables to the `awk` program, e.g.

```
1  #!/bin/bash
2
3  VAR="abc"
4  NUMBER="5.4"
5  OTHER="3"
6
7  # ...
8
9  echo "data_1_2_3" | awk -v "var=$VAR" -v "num=$NUMBER" -v "
↪other=$OTHER" '
10 {
11     print $1 "_" and "_" var
12
13     sum = $2 + $3
14     print num*sum
15     print $4 "_" other
16 }
17 '
```

8\_awk/vars\_from\_shell.sh

```
1  data_and_abc
2  16.2
3  3_3
```

**Exercise 8.3.** Take another look at your script from exercise 6.5 on page 81. Use `awk` to make it work for floating-point input as well.

## 8.7 awk conditions

Each action block may be preceded by a condition expression. `awk` evaluates the condition and checks whether the result is nonzero (“C-false”). Only if this is the case the corresponding action block is executed. Possible conditions include

- Comparison expressions, which may access or modify variables.

```
1  #!/bin/bash
2  VAR="print"
3  echo "some_test_data_5.3" | awk -v "var=$VAR" '
4      var == "print" { print $2 }
5      var == "noprint" { print "no" }
6      $4 > 2 { print "fulfilled" }
7  '
```

8\_awk/cond\_comp.sh

```

1 test
2 fulfilled

```

- Regular expressions matching the current record

```

1 #!/bin/bash
2
3 {
4     echo "not_important"
5     echo "data_begin:1_2_3"
6     echo "nodata:_itane_i_tae_n_end"
7     echo "other_things"
8 } | awk '
9     # start printing if line starts with data begin
10    /^data begin/ { pr=1 }
11
12    # print current line
13    pr == 1
14
15    # stop printing if end encountered
16    /end$/ { pr=0 }
17 '

```

8\_awk/cond\_regex\_record.sh

```

1 data_begin:1_2_3
2 nodata:_itane_i_tae_n_end

```

- Regular expressions matching the content of a variable (including \$0, \$1, ...)

```

1 #!/bin/bash
2 VAR="15"
3
4 echo "data_data_data" | awk -v "var=$VAR" '
5     # executed if var is a single-digit number:
6     var ~ /^[0-9]$/ {
7         print "var_is_a_single_digit_number"
8     }
9
10    # executed if var is NOT a single-digit
11    var !~ /^[0-9]$/ {
12        print "var_is_not_a_single_digit"
13    }
14
15    $2 ~ /\.a/ {
16        print "2nd_field_has_a_as_second_char"
17    }
18 '

```

8\_awk/cond\_regex\_var.sh

```

1 var_is_not_a_single_digit
2 2nd_field_has_a_as_second_char

```

- Combination of conditions using logical AND (&&) or OR (||)

```

1 #!/bin/bash
2 VAR="15"
3
4 echo "data_data_data" | awk -v "var=$VAR" '
5     var !~ /^[0-9]$/ && $2 == "data" {
6         print "Both_are_true"
7     }
8 '
```

8\_awk/cond\_combination.sh

```

1 Both_are_true
```

- The special **BEGIN** and **END** conditions, that match the beginning and the end of the execution. In other words **BEGIN**-blocks are executed *before* a the first line of input is read and **END**-blocks are executed right before **awk** terminates.

```

1 #!/bin/bash
2
3 {
4     echo "data_data_data"
5     echo "data_data_data"
6     echo "data_data_data"
7 } | awk '
8 BEGIN { number=0 } # optional: all uninitialised
9                 # variables are 0
10 { number += NF }
11 END { print number }
12 '
```

8\_awk/cond\_begin\_end.sh

```

1 9
```

Usually **BEGIN** is a good place to give variables an initial value.

Note, that it is a common source of errors to use an assignment `a=1` instead of a comparison `a==1` in condition expressions. Since the `=` operator returns the result of the assignment (like in C), the resulting action block will be executed independent of the value of `a`:

```

1 #!/bin/bash
2 {
3     echo "not_important"
4     echo "data_begin"
5     echo "1_2_3"
6     echo "end"
7     echo "other_things"
8 } | awk '
9 BEGIN {
10     # initialise pr as 0
11     # printing should only be done if pr==1
```

```

12   pr=0
13   }
14
15   # start printing if line starts with data begin
16   /^data begin/ { pr=1 }
17
18   # stop printing if end encountered
19   /end$/ { pr=0 }
20
21   # print first two fields of current line
22   # error here
23   pr = 1 { print $1 " " $2 }
24 '

```

8.awk/cond\_assign\_error.sh

```

1 not_important
2 data_begin
3 1_2
4 end
5 other_things

```

**Exercise 8.4.** Write a script using inline `awk` code to rebuild the piped version of the command `wc -l`, i.e. your script should count the number of lines of all data provided on *stdin*.

- A good starting point is the backbone script

```

1 #!/bin/bask
2 awk '
3     #your code here
4 '

```

- You will only need to add `awk` code to the upper script.
- Your `awk` program will need three rules: One that initialises everything, one that is run for each line unconditionally and one that runs at the end dealing with the results.
- Decide where the printing should happen. When do you know the final number of lines?
- Once you have a working version: One of the three rules can be omitted. Which one and why?

**Exercise 8.5.** The file `resources/chem_output/qchem.out` contains the logged output of a quantum-chemical calculation. During this calculation two so-called Davidson diagonalisations have been performed. Say we wanted to extract how many iterations steps were necessary to finish these diagonalisations.

Take a look at line 422 of this file. You should notice:

- Each Davidson iteration start is logged with the line

```

1   _ _ _ Starting Davidson _ _ _

```



- A nice table is printed afterwards with the iteration index given in the first column
- The procedure is concluded with the lines

```

1 -----
2  _Davidson_Summary:

```

Use what we discussed so far about `awk` in order to extract the number of iterations both Davidson diagonalisations took. A few hints:

- You will need a global variable to remember if the current record/line you are examining with `awk` is inside the Davidson table or not
- Store/Calculate the iteration count while you are inside the Davidson table
- Print the iteration count when you leave the table and reset your global variable, such that the second table is also found and processed properly.

## 8.8 Important `awk` action commands

`length` returns the number of characters a string has,  
e.g. `length("abra")` would return 4, `length("")` zero.

`next` Quit processing this record and immediately start processing the next one. This implies that neither the rest of this action block nor any of the rules below the current one are touched for this record. The execution begins with the next record again trying to match the first rule. In some sense this statement is comparable to the `continue` in a `bash` loop.

```

1 #!/bin/bash
2
3 {
4     echo record1 word2
5     echo record2 word4
6     echo record3 word6
7 } | awk '
8 BEGIN { c=0 }
9 { c++ }
10 { print c ":_first_rule" }
11 /4$/ { next; print c "_" $1 }
12 { print c ":" $2 }
13 '

```

8\_awk/action\_next.sh

```

1 1:_first_rule
2 1:_word2
3 2:_first_rule
4 3:_first_rule
5 3:_word6

```

**exit** Quit the awk program: Neither the current nor any further record are processed. Just run the code given in the **END**-block and return to the shell. Note, that we can provide the return code with which **awk** exits as an argument to this command.

```

1 #!/bin/bash
2
3 {
4     echo record1 word2
5     echo record2 word4
6     echo record3 word6
7 } | awk '
8 BEGIN { c=0 }
9 { c++ }
10 { print c ":_first_rule" }
11 /4$/ { exit 42; print c "_" $1 }
12 { print c ":_" $2 }
13 END { print "quitting..." }
14 '
15 echo "return_code:_"$?"

```

8\_awk/action\_exit.sh

```

1 1:_first_rule
2 1:_word2
3 2:_first_rule
4 quitting...
5 return_code:_42

```

**print** Print the strings supplied as arguments, followed by a newline character<sup>6</sup>. Just **print** (without an argument) is identical to **print \$0**.

**printf** Formatted print. Can be used to print something, but without a newline in the end.

```

1 #!/bin/bash
2 {
3     echo 1 2 3 4
4     echo 5 6 7 8
5 } | awk '
6 $1 < 4 { printf $3 "_" }
7 $1 > 4 { printf $3 }
8 '

```

8\_awk/action\_printf.sh

```

1 3_7

```

<sup>6</sup>Can be changed. See section 5.3. of the **awk** manual [3] for details

### 8.8.1 Conditions inside action blocks: if

`awk` also has analogous control structures to the ones we discussed in chapter 4 on page 32 for `bash`. We don't want to go through all of these here<sup>7</sup>, just note that conditional branching can also be achieved inside an action block using the `if` control structure:

```

1  if (condition) {
2      action_commands
3  } else {
4      action_commands
5  }
```

where `condition` may be any of the expressions discussed in section 8.7 on page 110. As usual the `else`-block may be omitted.

## 8.9 Further examples

**Example 8.6.** This script defines a simple version of `grep` in just a single line:

```

1  #!/bin/bash
2
3  # here we use DOUBLE quotes to have the shell
4  # insert the search pattern where awk expects it
5  awk "/$1/"
```

8\_awk/ex\_grep.sh

**Example 8.7.** Process some data from the `/etc/passwd`, where “:” or “,” are the field separators

```

1  #!/bin/bash
2  < /etc/passwd awk -v "user=$USER" '
3  # set field separator to be : or , or many of these chars
4  BEGIN {FS="[: ,]+" }
5
6  # found the entry for the current user?
7  $1 == user {
8      # print some info:
9      print "Your_username:         " $1
10     print "Your_uid:              " $3
11     print "Your_full_name:        " $5
12     print "Your_home:            " $6
13     print "Your_default_shell:    " $7
14 }
15 '
```

8\_awk/ex\_passwd.sh

<sup>7</sup>See section 7.4 of the `awk` manual [3] for all the remaining ones.

**Example 8.8.** This program finds duplicated words in a document. If there are some, they are printed and the program returns 1, else 0.<sup>8</sup>

```

1 #!/bin/bash
2 awk '
3     # change the record separator to anything
4     # which is not an alphanumeric (we consider
5     # a different word to start at each alnum-
6     # eric character)
7     BEGIN { RS="^[[:alnum:]]+" }
8     # now each word is a separate record
9
10    $0 == prev { print prev; ret=1; next }
11    { prev = $0 }
12    END { exit ret }
13 '
```

8\_awk/ex\_duplicate.sh

Note, that this program considers two words to be different if they are just capitalised differently.

**Exercise 8.9.** Use `awk` in order to rebuild the command `uniq`, i.e. discard duplicated lines in *sorted* input. Some hints:

- Since input is sorted, the duplicated lines will appear as records right after another in `awk`, i.e. on exactly subsequent executions of the rules.
- Note that whilst `$0` changes from record to record, a usual `awk` variable is global and hence does not.
- The solution takes not more than 2 lines of `awk` code.

*optional* Also try to implement `uniq -c`. It is easiest to do this in a separate script which only has the functionality of `uniq -c`.

**Exercise 8.10.** This exercise deals with writing another script that aids with the analysis of an output file like `resources/chem_output/qchem.out`. This time we will try to extract information about the so-called *excited states*, which is stored in this file.

- If one wants to achieve such a task with `awk`, it is important to find suitable character sequences that surround our region of interest, such that we can switch our main processing routine on and off.
- Take a look at lines 565 to 784. In this case we are interested in creating a list of the 10 excited states which contains their number, their term symbol (e.g. “1 (1) A” or “3 (1) A’”) and their excitation energy.

---

<sup>8</sup>If this program does not work on your computer, make sure that you are using the `awk` implementation `gawk` in order to execute the inline `awk` code in this script. It will not work properly in `mawk`.

- For the processing of the first state we hence need only the five lines

```

1  Excited_state_1(singlet,A) [converged]
2  -----
3  Term_symbol: 1(1) A R^2= 7.77227e-11
4
5  Total_energy: -7502.1159223236 a.u.
6  Excitation_energy: 3.612484 eV
```

Similarly for the other excited states blocks.

Proceed to write the script:

- Decide for a good starting and a good ending sequence.
- How you would extract the data (state number, term symbol, excitation energy) once `awk` parses the excited states block?
- Be careful when you extract the term symbol, because the data will sit in more than one field.
- Cache the extracted data for an excited states block until you reach the ending sequence. Then print it all at once in a nicely formatted table.

## 8.10 awk features not covered

This section is supposed to provide a quick overview of the features of `awk` we did not touch upon. For further reading about `awk` see the `gawk` manual “GAWK: Effective AWK programming” [3]. It is both comprehensive for beginners and very clearly structured. In the following list the paragraph numbers in brackets refer to appropriate sections of the `gawk` manual where more information can be found.

- Formatted printing (§5.5): Controlling the precision of floats printed
- Control structures and statements (§7.4) in `awk`: Loops, `case`, ...
- `awk` arrays (§8)
- `awk` string manipulation functions (§9.1.3): Substitutions, substrings, sorting
- Writing custom `awk` functions (§9.2)
- Reading records with fixed field length (§4.6): Fields separated by the number of characters, not a regex.
- Reading or writing multiple files (§4.9)
- Executing shell commands from within `awk` programs (§4.9)
- Creating `awk` code libraries (§10)
- Arbitrary precision arithmetic using `awk` (§15): Floating point computation and integer arithmetic with arbitrarily-high accuracy.

## Chapter 9

# A word about performance

Most of the time performance is not a key aspect when writing scripts. Compared to programs implemented in a compilable high-level language like `C++`, `Java`, ..., scripts will almost always be manifold slower. So the choice to use a scripting language is usually made because writing scripts is easier and takes considerably less time. Nevertheless badly-written scripts imply a worse performance. So even for `bash` scripts there are a few things which should be considered when large amounts of data are to be processed:

- Use the shell for as much as possible. Calling external programs is by far the most costly step in a script. So this should really only be done when the external program does more than just adding a few integers.
- If you need an external program, choose the cheapest that does everything you need. E.g. only use `grep -E`, where normal `grep` is not enough, only proceed to use `awk`, when `grep` does not do the trick any more.
- Don't pipe between external programs if you could just eradicate one of them. Just use the more feature-rich for everything. See the section below for examples.
- Sometimes a plain `bash` script is not enough:
  - Use a high-level language for the most costly parts of your algorithm.
  - Or use `python` as a subsidiary language: A large portion of `python` is implemented in `C`, which makes it quicker, especially for numerics. Nevertheless many concepts are similar and allow a `bash` programmer to pick up some `python` fairly quickly.

## 9.1 Collection of bad style examples

This section gives a few examples of bad coding style one frequently encounters and is loosely based on <http://www.smallo.ruhr.de/award.html>. Most things have already been covered in much more detail in the previous chapters.

### 9.1.1 Useless use of cat

There is no need to use `cat` just to read a file

```
1 cat file | program
```

because of input redirection:

```
1 < file program
```

### 9.1.2 Useless use of `ls *`

We already said that

```
1 for file in $(ls *); do
2     program "$file"
3     # or worse without the quotes:
4     program $file
5 done
```

is a bad idea because of the word-splitting that happens after command substitution. The better alternative is

```
1 for file in *; do
2     program "$file"
3 done
```

### 9.1.3 Ignoring the exit code

Many programs such as `grep` return a sensible exit code when things go wrong. So instead of

```
1 RESULT=$(< file some_program)
2
3 # check if we got something
4 if [ "$RESULT" ];then
5     do_sth_else
6 fi
```

we can just write

```
1 if <file some_program;then
2     do_sth_else
3 fi
```

### 9.1.4 Underestimating the powers of `grep`

One occasionally sees chains of `grep` commands piped to another, each with just a single word

```
1 grep word1 | grep word2 | grep word3
```

where the command

```
1 grep "word1.*word2.*word3"
```

is both more precise and faster, too.

Also `grep` already has numerous builtin flags such that e.g.

```
1 grep word | wc -l
```

are unnecessary, use e.g.

```
1 grep -c word
```

instead.

### 9.1.5 When `grep` is not enough ...

...then do not use it!

```
1 grep regex | awk '{commands}'
```

can be replaced by

```
1 awk '/regex/ {commands}'
```

and similarly

```
1 grep regex | sed 's/word1/word2/'
```

can be replaced by

```
1 sed '/regex/s/word1/word2/'
```

### 9.1.6 testing for the exit code

It feels awkward to see

```
1 program
2 if [ "$?" != "0" ]; then
3     echo "big PHAT error" >&2
4 fi
```

where

```
1 if ! program; then
2     echo "big PHAT error" >&2
3 fi
```

is much nicer to read and feels more natural, too.



## Appendix A

# Obtaining the files

In order to obtain the example scripts and the resource files, you will need for the exercises, you should run the following commands:

```
1 # clone the git repository:
2 git clone https://github.com/mfherbst/bash-course
3
4 # download the books from Project Gutenberg
5 cd bash-course/resources/gutenberg/
6 ./download.sh
```

All paths in this script are given relative to the directory **bash-course**, which you created using the first command in line 2 above.

All exercises and example scripts should run without any problem on all Linux systems that have the **bash** and the GNU **awk** implementation (**gawk**) installed. On other Unix-like operating systems like Mac OS X it can happen that examples give different output or produce errors, due to subtle differences in the precise interface of the Unix utility programs.

## Appendix B

# Other bash features worth mentioning

### B.1 bash customisation

#### B.1.1 The .bashrc and related configuration files

Not yet written.

#### B.1.2 Tab completion for script arguments

Not yet written.

### B.2 Making scripts locale-aware

Not yet written.

### B.3 bash command-line parsing in detail

#### B.3.1 Overview of the parsing process

When a commandline is entered into an interactive shell or is encountered on a script the **bash** deals with it in the following order

1. Word splitting on the line entered
2. Expansion
  - (a) brace expansion
  - (b) tilde expansion, parameter and variable expansion
  - (c) arithmetic expansion, and command substitution (done in a left-to-right fashion)
  - (d) word splitting
  - (e) pathname expansion
3. Execution

## B.4 Notable bash features not covered

The following list gives some keywords for further exploration into scripting using the `bash` shell. See the bash manual [2] or the advanced bash-scripting guide [5] for more details.

- `bash` arrays
- Brace expansion
- Tilde expansion
- Coprocesses

## Appendix C

# Supplementary information

### C.1 The `mtx` file format

The `mtx` files we use in this course<sup>1</sup> for demonstration purposes, follow a very simple structure

- All lines starting with “%” are comments
- The first line is a comment line.
- The first non-comment line contains three values separated by one or more `<space>` or `<tab>` characters:
  - The number of rows
  - The number of columns
  - The number of non-zero entries
- All following lines — the non-zero entries — have the structure
  - Column index
  - Row index
  - Value

where the values are again separated by one or more `<space>` or `<tab>` chars.

---

<sup>1</sup>We will only use a subset of the full format

# Bibliography

- [1] Eric S. Raymond. The Art of Unix Programming, September 2003. URL <http://www.faqs.org/docs/artu/>.
- [2] Bash manual. URL <https://www.gnu.org/software/bash/manual/>.
- [3] Arnold D. Robbins. GAWK: Effective AWK Programming, April 2014. URL <https://www.gnu.org/software/gawk/manual/>.
- [4] Sed manual. URL <https://www.gnu.org/software/sed/manual/>.
- [5] Mendel Cooper. Advanced bash-scripting guide, March 2014. URL <http://www.tldp.org/LDP/abs/html/>.

# List of Commands

<code>apropos</code>	Search in manpage summaries for keyword
<code>cat</code>	Concatenate one or many files together
<code>cd</code>	Change the current working directory
<code>chmod</code>	Change file or directory permissions (see section 1.3 on page 7)
<code>cut</code>	Extract columns from input
<code>echo</code>	Print something to output
<code>grep</code>	Filter input by pattern
<code>help</code>	Access help for <code>bash</code> builtin commands
<code>info</code>	Access the Texinfo manual for commands
<code>less</code>	View input or a file in a convenient way
<code>ls</code>	List the content of the current working directory
<code>man</code>	Open manual page for a command
<code>mkdir</code>	Create a directory
<code>pwd</code>	Print the current working directory
<code>rmdir</code>	Delete empty folders
<code>rm</code>	Delete files
<code>sort</code>	Sort input according to some parameters
<code>tac</code>	Concatenate files and print lines in reverse order
<code>tee</code>	Write input to file and output
<code>touch</code>	Change the modification time or create a file
<code>uniq</code>	Take a sorted input and discard double lines
<code>wc</code>	Count characters, lines or words on input
<code>whatis</code>	Print a short summary describing a command