# Advanced `bash` scripting

(block course)

Michael F. Herbst
`michael.herbst@iwr.uni-heidelberg.de`
`https://michael-herbst.com`

Interdisziplinäres Zentrum für wissenschaftliches Rechnen
Ruprecht-Karls-Universität Heidelberg

6th – 10th November 2017

# Contents

# List of Tables

## Course abstract

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 implies that writing `bash` scripts is conceptually very simple — essentially like making structured notes of the commands one would need to type in the shell anyway.

When it comes to more involved tasks and more powerful scripts, however, taking a deeper look at the underlying operating system is typically required. After all `bash` scripting is all about properly combining the available programs on the UNIX operating system in a clever way as we will see.

In the first part of the course we will hence revisit some basic concepts of a UNIX-like operating system and review the set of UNIX coreutils 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 (from files ...) using so-called regular expressions and programs like `awk`, `sed` or `grep`.

## Learning 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 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

Familiarity with a UNIX-like operating system like GNU/Linux and the `bash` shell is assumed. For example 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`, ...

It is not assumed, but highly recommended, that you have have some previous experiences with programming or scripting in a UNIX-like operating system.

## Compatibility of the exercises

All exercises and script samples have been tested on Debian 7 "Jessie" with the GNU `bash` 4.3, GNU `sed` 4.2.2 and GNU `awk` 4.1.1. Everything *should* work on other Unix-like operating systems as well, provided that these programs are installed in the denoted version or newer.

On BSD-like operating systems like Mac OS X it may happen, that examples give different output or produce errors, due to subtle differences in the precise interface of the Unix utility programs. Especially `awk` exists in a couple of different variants. Make sure that you install specifically `gawk`, the GNU `awk` implementation. See appendix A.1 on page 128 for some hints how to install the required programs.

## 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

An electronic version of this document is available from `https://michael-herbst.com/teaching/advanced-bash-scripting-2017/`. If you use any part of my work, please include a reference to this URL and as well cite

Michael F. Herbst. Advanced bash scripting 2017. November 2017. URL `http://doi.org/10.5281/zenodo.1038525`.

### 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[1], 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 OS**es, which derive from the traditional AT&T UNIX. Two of the most important modern OSes, Mac OS X and GNU/Linux, are part of this family. In other words: Unix's 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.

[1]Also the creator of the "C" programming language

For the Unix-like OSes this implies (at least in theory):

- The OS is a collection of
  - small helper programs or "utilities", that only do a simple thing (think about `ls`, `mkdir` ...)
  - 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 135 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` <u>`command`</u>[2] or try the tips in section 2.5 on page 20.

[2]We use `teletype` font in this course to denote shell commands or shell scripts. Furthermore we use <u>`underlined`</u> text for parts of the shell command, which represents a descriptive dummy. In this case, for example, <u>`command`</u> is a dummy for an actual shell command like `ls` or `mv`.

### 1.2.1 Accessing files or directories

| | |
|---|---|
| `cd` | Change the current working directory of the shell |
| `ls` | List the content of the current working directory. Important options: |
| | `-l` *long form:* More details |
| | `-a` *all:* Also include hidden files |
| | `-h` *human-readable:* Output sizes in more readable way |
| | `-t` *time:* Sort output by time |
| `pwd` | Print the current working directory of the shell |

### 1.2.2 Modifying files or directories

| | |
|---|---|
| `touch` | Change the modification time if the file exists, else create an empty file, options: |
| | `-t` Change modification time to the one provided |
| `mkdir` | Create a directory |
| `rm` | Delete files. Important options: |
| | `-r` *recursive:* Delete all files and directories in a directory |
| | `-i` Ask before each file deleted |
| | `-I` Ask only in certain circumstances and only once (mass-delete) |
| `rmdir` | Delete empty folders |
| `chown` | Change ownership for a file (see section 1.3 on page 7) |

### 1.2.3 Getting or filtering file content

| | |
|---|---|
| `cat` | Concatenate one or many files together |
| `tac` | Concatenate files and print lines in reverse order |
| `tee` | Write input to a file and to output as well |
| `cut` | Extract `<tab>`-separated columns from input |
| | `-d` *delimiter:* Character to use for the split |
| | `-f` *fields:* Which fields (columns) to print |
| `grep` | Filter input/ by a pattern |
| | `-i` ignore case |
| | `-v` *invert:* only non-matching lines are given |
| | `-o` *only-matching:* print only matching content |
| | `-C` *context:* print n lines of context as well |
| | `-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 [3]. If we run

```
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.

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

```
< 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

```
< 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[4] books pg74 and pg76 that contain the word “hunger”. One could run those two commands one after another

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

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

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

Reminder: The pipe connects the output of the first with the input of the second command. More details on this later.

[3]The top level is the directory in which this pdf is contained

[4]`https://www.gutenberg.org/`

**Example 1.3.** There exists a counterpart to "<", which writes to a file, namely ">". 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

```
< infile cat > outfile
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

```
< infile > outfile cat
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 for example

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

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

```
< 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, mostly for performance and convenience reasons.

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. `sort` is a good representative: The most recent command above is entirely equivalent to

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

In some cases this version, which takes the file as an argument, tends to be faster. Nevertheless I personally prefer the version `< resources/testfile sort -u` since this has a very suggestive syntax: The data flows from the source (`< file`) on the LHS to the sink on the RHS and on the way passes through all commands. Sources on the right, filters (i.e. commands) in the middle and sinks on the right.

### 1.2.4 Other

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

**Example 1.5.** If we want to find help for the commands `tail` and `head`, we could use the manpage

```
man tail
man head
```

Even `man` itself has a manpage, e.g.

```
man man
```

Problems arise with so-called shell builtins. We will talk about this in the next chapter (see section 2.5 on page 20).

### 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 that an article is only from a particular 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 effect of the `grep` options `-A`, `-B`, `-n`, `-H` or `-w` is.
- *(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 the first part 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 right at the top which begin with the special comment character “%”. Suggest a 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. You can check your result by looking at the file and comparing the output with your manual count.

- Provide a sequence of commands that prints *the smallest* value in the third column of `3.mtx`. Again make sure 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` to get 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. Please note, that this section is just a very concise introduction, which sacrifices correctness for simplicity at a couple of places.

### 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 file 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[5]
- *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

  ```
  foo/bar/baz
  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`

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

[5]Which are actually just some special kind of files

- He is the owner and the owner (r/w/x)-bit is 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

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

Further running `chmod g-r file` gave the result

```
drwxr-xr-x 4 mfh agdreuw   4096 Aug 15 19:07 resources
-rw----r-- 1 mfh agdreuw   4115 Aug 15 20:18 file
-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, redirects and exit codes.

## 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 (compile and) execute programs which do the 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 from actual 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 following page

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

Table 2.1: List of noteworthy shells. For more information see `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 (if not already done automatically)

```
. /etc/bash_completion
```

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

### 2.2.2 Accessing the command history

Consider a sequence of commands

```
ls resources/
cd resources/
ls -al
ls matrices
cd matrices
ls -al
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 ↑ Enter

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 for this course, we could just type

```
ls r ⇆ Enter
cd !$ Enter
ls -al Enter
ls m ⇆ Enter
cd !$ Enter
↑ ↑ ↑ Enter
↑ Enter
```

```
ls resources/
cd resources/
ls -al
ls matrices
cd matrices
ls -al
ls -al
```

to achieve the same thing as above.

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

- Start typing
- The shell will automatically display the most recent command matching command line
- type Enter to execute
- type more chars to continue searching
- use ← , → , Home , End , ... to edit the current match, then Enter to run the edited version
- type Ctrl + R to go to the next match further back in the history
- type Ctrl + C 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.

A few other `bash` keystrokes worth trying out:

- Ctrl + W deletes the word on the left
- Esc and then V opens the `vim` editor to edit the commandline[1]

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

```
cd resources
ls images | grep blue       #no file blue exists
ls|grep blue
mkdir grep_red grep_blue
```

Assume as usual 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):

```
ls images | grep red
ls tables
ls resources
```

[1]In order for this to work the `bash` needs to be in `vi` editing mode. Enable this by running the command "`set -o vi`" beforehand.

Note: Count special symbols like "`_`" or "`|`" or combined strokes like Ctrl + R as one keystroke. Also count all Enter 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 Enter key). The character `;` is entirely synonymous to Enter. So typing

```
cd -; ls Enter
```

or

```
cd - Enter
ls Enter
```

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.

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

would start copying the big file `BigFile` to the USB Stick 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

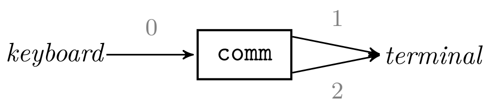
Each command which is run on the terminal per default opens 3 connections to the shell environment:

- *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 interpret this differently and sometimes very unintuitively. 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* and *stderr* separately.
3. Usually one can safely discard the output on *stdout* whereas *stderr* is typically still important to watch for any unforseen issues.
   - ⇒ Implicitly output split into two categories for sensible logging.

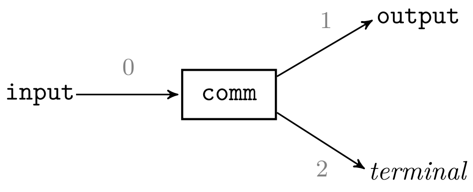
By default *stdin* is connected to the keyboard and both *stdout* and *stderr* are connected to the terminal. Running a `comm` in the shell hence gives a **redirection diagram** like



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

```
< 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

```
< 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`.

| syntax | comment |
|---|---|
| `> file` | Overwrite file with *stdout* |
| `>> file` | append *stdout* to file |
| `2> file` | Overwrite file with *stderr* |
| `2>> file` | append *stderr* to file |
| `&> file` | Overwrite file with *stdout* and *stderr* combined |
| `&>> file` | append *stdout* and *stderr* 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.

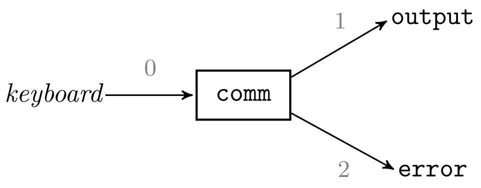
| syntax | comment |
|---|---|
| `\|` | connect *stdout* → *stdin* |
| `\|&` | connect *stdout* and *stderr* → *stdin* |

Table 2.3: Summary of the types of pipes

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

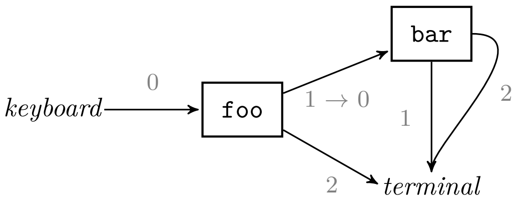
```
comm >output 2>error
```

or pictorially



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 but 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`[2] and say we really wanted to get all errors but we are not very much interested in *stdout*, then running

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

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

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

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

```
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

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

  Draw the redirection diagram. Does it work as intended? If not suggest an alternative 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 yet 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?
- *(optional)* Run just plainly `cut` in a terminal. What do you observe? (Recall that you can quit any execution by Ctrl + C .)

  Some hints to help you explore and explain what is going on:

  – Draw a redirection diagram for just `cat`.

  – Run `cat` followed by Ctrl + D . What happens?

  – Read up on the keywords "end-of-file" or "EOF" in the `bash` manual and on wikipedia[3].

[2]The script contains the *stderr* redirector `>&2`, which allows to redirect command output (like the printing of `echo`) to *stderr* explicitly. There is no need to worry about this right now, we will cover this aspect in more detail in chapter 4 on page 34.

[3]`https://en.wikipedia.org/wiki/End-of-file`

## 2.4 The return code 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 code 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. Note, however, that there exist handy addons to the shell to change this.
- In order to get the exit code of the most recently terminated command one may execute `echo $?`.
- Since this is in turn a command, which may be executed unsuccessfully, this by itself alters the return code and hence effects the value printed by the next execution of `echo $?`.

**Exercise 2.5.** To give you an idea why exit codes are useful as indicators what is going on, do the following

- Run a plain `cat` in your terminal:

  ```
  $ cat
  ```

  It hangs as expected after exercise 2.4 on the preceding page. Now key in Ctrl + D and check the return code by

  ```
  $ echo $?
  ```

  What is the output of the last command?
- Repeat the procedure using Ctrl + C instead of Ctrl + D . What is the result now? What is the reason for the difference, keeping the results of exercise 2.4 in mind.

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

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

```
foo ; bar
foo & bar
```

In both cases 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:

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

A few examples:

- Conditional `cd`:

  ```
  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

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

- Very common when developing code:

  ```
  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

  ```
  ./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 `||`

  ```
  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

```
! ls
```

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

**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. You can access the exit code of the most recent command via `echo $?`.

```
false || true
true && false || true
false && false && true
false || true || false
```

Run the following commands on the shell

```
false | true
true | true
true | false
false | false
false |& true
```

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

**Exercise 2.7.** The main use of `echo` is to print all of its arguments to *stdout*. This is typically not needed a lot in interactive terminal sessions, but in fact one nevertheless can make a lot use of `echo` to provide very particular input to another command using a pipe.

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 to the shell environment? 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`

**Exercise 2.8.** 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 in the above command, like "`Holmes`", "`a`", "`Baker`", "`it`", "`room`" as well.
- Count the number of words in the file `pg1661.txt`

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

Table 2.4: Summary of available commands to get help

**Exercise 2.9.** *(optional)* Go to the directory `resources/directories`.

- Run the rather confusing command

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

  and explain what goes on in terms of the output printed on the terminal.

  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

  ```
  mkdir -p 3/3
  ```

  to create the directory `resources/directories/3/3`. Explain the output of

  ```
  cd 3/3 || cd 4/2 && cd ../4 || cd ../3 && pwd
  ```

As a general hint for this exercise: Try to run each command of the list in a shell and check the action as well as the return code each time, before moving on to the next command which would run.

## 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 these 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 2.10.** 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

Now that we looked at the interactive `bash` shell and what can be achieved using return codes and conditional code execution, we will finally dive into proper scripting in this chapter.

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

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

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

This line, starting with a hash(`#`) and a bang(`!`) — hence the name — tells the OS which program should be used in order to interpret the following commands. If a file with executable rights is encountered, which furthermore begins with a shebang, the OS calls the program specified (in this case `/bin/bash`) and passes the path to the script file[1][2]
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.

[1]Strictly speaking the shebang is not required, since a missing shebang causes the default shell to be used, which is typically the `bash` as well. Therefore this works well in many cases. It is nevertheless good practice to include the shebang as it makes the scripts more portable to other setups.

[2]The precise process is that the OS calls the program in the shebang exactly as it is specified (i.e. including possible extra arguments) and then passes the path to the file as the *last* argument to the program as well. This allows to send the program some flags to infulence the processing of the whole script. Typical examples are `/bin/bash -e` to cause the shell to exit on any errors or `/bin/bash -x` for debugging scripts.

### 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

```
#!/bin/bash
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

```
./hello.sh
```

Otherwise we need to call it by either the full or the relative path of the script file[3]. E.g. if we are in the top directory of the course git repository, we need to execute

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

instead.

### 3.1.2 Scripts and *stdin*

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

```
#!/bin/bash
cat
```

3_simple_scripts/cat.sh

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

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

or pipe to it

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

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

```
< resources/testfile cat
```

or

```
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 51.

For *stdout* and *stderr* unsurprisingly the same applies, namely that all output of all programs called in the script is combined and presented to the caller of the script as a common stream of *stdout* and *stderr* for the full script.

[3]This can be changed by altering the `PATH` variable. See section 6.4 on page 90.

## 3.2 Shell variables

Shell variables are defined using the syntax

```
VAR=value
```

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

```
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.

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

- 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 26), arithmetic expansion (see section 5.1 on page 60 and many more (see manual [2])

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

- `value` may be empty

  ```
  VAR=
  ```

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

  ```
  VAR=123
  VAR2=$VAR23      #fails
  VAR2=${VAR}23    #correct
  ```

- Undefined variables expand to an empty string.
- All `bash` variables are stored as plain strings[4], but they can be interpreted as integers if a builtin command requires this (e.g. `test` — see section 4.2 on page 35)
- Variables can also be deleted[5] using

  ```
  unset VAR
  ```

- A wide range of predefined variables exist (see table 3.1 on the next page)

[4]This can be changed, however, see the `declare` command in the manual [2]

[5]This is not exactly the same thing as setting the variable to the empty string, but still often equivalent.

| 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 (see ex. 4.19 on page 57) |
| `SHELL` | Full path of the shell currently running |

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

### 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

```
#!/bin/bash

echo The first: $1
echo The second: $2
```

3_simple_scripts/first_script.sh

is executed like

```
$ 3_simple_scripts/first_script.sh ham egg and spam
```

we get[6]

```
The␣first:␣ham
The␣second:␣egg
```

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

[6]For command output we will sometimes use special symbols like "␣" to make whitespace characters visible.

**Example 3.1.** If the script

```
#!/bin/bash
echo 0: $0
echo 1: $1
echo 2: $2
echo 3: $3
echo 4: $4
echo @: $@
echo ?: $?
echo "#:␣$#"
```

3_simple_scripts/special_parameters.sh

is executed like

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

we get

```
0:␣3_simple_scripts/special_parameters.sh
1:␣1
2:␣2
3:␣3
4:␣4
@:␣1␣2␣3␣4␣5␣6␣7␣8␣9
?:␣0
#:␣9
```

For more details about the parameter expansion see chapter 5 on page 60.

### 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

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

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

  ```
  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 71), where output redirection is possible. We could hence suppress the error message by

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

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

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

gives

```
one
two
```

Note, that the double quote “"” is crucial here to keep the line break, for reasons we will discuss in section 3.4 on page 29.

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

```
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:

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

- Command substitutions may be nested:

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

**Exercise 3.2.** *(optional)* Write a `bash` quine[7], i.e. a script that produces its source code as output when executed. Hint: The solution has less then 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.** *(optional)* Remind yourself that all commands in a script are connected to the script's *stdin* and *stdout*.

(a) Write a script, which takes a keyword as first argument and `grep`s for this keyword on all data supplied on *stdin*. Test it with a call like

```
$ < resources/gutenberg/pg1661.txt ./your_script.sh bla
```

Suppose that now we want to do some further processing in the same script on the very filtered output we got by the initial `grep`.

(b) Adjust your script to only print the first matching line.

[7]`https://en.wikipedia.org/wiki/Quine_%28computing%29`

(c) Discard what you did in (b) and now print only the last matching line.

(d) Now try to combine (a), (b) and (c): The script should now print only the first and the last matching line, then an empty line (just a plain `echo`) and then all matching lines including the first and the last, exactly as they are returned from the initial `grep` you used in (a). Most importantly the script should always print all these things in exaclty the given order.

You will most probably run into problems. Read on to get an idea how to solve them.

Achieving part (d) of the exercise is a bit tricky, since both the *stdin* and *stdout* of are pretty volatile. Because they both are so-called **streams** everything which is received on *stdin* or sent to *stdout* is gone immediately and cannot be processed again.

In order to be able to use for example *stdin* twice in the same script, one can make use of the following trick:

```
# Cache from stdin
CACHE=$(cat)
# Use it once
echo "$CACHE" | grep ...
# Use it twice
echo "$CACHE" | grep ...
```

where the double quote “`"`” are again neccessary to keep the line breaks.

(e) Try to understand how this works in light of what we discussed in section 3.1.2 on page 23.

(f) Use this (or something similar) to finally solve part (d).

## 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**, i.e. original 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.

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

produces

```
$blubber␣#;\
```

whereas

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

gives rise to

```
foo
```

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

```
echo some very \
  long line of code \
  | grep line
```

```
some␣very␣long␣line␣of␣code
```

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

## 3.4 Word splitting and quoting

Right before the execution of a commandline[8],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[9].

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

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

it first substitutes the commands and parameters:

```
# assume KEYWORD=search and 4=3:
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.

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

[8]See appendix B.3.1 on page 130 for more details how a commandline is parsed

[9]With command lists the shell obviously interprets the first word of each "instruction" as the command to be executed an the remaining ones as corresponding arguments.

Similar to escaping, quoting also causes some special characters to loose 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

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

```
#!/bin/bash

ABC=abcdef
NUM=123
EXAMPLE="$ABC$NUM$(date)␣next"
EXAMPLE2='$ABC$NUM$(data)'
echo "$EXAMPLE"
echo "\"some other example: "␣$EXAMPLE2

CODE="echo"
CODE="$CODE 'test'"
$CODE

#␣we␣can␣quote␣inside␣command␣substitutions:
TEST="$(echo "some␣words")"
echo␣"$TEST"
```

3_simple_scripts/quoting_example.sh

which is

```
abcdef123Mo␣24.␣Aug␣21:07:23␣CEST␣2015␣next
"some␣other␣example:␣␣$ABC$NUM$(data)
'test'
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

```
VAR=
3_simple_scripts/first_script.sh $VAR -h
```

gives

```
The␣first:␣-h
The␣second:
```

Whilst

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

gives

```
The␣first:
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 on the right of the `=` using double quotes
  - ⇒ Same reason as above
- When a variable contains a path be extra careful that you *really* use double quotes everywhere you use the variable
  - ⇒ Paths or filenames may contain spaces
- Use syntax highlighting in your editor[10]
  - ⇒ You will discover missing escapes or closing quotes much more quickly

**Exercise 3.8.** The following script is supposed to search for a keyword in a few selected Project Gutenberg books. Right now it does not quite work as expected. Identify and correct possible problems.

```bash
#!/bin/bash
# Script to extract matching lines from a few project
# gutenberg books and show the results
# $1: Keyword to search for
#
cd resources
ILLIAD=$(<Project Gutenberg selection/The Iliad.txt grep -i $1)
YELLOW=$(<Project Gutenberg selection/The Yellow Wallpaper.txt grep -i $1)

cd Project Gutenberg selection
OTHERS=$(<Dracula.txt grep -H $1; <The Count of Monte Cristo.txt grep -H $1)
COUNT=$(echo '$OTHERS' | wc -l)

echo Searching for the keyword $1:
echo "   Illiad:" $ILLIAD
echo '   Yellow Wallpaper: $YELLOW'
echo We found $COUNT more findings in
echo $OTHERS
```

3_simple_scripts/ex_quoting.sh

[10] vi: `syntax on`, Emacs: `font-lock-mode`

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

```
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, namely

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

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 92. 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.** In this exercise we want to write a script that searches for keywords in a file and displays how many findings there were and where these were found.

- Familiarise yourself with the way the `-n` flag changes the output of grep. How could you use this together with another core UNIX tool to find all line numbers where a particular keyword was found?
- Proceed to write a script that takes a filename as first argument and a search word a second argument. Return the line numbers where the *word* was found.
- Now also display a summarising message, which shows how many matches were found.
- Test your results for some keywords and a few project gutenberg books.
- Now take a look at the `exit` command (`help exit`). It can be used to abort a script prematurely and provide a return code to the caller. Use it to amend your script such that it returns 0 if any match is found and 1 otherwise.
  Hint: You probably need something from section 2.4.1 on page 18.
- Count the number of characters of your script, 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 program flows, where scripts may 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`. You can think of this syntax like sending output to the special, non-existent file `&2`, symbolising *stderr*[1].

  ```
  echo "This goes to stdout"
  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 32)
- The argument `-e` enables the interpretation of a few special escapes (see `help echo` and table 4.1 on the next page)

[1]This redirector is general: It works also in command substitution expressions or anywhere else on the shell

| escape | meaning |
|---|---|
| `\t` | `<tab>` char |
| `\\` | literal \ |
| `\n` | `<newline>` char |

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

## 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 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

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

or

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

e.g.

```
test -z "$VAR"   # Test if a string is empty
test "a" == "b"  # Test if two strings are equal
test 9 -lt 3     # Test if the first number is less than the second
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

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

There are a few things to note:

- The space after the "`[`" and before the "`]`" 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 65)
- 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.

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

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

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

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

- 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 92.

**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[2].
- If the path is a directory `cd` to it and list its content.

## 4.3 Conditionals: `if`

The simplest syntax of the `if` command is

```
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

[2] `man wc`

```bash
#!/bin/bash
if [ 1 -gt 2 ]; then echo "Cannot␣happen"; fi
if [ 1 -gt 2 ]; true; then echo "Will␣always␣be␣true"; fi
if ! cd ..; then echo "Could␣not␣change␣directory" >&2 ; fi
echo $PWD
```

4_control_io/ifexamples.sh

gives output

```
Will␣always␣be␣true
/export/home/abs/abs001/bash-course
```

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

```
if list; then
  list
elif list; then
  list
...
else list
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`-`list`s 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

```bash
#!/bin/bash
USERARG=0 # bash does not know bolean
    # convention is to use 0/1 or y/n for this purpose

# [ "$1" ] is the same as ! [ -z "$1" ]
if [ "$1" ]; then
  USERARG=1
  echo "Dear␣user:␣Thanks␣for␣feeding␣me␣input"
fi

if [ $USERARG -ne 1 ];then
  echo "Nothing␣to␣do"
  exit 0
fi

if [ "$1" == "status" ]; then
  echo "I␣am␣very␣happy"
elif [ "$1" == "weather" ]; then
  echo "No␣clue"
```

```
elif [ "$1" == "date" ]; then
  date
elif [ -f "$1" ];then
  if ! < "$1" grep "robot"; then
    echo "Could␣not␣find␣keyword" >&2
    exit 1
  fi
else
  echo "Unknown␣command:␣$1" >&2
  exit 1
fi
```

4_control_io/more_ifexamples.sh

when run with arg "date" produces the output

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

when run with arg "4_control_io/more_ifexamples.sh"

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

when run with arg "/nonexistent"

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

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

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

and

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

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

## 4.4 Loops: while

`while` syntax:

```
  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.

```bash
#!/bin/bash

C=0
while echo "while:␣$C"; [ $C -lt 3 ]; do
  ((C++))  #increase C by 1
  echo $C
done

# a nested loop
N=5
while [ $N -gt 2 ]; do
  ((N--)) #decrease N by 1
  echo "N␣is␣now␣$N"
  M=2
  while [ $M -lt 4 ]; do
    echo "␣␣␣␣M␣is␣now␣$M"
    ((M++))
  done
done

# more generally the statement
#     ((I++))
# increases the value of the variable I
# by one. Analoguously
#     ((I--))
# decreases it by one.
```

4_control_io/whileloop.sh

produces the output

```
while:␣0
1
while:␣1
2
while:␣2
3
while:␣3
N␣is␣now␣4
␣␣␣␣M␣is␣now␣2
␣␣␣␣M␣is␣now␣3
N␣is␣now␣3
␣␣␣␣M␣is␣now␣2
␣␣␣␣M␣is␣now␣3
N␣is␣now␣2
␣␣␣␣M␣is␣now␣2
␣␣␣␣M␣is␣now␣3
```

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

```
#!/bin/bash

C=0
while echo "while:␣$C"; [ $C -lt 3 ]; do
  ((C++))  #increase C by 1
  echo $C
  [ $C -eq 2 ] && break
done

# a nested loop
N=5
while [ $N -gt 2 ]; do
  ((N--)) #decrease N by 1
  echo "N␣is␣now␣$N"
  M=2
  while [ $M -lt 4 ]; do
    echo "␣␣␣␣M␣is␣now␣$M"
    ((M++))
    [ $M -eq 3 -a $N -eq 3 ] && break
  done
done
```

4_control_io/whilebreak.sh

produces the output

```
while:␣0
1
while:␣1
2
N␣is␣now␣4
␣␣␣␣M␣is␣now␣2
␣␣M␣is␣now␣3
N␣is␣now␣3
␣␣M␣is␣now␣2
N␣is␣now␣2
␣␣M␣is␣now␣2
␣␣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.

```bash
#!/bin/bash

C=0
while echo "while: $C"; [ $C -lt 3 ]; do
  ((C++))  #increase C by 1
  [ $C -eq 2 ] && continue
  echo $C
done

# a nested loop
N=5
while [ $N -gt 2 ]; do
  ((N--)) #decrease N by 1
  echo "N is now $N"
  M=2
  while [ $M -lt 4 ]; do
    ((M++))
    [ $M -eq 3 -a $N -eq 3 ] && continue
    echo "    M is now $M"
  done
done
```

4_control_io/whilecontinue.sh

produces the output

```
while: 0
1
while: 1
while: 2
3
while: 3
N is now 4
    M is now 3
    M is now 4
N is now 3
    M is now 4
N is now 2
    M is now 3
    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.

  ```
  seq 5
  ```

  should give

  ```
  1
  2
  3
  4
  5
  ```

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

  ```
  3
  4
  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

  ```
  seq 1 4 13
  ```

  gives

  ```
  1
  5
  9
  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:

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

- The variable name is subsequently set to all words following in and the full list executed each time thereafter:

```
#!/bin/bash

for word in 1 2 dadongs blubber; do
  echo $word
done

for row in 1 2 3 4 5; do
  for col in 1 2 3 4 5; do
    echo -n "$row.$col␣"
  done
  echo
done
```

4_control_io/forbasic.sh

which gives the output

```
1
2
dadongs
blubber
1.1␣1.2␣1.3␣1.4␣1.5
2.1␣2.2␣2.3␣2.4␣2.5
3.1␣3.2␣3.3␣3.4␣3.5
4.1␣4.2␣4.3␣4.4␣4.5
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:

```
#!/bin/bash

for word in 1 2 dadongs blubber; do
  echo "$word" | grep -q da && continue
  echo $word
done

for row in 1 2 3 4 5; do
  for col in 1 2 3 4 5; do
    [ $col -gt $row ] && break
    echo -n "$row.$col␣"
  done
  echo
done
```

4_control_io/forbreakcontinue.sh

with output

```
1
2
blubber
1.1
2.1␣2.2
3.1␣3.2␣3.3
4.1␣4.2␣4.3␣4.4
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”. This section gives an overview.

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

```
#!/bin/bash
VAR="a␣b␣c␣d"
VAR2="date␣$(date)"
for i in $VAR $VAR2; do
  echo $i   #note: all spaces become line breaks
done | head
```

4_control_io/forparameter.sh

```
a
b
c
d
date
Sa
4.
Aug
13:44:57
CEST
```

- Command substitution

```
#!/bin/bash
N=10
for i in $(seq $N); do
  echo $i
done
```

4_control_io/forcommandsubst.sh

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

- The characters `*` and `?` are so-called **glob** characters and are again treated specially by the `bash`: If replacement of `*` by zero or more arbitrary characters gives the name of an *existing* file, this replacement is done *before execution* of the commandline. In a similar manor `?` is may be replaced by exactly *one* arbitrary character if this leads to the name of a file[3]. In the context of `for` loops this is usually encountered like so

```bash
#!/bin/bash
cd resources/matrices/
for i in *.mtx; do
  echo $i
done

# there is no need for a file to be in pwd
for i in ../matrices/?a.mtx; do
  echo $i
done

#NOTE: Non-matching strings still contain * or ?
for i in /non?exist*ant; do
  echo $i
done
```

4_control_io/forwildcard.sh

```
3a.mtx
3␣b.mtx
3.mtx
bcsstm01.mtx
lund_b.mtx
../matrices/3a.mtx
/non?exist*ant
```

- Of course combinations of these in one `for` loop in any arbitrary order are fine as well.

[3]This process is called *pathname expansion* and a few other glob patterns exist as well. See [2] for details.

A word of warning: The paradigm

```
for file in $(ls); do
  # some stuff with $file
done
```

is extremely problematic, since files with spaces are not properly accounted for[4] Compare the following results with the last example we had above

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

4_control_io/forlscommandsubst.sh

```
resources/matrices/3a.mtx
resources/matrices/3
b.mtx
resources/matrices/3.mtx
resources/matrices/bcsstm01.mtx
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 `grep`s 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 `grep`ping in the files
  - You may assume all books of Project Gutenberg to end in the extension `.txt`.
  - *(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

```
pg74.txt␣␣45␣␣1045
pg345.txt␣60␣␣965
```

- *(optional)* Suppress the output of books without any match.

[4]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 130 for more details.

**Exercise 4.7.** *(demo)* 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.

## 4.6 Conditionals: `case`

The `case` command has the following basic syntax:

```
case word in
  pattern) list ;;
  pattern) list ;;
  # as many such statements as desired ...
esac
```

- The command tries to match `word` against one of the `pattern`s provided[5]
- If a match occurs the respective `list` block is executed.
- Both the `word` as well as the inspected `pattern`s 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.

```
#!/bin/bash
VAR=$@      # VAR assigned to all arguments
case $VAR in
   a)   echo "VAR␣is␣\"a\""
        ;;  #<- do not omit these
   l*)  echo "VAR␣starts␣with␣l"
        ;;
   l?)  echo "VAR␣is␣l␣and␣something"
        echo "Never␣matched"
        # because it is more speciffic
        # than pattern l* above
        ;;
   $1)  echo "VAR␣is␣\$1"
        # i.e. there is none or only one arg
        # because exaclty then $1 == $@
        ;;
   *)   echo "VAR␣is␣something␣else"
        ;;
esac
```

4_control_io/caseexample.sh

[5]In the sense of globbing like in pathname expansion

The output is

- `4_control_io/caseexample.sh lo`

```
VAR␣starts␣with␣l
```

- `4_control_io/caseexample.sh`

```
VAR␣is␣$1
```

- `4_control_io/caseexample.sh bash is so cool`

```
VAR␣is␣something␣else
```

- `4_control_io/caseexample.sh unihd`

```
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`:

```
#!/bin/bash
# assume we allow the arguments -h, -f and --show
# assume further that after -f there needs to be a
# filename following
#
FILE=default_file    # default if -f is not given
while [ "$1" ]; do   # are there commandline arguments left?
   case "$1" in      # deal with current argument
      -h|--help)  echo "-h␣encountered"
                  ;;
      # it is common to have "long" and "short" options
      -f|--file)  shift # access filename on $1
                  echo "-f␣encountered,␣file:␣$1"
                  FILE=$1
                  ;;
      --show)     echo "--show␣encountered"
                  ;;
      *)          echo "Unknown␣argument:␣$1" >&2
                  exit 1
   esac
   shift # discard current argument
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`

```
-h encountered
--show encountered
```

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

```
-f encountered, file: file
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

```
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

```
%%MatrixMarket␣matrix␣coordinate␣real␣symmetric
```

So if we execute

```
#!/bin/bash
< resources/matrices/3.mtx read COMMENT MTX FLAGS
echo "com:␣␣␣$COMMENT"
echo "mtx:␣␣␣$MTX"
echo "flags:␣$FLAGS"
```

4_control_io/readexample.sh

we obtain

```
com:   %%MatrixMarket
mtx:   matrix
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. This way we can easily check, 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.

**Example 4.10.** Consider the script

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

4_control_io/readerror.sh

We run it and enter a few numbers:

```
$ 4_control_io/readerror.sh
Please type 3 numbers >1 2 3
```

it gives

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

and similarly

```
$ 4_control_io/readerror.sh
Please type 3 numbers >1 2
```

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

On the other hand if we issue a Ctrl + D — the EOF character — we get

```
Did not understand your results, please try again
```

followed by the prompt to enter numbers 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 4.11.** Consider the script

```
#!/bin/bash
cat
cat
```

4_control_io/cat_script.sh

If we run it like so

```
< 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 be 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 `grep`s:

```
grep match
grep "i will never match anything"
```

the second `grep` is pointless. If subsequent `grep`s on *stdin* are desired, one usually employs a temporary caching variable in order to circumvent these problems[6]:

```
CACHE=$(cat)
echo "$CACHE" | grep match
echo "$CACHE" | grep "i have a chance to match sth."
```

**Example 4.12.** In contrast to `cat` the `read` only reads a single line. Therefore a script may swap the first two lines of *stdin* like this

```
#!/bin/bash
read OLINE       # read the first line
read LINE        # read the second line
echo "$OLINE"    # print second line
echo "$LINE"     # print first line
cat
```

4_control_io/swaplines.sh

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

[6]like we introduced it in exercise 3.4 on page 27

**Exercise 4.13.** 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

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

it should provide the output

```
some
other
```

and when called like

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

it should only print

```
other
```

**Exercise 4.14.**

- Write a script which asks the user for two numbers `N` and `M`. and then counts from `N` to `M`. You may assume that `N` << `M`.
- *(optional)* Lift the assumption and generalise your script such that it will count from the smaller of `N` and `M` to the larger of `N` and `M`.

### 4.7.3 The `while read line` paradigm

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

```
#!/bin/bash
while read line; do
  echo $line
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 an 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

```
#!/bin/bash

if [ "$1" == "-h" ];then
  echo "Scipt␣adds␣line␣numbers␣to␣a␣file␣on␣\$1"
  exit 1
fi

if [ ! -f "$1" ]; then
  echo "File␣$1␣not␣found" >&2
  exit 1
fi

C=0
while read line; do
  echo "$C:␣␣$line"
  (( C++))
done < "$1"
```

4_control_io/addlinenumbers.sh

Note: The `<` input arrow has to be added *after* the `done` and not in front of the `while` or similar — otherwise an error results.

- we can pipe the output of a command to it

```
#!/bin/bash
if [ "$1" == "-h" ];then
  echo "Scipt␣sorts␣lines␣of␣file␣\$1␣and␣adds␣indention"
  echo "Sorted␣file␣is␣written␣to␣\$1.sorted"
  exit 1
fi

if [ ! -f "$1" ]; then
  echo "File␣$1␣not␣found" >&2
  exit 1
fi

echo "Writing␣sorted␣data␣to␣\"$1.sorted\""
< "$1" sort | while read line; do
  echo "␣␣␣$line"
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 4.15.** We want to write a more general version of exercise 3.3 on page 27.

- 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 that the value to `--from`
- Change your code such that the line `--from` is moved to the line `--to`.
- Be careful when comparing line numbers to variables that may contain a string:

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

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

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

**Exercise 4.16.** *(optional)*

- Run the following lines of code

  ```
      CACHE=$(echo "$CACHE"; echo "line␣1")
      CACHE=$(echo "$CACHE"; echo "line␣2")
      CACHE=$(echo "$CACHE"; echo "line␣3")
      echo "$CACHE"
  ```

  by pasting them into a script or just by running them in a terminal interactively. Can you explain the result of the final echo?
- How would you alter the code in order to reverse the order in which the lines are printed by `echo`. Try to achieve this without changing the order in which the strings `line 1`, `line 2` or `line 3` appear.
- Use your above findings with the `while read line` paradigm to build a simple `bash` version of the `tac` command, where all input on *stdin* is printed to *stdout* in reverse order.
- Note: The final solution takes just 5 lines of `bash`.

**Exercise 4.17.** Recall that `read` can take more than one argument to read into multiple variables at once.

- Assume you will get some data on *stdin*, which consists of a few columns separated by one ore 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.

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

- How does it deal with multiple spaces compared to `cut`?

**Exercise 4.18.** *(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

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

Table 4.3: The most important options of find

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 25 we already mentioned the variable `IFS`.

- `IFS` is short for "internal field separator"
- This variable is considered in the word splitting whenever the shell performs word splitting (see appendix B.3.1 on page 130), i.e. especially after parameter expansion and command substitution have happened.
- Its value specifies the characters at which commandline is split into individual words during word splitting.
- Default value: `<space><tab><newline>`

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

- Manipulation of the way `for` loops iterate:

```bash
#!/bin/bash
# Store the original field separator
# and change to + for the next for loop
OIFS=$IFS
IFS="+"
for number in 4+5+6+7; do
  echo $number
done

# it is good practice to change IFS back to the
# original after you used the trick, otherwise
# all sorts of crazy errors can occur at a later
# point during the script
IFS=$OIFS
for val in 1 2 3; do
  echo $val;
done
```

4_control_io/IFS_for.sh

```
first␣loop
4
5
6
7

second␣loop
1
2
3
4
```

- Influencing `read`:

```bash
#!/bin/bash
# In this script we want to parse the /etc/passwd
# file where the columns of information are
# separated by : in each line.
OIFS="$IFS"
IFS=":"
echo "------------------"
while read user pw uid gid gecos home shell; do
  echo "Username:␣␣␣␣␣␣$user"
  echo "User␣id:␣␣␣␣␣␣␣$uid"
  echo "Group␣id:␣␣␣␣␣␣$gid"
  echo "Home␣dir:␣␣␣␣␣␣$home"
  echo "Default␣shell:␣$shell"
  echo "------------------"
done < /etc/passwd
IFS=$OIFS
```

4_control_io/IFSread.sh

```
------------------
Username:␣␣␣␣␣␣root
User␣id:␣␣␣␣␣␣␣0
Group␣id:␣␣␣␣␣␣0
Home␣dir:␣␣␣␣␣␣/root
Default␣shell:␣/bin/bash
------------------
Username:␣␣␣␣␣␣daemon
User␣id:␣␣␣␣␣␣␣1
Group␣id:␣␣␣␣␣␣1
Home␣dir:␣␣␣␣␣␣/usr/sbin
Default␣shell:␣/usr/sbin/nologin
------------------

...
```

**Exercise 4.19.** The shell uses the following procedure to lookup the path of the commands to be executed[7]:

- 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

```
vim testfile
```

has the first word/command `vim`. Assume

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

Then a lookup reveals that the file `/usr/bin/vim` exists and is executable. So this file is executed with `testfile` as the argument.

There exists a commandline 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

[7]This is a slight simplification since e.g. commandlines can be far more complex.

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 ouf 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.

### 4.9.1 Script structure

- Always use a `shebang` as the first line of your script.
- A block of code doing a task should have a comment explaining what goes in and what the expected result should be. This is especially true for functions (see section 6.2 on page 79).
- Whenever funny `bashisms`[8] are used that could make code unclear, explain what happens and why. Think about the future you `;)`.
- One script should only do one job and no more. Split complicated tasks into many scripts. This makes it easier to code and easier to reuse.
- Try to design scripts as filters, i.e. better read from *stdin* and write to *stdout* rather than to/from files. This simplifies code reuse, too. Think about the core Unix tools: The utilities you use most often are very likely just some kind of elaborate filter from *stdin* to *stdout*.
- Use shell functions (see section 6.2 on page 79) to structure your script. Have a comment what each function does.

[8]Chain of special characters which look like magic to someone new to `shell` scripting

### 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. The reason is that using `read` interferes with reading data from *stdin* (cf. section 4.7.2 on page 51).
- 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*.

⇒ Again all of this can be summarised as "design each script as a filter"

### 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 130) 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 24).

## 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

```
((expression))
```

Everything within the brackets is subject to **arithmetic evaluation**[1]:

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

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

- The full range of parameter expansion expressions is available (see section 5.3 on page 67). One may, however, also access or assign variables without the leading $

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

  ```
  7
  ```

- Exception: Positional parameters are *not* available

[1]The precise rules with regards to operator precedence and evaluation order are more or less identical to those of the C programming language

- All common operators are available:
  - `+ -` addition, subtraction
  - `* / %` multiplication, (integer) division[2] , remainder
  - `**` exponentiation
  - `name++ ++name name-- --name` increment and decrement operators
  - `+= -= *= /= %=` Infix assignment

```
#!/bin/bash
((
  C=1,
  D=2,

  SUM=C+D,
  DIV=C/D,
  MOD=C%D,
  EXP=D**4
))
echo "C:␣␣␣␣␣␣␣␣$C"
echo "D:␣␣␣␣␣␣␣␣$D"
echo
echo "SUM=C+D:␣␣$SUM"
echo "DIV=C/D:␣␣$DIV"
echo "MOD=C%D:␣␣$MOD"
echo "EXP=D**4:␣$EXP"

((
  CAFTER=C++,
  DAFTER=--D
))
echo "C:␣␣␣␣␣␣␣␣$C"
echo "D:␣␣␣␣␣␣␣␣$D"
echo "CAFTER:␣␣␣$CAFTER"
echo "DAFTER:␣␣␣$DAFTER"
```

5_variables/arith_operator_ex.sh

```
C:␣␣␣␣␣␣␣␣1
D:␣␣␣␣␣␣␣␣2

SUM=C+D:␣␣3
DIV=C/D:␣␣0
MOD=C%D:␣␣1
EXP=D**4:␣16
C:␣␣␣␣␣␣␣␣2
D:␣␣␣␣␣␣␣␣1
CAFTER:␣␣␣1
DAFTER:␣␣␣1
```

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

[2]This is meant to say that the `bash` cannot return floating point results and instead will truncate all non-integer results to the next integer in the direction towards zero. We will go into this further down.

- 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`)

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

5_variables/arith_logic_ex.sh

```
0
1
1
0
```

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

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

```
1
```

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

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

```
0
```

Especially the last two points 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

```
#!/bin/bash

C=1
while ((++C < 40)); do
  if ((C % 3 == 0));then
    echo "divisible␣by␣3:␣$C"
  fi
done
```

5_variables/arith_replacement.sh

```
divisible␣by␣3:␣3
divisible␣by␣3:␣6
divisible␣by␣3:␣9

...

divisible␣by␣3:␣33
divisible␣by␣3:␣36
divisible␣by␣3:␣39
```

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

```
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`.

```
#!/bin/bash
MAX=4
for((I=0; I<MAX; ++I)); do
  echo $I
done
echo
for((I=MAX-1; I>=0; --I));do
  echo $I
done
```

5_variables/arith_for_cloop.sh

```
0
1
2
3

3
2
1
0
```

Finally **arithmetic expansion** is invoked by a syntax like

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

- `expression` is subject to arithmetic evaluation as described above.
- This implies that the last arithmetic subexpression inside the `(( ... ))` gives the value of the arithmetic expansion. In other words the value of `$((expr1, expr2))` is solely determined by `expr2`.
- The whole construct is replaced by the final value the `expression` results in.
- The return code of `(( ))` is not available.
- The expression may be used just like a parameter expansion `${VAR}`

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

5_variables/arith_expansion.sh

```
You␣kindly␣supplied:␣␣␣␣5
The␣square␣is:␣␣␣␣␣␣␣␣␣␣␣25
I␣can␣add␣some␣stuff:␣␣␣8
```

A big drawback on all these paradigms is that the `bash` only supports integer arithmetic. Even intermediate values are only stored as integers, e.g.

```
#!/bin/bash
echo $((100*13/50))
echo $((13/50*100))
```

5_variables/arith_intermediate_floats.sh

```
26
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?

```
((B=0))
echo $((B=0))
echo $((B=0)) | grep 0
(( 3 - 4 ))
(( 0*4, 0 ))
(( 0*4, 3 ))
for((C=100,A=99 ; C%A-3 ; C++,A-- )); do ((B=(B+1)%2)) ;done; ((B))
((B=1001%10)) | grep 4 || (( C=$(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:

  ```
  A="string"
  echo $((A+0))
  A="4"
  echo $((A+0))
  ```

  contrast this with

  ```
  A="string"
  echo $A
  A="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 cause 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

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

```
26.00000000000000000000
```

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

```
echo "3<4" | bc -l  # gives true
echo "1␣==␣42" | bc -l  # gives false
```

```
1
0
```

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

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

```
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)

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

  ```
  110
  ```

- or the number of decimal figures

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

  ```
  .8333
  ```

Next to `bc` one can in principle also use any other floating-point aware program like `awk` (see chapter 8 on page 104) or `python`. Most of the time it is, however, still sensible to use `bc`, since it is extremely small, 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 46 gives output of the form

```
pg74.txt	45	1045
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 46, e.g.

  ```
  RESULT=$(./4_control_io/book_parse.sh "$PATTERN")
  ```

- parses the respective script output in the variable `RESULT`.
- 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 built 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 132.

- Write a script that takes an `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/3 b.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*

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

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

```
#!/bin/bash
A="value"
echo 1 ${A:="new value"}
echo 2 $A

unset A
echo 3 ${A:="newer value"}
echo 4 $A
```

5_variables/pexp_assign_default.sh

```
1 value
2 value
3 newer value
4 newer value
```

- *use-default*

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

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

```
#!/bin/bash
DEFAULT="default"
A="value"
echo 1 ${A:-${DEFAULT}}
echo 2 $A

unset A
echo 3 ${A:-${DEFAULT}}
echo 4 $A
```

5_variables/pexp_use_default.sh

```
1␣value
2␣value
3␣default
4
```

- *use-alternate*

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

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

```bash
#!/bin/bash
ALTERNATE="alternate"
A="value"
echo 1 ${A:+${ALTERNATE}}
echo 2 $A

unset A
echo 3 ${A:+${ALTERNATE}}
echo 4 $A
```

5_variables/pexp_use_alternate.sh

```
1␣alternate
2␣value
3
4
```

- *parameter length*

```
${#parameter}
```

Expands into the number of characters parameter currently has.

```bash
#!/bin/bash
STRING="1234567"
ABC="thirteen"
echo ${#STRING}
echo ${#ABC}
```

5_variables/pexp_length.sh

```
7
8
```

- *substring expansion*

```
${parameter:offset}
${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.

```
#!/bin/bash
VAR="some␣super␣long␣string"
LEN=${#VAR}

# remove first and last word:
echo ${VAR:4:LEN-10}

# since parameter expansion is allowed
# in arithmetic expressions
echo ${VAR:2+2:${#VAR}-10}
```

5_variables/pexp_substr.sh

```
super␣long
super␣long
```

- *pattern substitution*

```
${parameter/pattern/string}     # one occurrence
${parameter//pattern/string}    # global
```

parameter is expanded and the *longest* match of pattern[3] 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.

```
#!/bin/bash
VAR="some␣super␣long␣string"
SE_PAT="s*e"
R_PAT="?r"
REPLACEMENT="FOOOO"

# the longest match is replaced:
echo ${VAR/$SE_PAT/$REPLACEMENT}
echo ${VAR/$R_PAT/$REPLACEMENT}

# all matches are replaced
echo ${VAR//$R_PAT/$REPLACEMENT}
```

5_variables/pexp_subst.sh

```
FOOOOr␣long␣string
some␣supFOOOO␣long␣string
some␣supFOOOO␣long␣sFOOOOing
```

[3]Again a pattern in the sense of a glob expression like for pathname expansion.

**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` $\rightarrow$ `tset`     `abcdef` $\rightarrow$ `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 than 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

```
{ list; }
```

- Both the space in the beginning as well as the `;` in the end are crucial.
- The `;` may — as usual — be replaced by a line break, however.
- 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

```
#!/bin/bash
< resources/matrices/3.mtx grep -v "%" | {
  read ROW COL ENTRIES
  echo "Number␣of␣rows:␣␣␣␣␣␣$ROW"
  echo "Number␣of␣cols:␣␣␣␣␣␣$COL"
  echo "Number␣of␣entries:␣␣␣$ENTRIES"
  echo "List␣of␣all␣entries:"
  while read ROW COL VAL; do
    echo "␣␣␣M($ROW,$COL)␣=␣$VAL"
  done
}
```

6_functions_subshells/group_unpack.sh

```
Number␣of␣rows:␣␣␣␣␣␣3
Number␣of␣cols:␣␣␣␣␣␣3
Number␣of␣entries:␣␣␣9
List␣of␣all␣entries:
␣␣␣M(1,1)␣=␣1
␣␣␣M(1,2)␣=␣1
␣␣␣M(1,3)␣=␣1
␣␣␣M(2,1)␣=␣2
␣␣␣M(2,2)␣=␣2
␣␣␣M(2,3)␣=␣2
␣␣␣M(3,1)␣=␣3
␣␣␣M(3,2)␣=␣3
␣␣␣M(3,3)␣=␣3
```

- sending data to a file

```bash
#!/bin/bash

{
  echo "A␣first␣message␣to␣stderr" >&2
  echo "Grepping␣for␣fish" | grep -w fish
  echo "Hello␣to␣stdout"
  echo "Again␣to␣to␣stderr" >&2
} > /tmp/file-stdout 2> /tmp/file-stderr

# print content
echo "Everything␣in␣/tmp/file-stdout:"
echo -----------
cat /tmp/file-stdout
echo -----------
echo
echo "Everything␣in␣/tmp/file-stderr:"
echo -----------
cat /tmp/file-stderr
echo -----------

# cleanup
rm /tmp/file-stdout /tmp/file-stderr
```

6_functions_subshells/group_write_file.sh

```
Everything␣in␣/tmp/file-stdout:
-----------
Grepping␣for␣fish
Hello␣to␣stdout
-----------

Everything␣in␣/tmp/file-stderr:
-----------
A␣first␣message␣to␣stderr
Again␣to␣to␣stderr
-----------
```

- There surely are alternative ways in order to write many lines of data to a file. For example instead of

  ```
  {
    echo line1
    echo line2
    echo line3
  } > /tmp/file
  ```

  we could also use

  ```
  echo line1 > /tmp/file
  echo line2 >> /tmp/file
  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 writing the code, such that some of the stuff gets accidentally overwritten.
  - If we want to rearrange the order in which the data gets written at any later point, we need to be careful to change the `>` and `>>` redirects in a consistent manor as well. One easily forgets this.

### 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:

```
( 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.

**Example 6.1.**

```
#!/bin/bash
A=3
B=6
pwd
(
  A=5  #locally change varible
  echo "Hello␣from␣subshell:␣A:␣$A␣␣␣B:␣$B"
  cd ..  #locally change directory
  pwd
)
echo "Hello␣from␣main␣shell:␣A:␣$A␣␣␣B:␣$B"
pwd
```

6_functions_subshells/subshell_example.sh

```
/export/home/abs/abs001/bash-course
Hello␣from␣subshell:␣A:␣5␣␣␣B:␣6
/export/home/abs/abs001
Hello␣from␣main␣shell:␣A:␣3␣␣␣B:␣6
/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 cleanup cannot be forgotten.

```
#!/bin/bash
# Here want to do some stuff in the PWD
echo "The␣list␣of␣files␣in␣the␣PWD:"
ls | head -n 4
(
  # Alter the environment:
  # different working directory and IFS separator
  cd resources/matrices
  IFS=":"

  echo
  echo "The␣list␣of␣files␣in␣resources/matrices"
  ls | head -n4

  echo
  echo "Some␣paths:"
  for path in $PATH; do
    echo $path
  done | head -n4
)

# and we are back to the original
echo
for i in word1:word2; do
  echo $i
done
```

6_functions_subshells/subshell_cdifs.sh

```
The␣list␣of␣files␣in␣the␣PWD:
1_intro_Unix
2_intro_bash
3_simple_scripts
4_control_io

The␣list␣of␣files␣in␣resources/matrices
3a.mtx
3␣b.mtx
3.mtx
bcsstm01.mtx

Some␣paths:
/usr/local/bin
/usr/bin
/bin
/usr/local/games

word1:word2
```

### 6.1.3 Implicit subshells

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

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

```bash
#!/bin/bash
C=0 # initialise counter
< resources/testfile grep "e" | while read line; do
  # subshell here!
  ((C++))
done
# Postprocessing not in subshell any more:
echo "We␣found␣$C␣matches␣for␣\"e\"."
```

6_functions_subshells/subshell_pipes.sh

```
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:

```bash
#!/bin/bash
COUNT=$(< resources/testfile grep "e" | {
  C=0
  while read line; do
    ((C++))
  done
  echo $C
})
```

```

# Do postprocessing on COUNT, e.g. print
echo "We␣found␣$COUNT␣matches␣for␣\"e\"."
```

6_functions_subshells/subshell_pipes_correct.sh

```
We␣found␣4␣matches␣for␣"e".
```

If the post-processing can be done inside the command group as well, like in this simple case, we could alternatively do

```
#!/bin/bash
< resources/testfile grep "e" | {
  C=0
  while read line; do
    ((C++))
  done
  echo "We␣found␣$C␣matches␣for␣\"e\"."
}
```

6_functions_subshells/subshell_pipes_correct2.sh

- Command substitutions: Usually less of a problem

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

6_functions_subshells/subshell_commandsubst.sh

```
0
-1
```

- Since command substitutions starts 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 `$( ... )` are lost. We only have *stdout*, which we can use to retrieve data in the main shell from the executed commands. The solution to this problem is to pack the data inside the subshell and to unpack it later, e.g.

```
#!/bin/bash
# Some input state inside the main shell
N=15
RES=$(
  # Do calculations in the subshell
  SUM=$((N+13))
  SQUARE=$((N*N))

  # Pack the results with a :
  # i.e. echo them separated by a :
  echo "$SUM:$SQUARE"
)
```

```

# now use cut to unpack them and recover
# the individual values
SUM=$(echo "$RES" | cut -d: -f1)
SQUARE=$(echo "$RES" | cut -d: -f2)

# Echo the results:
echo "sum: $SUM"
echo "square: $SQUARE"
```

6_functions_subshells/subshell_pack.sh

```
sum: 28
square: 225
```

**Exercise 6.2.** The fact that subshells forget certain things once they are left, is not only a pain, but can be really useful as well. A typically example is if one wants to do a particular task for all subdirectories of a particular directory.

In this exercise, we want to design a script, which prints the name of the largest file for each subdirectory of the `resources` directory of the `bash` course.

There are many ways to do this. For the sake of the exercise do not use an external program like `find` to traverse the directory tree, but instead really `cd` into a directory first, before finding the largest file in it.

A few hints:

- For now there is no need to recurse, i.e. just go into all immediate subdirectories of `resources`, find the largest file and print it. No need to look at subdirectories of subdirectories ...
- If multiple files have the same size, just print one of them for simplicity.
- The result could look something like:

```
Directory:             largest file
-----------------------------------
resources/chem_output:    qchem.out
resources/directories:
resources/gutenberg:    pg135.txt
resources/matrices:    lund_b.mtx
resources/Project Gutenberg selection:    The Count of Monte 
    ↪Cristo.txt
```

- A very helpful commands for this exercise is `wc`. You may use `ls` as well, but if you think the wrong way, the exercise can become complicated.

**Exercise 6.3.** This script does not produce the results the author expected. Spot the errors and correct them. You should find roughly 3 problems.

```
#!/bin/bash
# initial note:
#    this script is deliberately made cumbersome
#    this script is bad style. DO NOT COPY
KEYWORD=$1

ERROR=0    # Error flag
[ ! -f "bash_course.pdf" ] && (
  echo "Please␣run␣at␣the␣top␣of␣the␣bash_course␣repository" >&2
  ERROR=1
)

# Change to the resources directory
if ! cd resources/; then
  echo "Could␣not␣change␣to␣resources␣directory" >&2
  echo "Are␣we␣in␣the␣right␣directory?"
  ERROR=1
fi

[ $ERROR -eq 1 ] && (
  echo "A␣fatal␣error␣occurred"
  exit 1
)

# List of all matching files
MATCHING=

# Add files to list
ls matrices/*.mtx gutenberg/*.txt | while read line; do
  if < "$line" grep -q "$KEYWORD"; then
    MATCHING=$(
      echo "$MATCHING"
      echo $line
    )
  fi
done

# count the number of matches:
COUNT=$(echo "$MATCHING" | wc -l)

if [ $COUNT -gt 0 ]; then
  echo "We␣found␣$COUNT␣matches!"
  exit 0
else
  echo "No␣match" >&2
  exit 1
fi
```

6_functions_subshells/subshell_exercise.sh

```
We␣found␣1␣matches!
```

## 6.2 `bash` functions

The best way to structure shell code by far are `bash` functions. Functions are defined[1] like

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

or

```
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*.

```
#!/bin/bash
# Typically functions defined at the top and
# global code at the bottom

readfct() {
  # Read two lines from stdin
  read test
  read test2

  # Write them to stdout
  echo "Your␣input:"
  echo $test2 $test
}

log_error() {
  # Write to stderr only
  echo "ERROR:␣Something␣bad␣happened!" >&2
}

# Still see the error, since only stdout redirected
log_error >/dev/null

# Pipe to/from a function
{
  echo line1
  echo line 2
} | readfct | grep 2
```

6_functions_subshells/fun_pipe.sh

```
ERROR:␣Something␣bad␣happened!
line␣2␣line1
```

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

[1]There are more ways to define functions. See the `bash` manual [2] for the others

```
#!/bin/bash

argument_analysis() {
  echo $1
  echo $2
  echo $@
  echo $#
}

# call function
argument_analysis 1 "2␣3" 4 5
```

6_functions_subshells/fun_arguments.sh

```
1
2␣3
1␣2␣3␣4␣5
4
```

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

```
#!/bin/bash

comment_on_letter() {
  if [ "$1" != "a" ]; then
    echo "Gwk␣...␣I␣only␣like␣a,␣not␣$1"
    return 1
  fi
  echo "Ah␣...␣a␣is␣my␣favorite␣letter"
}

is_letter_b() {
  [ "$1" == "b" ]
}

VAR=b
if is_letter_b "$VAR"; then
  comment_on_letter "$VAR"
  echo "RC␣of␣comment_on_letter:␣$?"
fi

comment_on_letter "a"
echo "RC␣of␣comment_on_letter:␣$?"
```

6_functions_subshells/fun_return.sh

```
Gwk␣...␣I␣only␣like␣a,␣not␣b
RC␣of␣comment_on_letter:␣1
Ah␣...␣a␣is␣my␣favorite␣letter
RC␣of␣comment_on_letter:␣0
```

- All variables of the calling shell are available inside the function. They may not only be read, but also modified. If the version `fun() { ... }` is used, this modification is global, i.e. effects the shell variables of the caller as well.
- To circumvent this issue a variable inside a function may be defined as `local`. In this case they are only available to the function and all its children, i.e. other functions which may be called by directly or indirectly[2] by said function. The global state of the caller is not effected.

```
#!/bin/bash
# Global variables:
VAR1=vvv
VAR3=lll

variable_test() {
  local FOO=bar
  echo $VAR1
  VAR3=$FOO
}

echo "--$VAR1--$FOO--$VAR3--"
variable_test
echo "--$VAR1--$FOO--$VAR3--"
```

6_functions_subshells/fun_vars.sh

```
--vvv----lll--
vvv
--vvv----bar--
```

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

**Exercise 6.4.** Rebuild the `find -type f` command (see exercise 4.18 on page 54) using only the features of the `bash` shell. That is your script should list the relative path to all files in all subdirectories and subsubdirs . . . of the current working dir. Some hints:

- Do not worry about the full task at first. Imagine your working directory is a particular directory, `resources` say. In this directory you will find other directories and of course files. Only deal with the files for now, i.e.: Write a `bash` function, which lists all files within a directory.
- The `for file in *; do`-loop is your friend here.
- Extend the above function such that it calls itself to process the immediate subdirectories as well. This strategy to solve this problem is called **recursive** processing.
- Now try to achieve the full goal. Use subshells to keep track of the current directory level you are in and be careful to print really the full path to a particular file like `find -type f` does it as well.

[2]i.e. by the means of other functions, which call functions, . . .

### 6.2.1 Good practice when using functions

A couple of helpful notes for writing functions, which are easy to understand and easy to use.

- 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 following two code snippets, which display some basic information of an `mtx` file. Decide for yourself what is more readable[3]

```
#!/bin/bash
# a good example

mtr_read_head() {
  #$1: file name of mtx file
  # echos the first content line (including the matrix size) to
    ↪stdout
  # returns 0 if all is well
  # returns 1 if an error occurred (file could not be read)

  # check we can read the file
  [ ! -r "$1" ] && return 1

  # get the data
  local DATA=$(< "$1" grep -v "%" | head -n1)

  # did we get any data?
  if [ "$DATA" ]; then
    echo "$DATA"
    return 0
  else
    return 1
  fi
```

[3]By the way: `6_functions_subshells/fun_bad.sh` contains an error. Good luck finding it.

```
}

gcut() {
  # this a more general version of cut
  # that can be tuned using the IFS
  #
  # $1: n -- the field to get from stdin
  # return 1 on any error

  local n=$1
  if ((n<1)); then
    return 1
  elif ((n==1)); then
    local FIELD BIN

    # read two fields and return
    # the first we care about
    read FIELD BIN
    echo "$FIELD"
  else
    local FIELD REST

    # discard the first field
    read FIELD REST

    # and call myself
    echo "$REST" | gcut $((n-1))
  fi
}

mtx_get_rows() {
  # get the number of rows in the matrix from an mtx file
  # echo the result to stdout
  # return 1 if there is an error

  local DATA

  # read the data and return when error
  DATA=$(mtr_read_head "$1") #|| return $?
  # parse the data -> row is the first field
  echo "$DATA" | gcut 1

  # implicit return of return code of gcut
}

mtx_get_cols() {
  # get the number of columns in the matrix file
  # return 1 on any error

  local DATA
  DATA=$(mtr_read_head "$1") || return $?
  echo "$DATA" | gcut 2  #cols on field 2
}
```

```bash

mtx_get_nonzero() {
  # get the number of nonzero entries in the matrix file
  # return 1 on any error

  local DATA
  DATA=$(mtr_read_head "$1") || return $?
  echo "$DATA" | gcut 3  #cols on field 2
}

mtx_get_comment() {
  mtx_fill_cache "$1" && echo "$__MTX_INFO_CACHE_COMMENT"
}

####################################
# the main script

if [ "$1" == "-h" -o "$1" == "--help" ];then
  echo "Script␣to␣display␣basic␣information␣in␣an␣mtx␣file"
  exit 0
fi

if [ ! -r "$1" ]; then
  echo "Please␣specify␣mtx␣file␣as␣first␣arg." >&2
  exit 1
fi

echo "No␣rows:␣␣␣␣␣$(mtx_get_rows␣"$1")"
echo "No␣cols:␣␣␣␣␣$(mtx_get_cols␣"$1")"
echo "No␣nonzero:␣␣$(mtx_get_nonzero␣"$1")"

exit 0
```

6_functions_subshells/fun_good.sh

```bash
#!/bin/bash
# a bad example

if [ "$1" == "-h" -o "$1" == "--help" ];then
  echo "Script␣to␣display␣basic␣information␣in␣an␣mtx␣file"
  exit 0
fi

foo() {
  echo $NONZERO
}

DATA=""

check2() {
  if [ -z "$DATA" ]; then
    echo "Can't␣read␣file" >&2
    return 1
  fi
  return 0
}

blubb() {
  echo $ROW
}

check1() {
  if [ ! -r "$1" ]; then
    echo "Can't␣read␣file" >&2
    return 1
  fi
  return 0
}

check1 "$1" || exit 1

fun1() {
  DATA=$(< "$1" grep -v "%" | head -n1)
}

fun1 "$1"
check2 || exit 1

reader() {
  echo $DATA | {
    read COL ROW NONZERO
  }
}

reader
echo -n "No␣rows:␣␣␣␣␣"; blubb

tester() {
  echo $COL
}
echo -n "No␣cols:␣␣␣␣␣"; tester
echo -n "No␣nonzero:␣␣"; foo

exit 0
```

6_functions_subshells/fun_bad.sh

**Exercise 6.5.** Take another look at your script from the second Project Gutenberg exercise (exercise 5.4 on page 66). Split the script into a few sensible functions. Some ideas:

- Have one function to parse read the tabular output of ex. 4.6 and compute the $\xi$ numbers. The results could be sent to *stdout* in another tabular form which shows the $\xi$ numbers and the file:

```
0.01␣pg74.txt
0.2␣pg345.txt
```

- One function to read the list produced above and print three recommended books to *stdout*
- The main body should just call the example 4.6 script and use the functions defined above to process what the ex-4.6-script yields.

**Exercise 6.6.** *(demo)* In this exercise we will try some abstract `bash` programming using functions. First take a look at the following code:

```
map() {
  COMMAND=$1  # read the command
  shift       # shift $1 away

  # now for all remaining arguments execute
  # the command with the argument:
  for val in $@; do
    $COMMAND $val
  done
}
```

6_functions_subshells/map.lib.sh

It defines a so-called mapping function, which 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:

```
map echo "some" "variables␣on␣the" "commandline"

cd ~/bash-course #replace by dir where you downloaded the git into
map head "resources/testfile" "resources/matrices/3.mtx"
```

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?

### 6.2.2 Overwriting commands

At the stage of execution the **bash** gives preference to user-defined functions over builtin commands or commands from the operating system. This implies that care must be taken when naming your functions, since these can “overwrite” commands[4], which may lead to very surprising results:

```
#!/bin/bash
test() {
  echo "Hi␣from␣the␣test␣function"
}
test 1 -gt 2 && echo "1␣is␣greater␣than␣2"
```

6_functions_subshells/overwrite_fail.sh

```
Hi␣from␣the␣test␣function
1␣is␣greater␣than␣2
```

Since commands within a function are of course subject to the same evaluation strategy by the **bash** as “free” commands in the script, accidental overwriting of commands can lead to very subtle infinite loops:

```
#!/bin/bash
C=0
[() { # overwrite the [ builtin

  # Increase and print a counter
  ((C++))
  echo $C

  # this gives an infinite loop:
  if [ $C -gt 100 ] ; then
    echo "never␣printed"
    exit 1
  fi
}

if [ "$VAR" ]; then
  echo "VAR␣is␣not␣empty"  #never reached
fi
```

6_functions_subshells/overwrite_loop.sh

```
1
2
3

...
```

In scripts it is best to avoid this for overwriting builtins or system commands, since it can make code very cumbersome and hard to understand. For customising your interactive **bash**, however, this can become very handy (see appendix B.1.1 on page 130).

[4] *Overwriting* is a concept from object-oriented programming where functions of the same name are called depending on the context of the call

Another very handy use case for this is to dynamically change the meaning of a function during the execution of a script. This works, since the `bash` only remembers the most recently defined body for a particular function name. A good example for using this is logging:

```
#!/bin/bash
# Default logging function
log() { echo "$@"; }

if [ "$1" == "--quiet" ]; then
  # Empty logging function:
  # Works since ":" is the idle command doing exactly nothing
  log() { :; }
fi

# Log something ... or not
log Hello and welcome to this script!
```

6_functions_subshells/overwrite_mostrecent.sh

Without "`--quiet`" the script prints

```
Hello␣and␣welcome␣to␣this␣script!
```

With "`--quiet`" all `log` calls are essentially ignored.

## 6.3 Cleanup routines

Using subshells it becomes easy to temporarily alter variables or the working directory and have these changes "automatically" changed back to the original — no matter where and how the subshell exited. Especially for larger scripts this helps to prevent many errors.

Sometimes it would be nice to be able to do more than that in case a script exits or gets interrupted. Consider for example the following program, where we need a temporary file to store some intermediate results:

```
#!/bin/bash
TMP=$(mktemp)  # create temporary file

# add some stuff to it
echo "data" >> "$TMP"

##
#  many lines of code
##

# and now we forgot about the teporary file
if [ "$CONDITION" != "true" ]; then
  exit 0
fi

##
```

```
#  many more lines of code
##

#cleanup
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.

```
#!/bin/bash
TMP=$(mktemp)  # create temporary file

# define the cleanup routine
cleanup() {
  echo cleanup called
  rm $TMP
}
# make cleanup be called WHENEVER the shell exits
trap cleanup EXIT

# add some stuff to it
echo "data" >> "$TMP"

##
#  many lines of code
##

# and now we forgot about the teporary file
if [ "$CONDITION" != "true" ]; then
  exit 0
fi

##
#  many more lines of code
##

#no need to do explicit cleanup
```

6_functions_subshells/cleanup_trap.sh

```
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. 6.6 on page 86). For this purpose the `bash` provides a feature called **sourcing**. Using the syntax

```
. otherscript
```

a file `otherscript` can be executed in the environment of the *current* shell, i.e. just like copying the full content of `otherscript` at precisely the location of the call. This implies of course that all variables and functions defined in `otherscript` are also available to the shell afterwards. An example:

```
testfunction() {
  echo "Hey I exist"
}
VAR=foo
```

6_functions_subshells/sourcing.lib.sh

```
#!/bin/bash

# Extend path such that the bash can find the script
# to be sourced.
PATH="$PATH:6_functions_subshells"
. sourcing.lib.sh  # lookup of sourcing.lib.sh performed using PATH

echo $VAR
testfunction
```

6_functions_subshells/sourcing.script.sh

```
foo
Hey I exist
```

In order to find `otherscript` the `bash` honours the environment variable `PATH`.[5] As the example suggests this way libraries defining common or important functionality may be stored in a particular library directory and used from many other scripts located in very different places by adding this library directory to the `PATH` environment variable.

On top of that there exists a dirty trick to make each script sourcable by default, such that functions or global values inside the script may be used by other scripts at a later point in time.

The trick relies on the fact that the `return` statement is only allowed in files, which are sourced, but *not* in scripts which are executed normally That way one can distinguish inside the script and separate function definitions and "global code" — to be executed in all cases — and code, which should only be touched if a script is not just sourced, but properly executed. For the script `fun_good.sh` presented in section 6.2.1 on page 82, we just add a

```
return 0 &> /dev/null
```

[5]See exercise 4.19 on page 57 for more details on the path lookup.

after the function definitions:

```
mtx_get_comment() {
  mtx_fill_cache "$1" && echo "$__MTX_INFO_CACHE_COMMENT"
}

#if we have been sourced this exits execution here:
# so by sourcing we can use gcut, mtx_get_rows, ...
return 0 &> /dev/null

####################################

if [ "$1" == "-h" -o "$1" == "--help" ];then
```

6_functions_subshells/source_sourcability.sh

**Exercise 6.7.** Make your script from exercise 6.6 on page 86 sourcable and amend the following script in order to get the functionality described in the comments:

```
#!/bin/bash

# do something in order to get the functions
# add and multiply from the exercise we had before

# add 4 and 5 and print result to stdout:
add 4 5

# multiply 6 and 7 and print result to stdout:
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[1]. We will now turn our attention to a few very handy programs, which are typically key in solving the tasks of everyday scripting, namely grep, sed and — in the next chapter 8 — awk.

All of these use so-called regular expressions, which are a key tool in the Unix word to find or describe strings in a text. We will introduce regular expressions in this chapter first in a general setting and then specifically in the context of grep and sed.

## 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 *regex*es:

```
[[ "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 be done with [[ ... ]] in exactly the same syntax. Just it offers a few extras as well.

Long story short: A simple bash command line like

```
$ [[ "string" =~ regex ]]; echo $?
```

[1]A list of things we left out can be found in appendix B.4 on page 131

will aid us with exploring regular expressions. It will print `0` whenever `string` is matched by `regex` and `1` otherwise.

**Example 7.1.** The regex `r.t` matches all lines which contain an `r` and two characters later an `t` as we will see in a second. So if we run

```
$ [[ "somer␣morer␣things" =~ r.t ]]; echo $?
```

we get

```
0
```

as expected, since the string matches at `morer things`. For

```
$ [[ "more␣other␣thing" =~ r.t ]]; echo $?
```

we get

```
0
```

as well because of `other thing`. On the other hand

```
$ [[ "more␣otherthing" =~ r.t ]]; echo $?
```

gives

```
1
```

It is important to note here that really the full string which is specified on the left is matched to the expression on the right.

One final note before we dive into the matter: The `[[` construct has the subtlety that it gives rise to really surprising and weird results if the `regex` itself is quoted using `"` as well. So always specify the regex unquoted on the rhs of the `=~` operator.

### 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`"[2].

Without further ado a non-exhaustive list of **regular expression operators**[3]:

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

[2]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

[3]More can be found e.g. in the `awk` manual [3]

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

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

7_regular_expressions/regex_anchor.sh

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

```
[[ $(echo -e "word\ntest") =~ test$ ]]; echo $?  #0=true
[[ $(echo -e "test\nword") =~ test$ ]]; 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.

```
[[ "o" =~ ^[oale]$ ]]; echo $?   #0=true
[[ "a" =~ ^[oale]$ ]]; echo $?   #0=true
[[ "oo" =~ ^[oale]$ ]]; echo $?  #1=false
[[ "\$" =~ ^[$]$ ]]; echo $?     #0=true
```

7_regular_expressions/regex_bracket.sh

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

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

```
[[ "o" =~ [^eulr] ]]; echo $?   #0=true
[[ "e" =~ [^eulr] ]]; echo $?   #1=false

#ATTENTION: this is not a cbe
[[ "a" =~ [o^ale] ]]; echo $?   #0=true
```

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 is **greedy**: It applies to the largest possible regexes on either side.

```
#gives true, since ^wo
[[ "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.

```
[[ "word" =~ ^(wo|rrd)$ ]]; echo $?   #1=false
```

7_regular_expressions/regex_grouping.sh

`*` The *preceding* regular expression operator (or operator group) should be repeated as many times as necessary to find a match, e.g. "`ico*`' matches "`ic`", "`ico`" or "`icooooo`", but not "`icco`". The "`*`" applies only to a single character by default. If you want it to apply to more than one preceeding character, you need to use a grouping statement `(...)`.

```
[[ "wo␣(rd" =~ wo* \( ]]; echo $?        #0=true
[[ "woo␣(rd" =~ wo* \( ]]; echo $?       #0=true
[[ "oo␣(rd" =~ wo* \( ]]; echo $?        #1=false
[[ "oo␣(rd" =~ (wo)* \( ]]; echo $?      #0=true
[[ "wowo␣(rd" =~ (wo)* \( ]]; echo $?    #0=true
```

7_regular_expressions/regex_star.sh

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

```
[[ "woo␣(rd" =~ wo+ \( ]]; echo $?       #0=true
[[ "oo␣(rd" =~ (wo)+ \( ]]; echo $?      #1=false
[[ "wo␣(rd" =~ (wo)+ \( ]]; echo $?      #0=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 generally try to match as much of the input string as possible.
- Regexes are case-sensitive.
- Unless `^` or `$` are specified, the matched substring may start and end anywhere.
- As soon as a single matching substring exists in the input string, the string is considered a match and the `[[` statement returns 0.

### 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]` | `[^abcdefgh ... xyz456789ABCDEFG]` |

| 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 |

Table 7.1: Some POSIX character classes

**Exercise 7.2.** 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**. See table 7.1 for some examples and [3] for more details. POSIX character classes can only be used within bracket expansions, e.g.

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

7_regular_expressions/regex_posixclass.sh

which gives the output

```
No␣match
```

and returns 1.

### 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 7.3.** 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 7.4.** Give regular expressions that satisfy the following

| | matches | does not match | chars |
|---|---|---|---|
| a) | `abbbc`, `abbc`, `abc`, `ac` | `aba` | 2 |
| b) | `abbbc`, `abbc`, `abc` | `bac`, `ab` | 3 |
| c) | `ac`, `abashc`, `a123c` | `cbluba`, `aefg` | 2 |
| d) | `␣␣qome`, `␣qol` , `qde` | `eqo`, `efeq` | 3 |
| 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 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.

```
# find lines containing foo!bar:
< 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. This can, however, lead to very nasty constructs like

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

where a lot of backslashes are needed.

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

```
#!/bin/bash

echo "Plain␣grep␣gives:"
<  resources/testfile grep ".[a-f]$"

echo "grep␣-o␣gives:"
<  resources/testfile grep -o ".[a-f]$"
```

7_regular_expressions/grep_only_matching.sh

gives

```
Plain grep gives:
some
data
some
date
grep -o gives:
me
ta
me
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 orgin for this behaviour is that `grep` actually implements two different kinds of regular expression languages. Once the so-called **basic regular expression** or BRE, which has a reduced feature set and is hence faster to process and the more feature-rich **extended regular expression** syntax or ERE. For our purposes it suffices to know that ERE is pretty much a superset of BRE[4] and that some of the operators we mentioned in the previous sections do not work in the BRE syntax. Since `grep` by default only uses BREs for performance reasons, we occasionally need the `-E` to switch to ERE-mode. Since using EREs really does have a performance impact, we should only use `-E` in cases where plain `grep` fails.

Without going into too much detail on the matter of the different regular expression dialects, we should note at this point, that BREs and EREs are not the only ones around. Most notably there also exist PCREs, **perl-compatible regular expressions** and for example the scripting language Python has its regular expression version, too. The

[4]See the `grep` manpage for details.

reasons for this are out of scope of this course, just note that in almost all cases, the syntax we present in this chapter will just work[5]. For more details consider the relevant manpages and help pages.

**Exercise 7.5.** 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, ...

**Exercise 7.6.** Take a look at the file `resources/digitfile`. This file contains both lines which contain only text as well as lines which contain numbers. The aim of this exercise is to design a regular expression which matches only those lines that contain numbers in proper scientific format, i.e. a number of the form

$$\underline{\texttt{sign}}\,\underline{\texttt{prefactor}}\,\texttt{e}\,\underline{\texttt{sign}}\,\underline{\texttt{exponent}}$$

e.g.

```
0.123e-4   0.45e1   -0.4e9
```

These numbers follow the rules

- The <u>`sign`</u> may be `+` or `-` or absent
- The <u>`prefactor`</u> has to be in the range between `0.` and `1.` In other words it will always contain a `.` as the second digit and the first character will always be a 0 or 1. The number of characters after the `.` may, however, vary.
- The <u>`exponent`</u> may be any integer number, i.e. it may not contain a `.`, but otherwise any number. It may have leading zeros.

In order to design the regular expression, proceed as follows:

- First design regexes to match the individual parts: <u>`sign`</u>, <u>`prefactor`</u> and <u>`exponent`</u>.
- Paste the indivdual expression parts together. Pay attention to which parts are required and which are optional.
- You will most certainly need EREs for some of them, so do not forget the `-E` flag for `grep`.
- `grep` has some issues if the regular expression itself starts with a `-` sign, because then it sometimes has trouble to distinguish its commandline options (which all start with a dash as well) from the actual regex. Depending on how you design your regexes you might run into this problem or not. In either case the `grep` flag `-e` is your friend here. Consult the manpage for more information.

[5] To make matters worse sometimes even the implementation matters: For example what precisely is understood as BREs and EREs in the GNU version of `grep` and the BSD version of `grep` is not fully identical.

- *(optional)* Introduce some fault tolerance:
  - Make your expression work if between prefactor and exponent one of the characters `E`, `D` or `d` is used instead.
  - Be less strict on the requirements of the prefactor. Allow prefactors outside of the range `0` to `1`.

**Exercise 7.7.** *(optional)* Here 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 132 if necessary.

- Use the final regular expression from the previous exercise including the fault tolerance as well as `grep -o` to extract all the values of the 3rd column from `resources/matrices/bcsstm01.mtx`.
- Use this and some standard unix tools 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:

```
echo "data␣stream" | sed 'sed_commands'
```

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

```
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 doube quotes can lead to an accumulation of escapes for both `sed` as well as the shell:

```
# compare
echo '\$a' | sed "s/\\\\\$a/bbb/g"

# with the single-quote example
echo '\$a' | sed 's/\\$a/bbb/g'
```

7_regular_expressions/sed_double_quotes.sh

Overview of basic sed commands[6]:

| | |
|---|---|
| `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`. |

[6]see e.g. the `sed` manual [4] for more details.

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

```
#!/bin/bash

{
  echo blub
  echo blbl
} | sed '/bl/a11111'

echo ------

{
  echo blub
  echo blbl
} | sed '/bl/i11111'
```

7_regular_expressions/sed_insertion.sh

```
blub
11111
blbl
11111
------
11111
blub
11111
blbl
```

`/regex/d` Delete matching lines.

```
#!/bin/bash
{
  echo line1
  echo line2
  echo line3
} | sed '/2$/d'
```

7_regular_expressions/sed_delete.sh

```
line1
line3
```

`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`.

```
#!/bin/bash

generator() {
  echo "line1"
  echo "␣␣␣␣␣␣line␣␣2␣␣"
  echo "LiNE3"
  echo
}

generator | sed 's/in/blablabla/'
echo "-----"
generator | sed 's/i.*[1-3]/...&.../'
echo "-----"

# a very common sequence to normalise input
generator | sed '
  s/[[:space:]][[:space:]]*/ /g
  s/^[[:space:]]//
  s/[[:space:]]$//
  /^$/d
'
```

7_regular_expressions/sed_substitute.sh

```
lblablablae1
␣␣␣␣␣␣lblablablae␣␣2
LiNE3

-----
l...ine1...
␣␣␣␣␣␣l...ine␣␣2...
L...iNE3...

-----
line1
line␣2
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 `\cregexc`, where `c` is an arbitrary character.
- For the command `s`: `scregexctextc` is equivalent to `s/regex/text/`.

```bash
#!/bin/bash
VAR="/some"
echo "/some/crazy/some/path" | sed "s#$VAR#/mORe#g"
echo "--"
echo "/some/crazy/path" | sed "\#crazy#d"
echo "--"
```

7_regular_expressions/sed_altmatch.sh

```
/mORe/crazy/mORe/path
--
--
```

**Exercise 7.8.** *(demo)* 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. Be careful, however, to not remove the "`-`" on compound surnames.
- Replace all `,` by `\n`, the escape sequence for a line break.
- Do cleanup: Remove unnecessary leading or tailing 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.

This chapter really only serves as a short introduction. Further information can be found in the *Introduction to `awk` programming* course [5, 6], which was taught in 2016 specifically as an addendum to this course. Another very noteworthy resource is the `gawk` manual "GAWK: Effective AWK programming" [3].

## 8.1 Structure of an `awk` program

All input given to an `awk` program is automatically split up into larger chunks called **record**s. Typically this is equivalent to a single line of the input data. Each record is then further split into smaller chunks called **field**s, which is usually just a single word. In other words records are separated by `<newline>` and fields by any character from `[:space:]`.

`awk` programs are a list of instructions like

```
condition { action }
condition { action }
...
```

During execution `awk` goes from record to record and checks for each of the `condition`s whether they are satisfied. If this is the casse the corresponding `action` is executed. Each pair of `condition` and `action` is called a **rule**. Rules are always processed top to bottom and the `action` is immediately executed if the corresponding `condition` is satisfied.

Both the `condition` as well as the action block `{ action }` may be missing from an `awk` rule. If the condition is missing, the `action` is executed for each input record. If the action block is missing the *default* action is executed, which is just printing the full record (i.e. line of text) to *stdout*.

Similar to the shell the `#` starts a comment in `awk` programs and `<newline>` and "`;`" may be 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

```
  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:

```
awk '
  ...
  awk_source
  ...
'
```

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.** To give you an example for what we discussed in this section, just a very simple shell script to pipe some data through an inline `awk` program[1]. The code makes use of the `awk` action command `print` (see 8.8 on page 120 below for details), which is essentially `awk`'s version of `echo`.

```
#!/bin/bash
{
  echo "awk input"
} | awk '
  # missing condition => always done
  { print "Hi user. This is what you gave me:" }

  # condition which is true and no action
  # => default print action
  1 == 1

  # another message which is always printed
  { print "Thank you" }
'
```

8_awk/basic_example.sh

[1]I will use syntax highlighting adapted for `awk` code for all example code in this chapter.

```
Hi user. This is what you gave me:
awk input
Thank you
```

So far so easy. We give `awk` some input. It runs through each rule and since all conditions (including the trivial `1 == 1` are satisfied, it executes all the actions top to bottom. For the second rule, the default action, i.e. printing the input, is executed, since no other action is given.

Now what happens if we give the `awk` snippet two lines of input?

```
#!/bin/bash
{
  echo "awk input 1"
  echo "awk input 2"
} | awk '
  # missing condition => always done
  { print "Hi user. This is what you gave me:" }

  # condition which is true and no action
  # => default print action
  1 == 1

  # another message which is always printed
  { print "Thank you" }
'
```

8_awk/less_basic_example.sh

```
Hi user. This is what you gave me:
awk input 1
Thank you
Hi user. This is what you gave me:
awk input 2
Thank you
```

This result might seem surprising at first, but can be easily explained by the fact that `awk` executes the full program *for each* record, i.e. each line of input!

Even though most people find this speciality of `awk` a little odd at first, the great power of `awk` also truely originates from this very fact. A good reason to look into this a little more in the next section.

## 8.3 awk programs have an implicit loop

As we said in section 8.1 on page 104, 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 examples:

```
#!/bin/bash

# function generating the output
output() {
  echo "line␣1"
}

echo "Program1:"
# a small awk program which just prints the output
# line-by-line as it is
# we use a condition which is always true and the
# default action here (implicit print of the whole
# record, i.e. line)
output | awk '1==1'

echo
echo "Program2:"
# a program with two rules:
# one which does the default printing
# and a second one which prints an extra line
# unconditionally
output | awk '
  1==1 #default print action
  { print "some␣stuff" }
'
```

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

```
Program1:
line␣1

Program2:
line␣1
some␣stuff
```

We note, 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, since of course both actions are associated to conditions which are either trivially true or not present (and hence implicitly true).

Now let us try the same thing but pass two or three lines of input

```
#!/bin/bash

# function generating the output
output() {
  echo "line␣1"
  echo "line␣2"
}
```

```

echo "Program1:"
output | awk '1==1'

echo
echo "Program2:"
output | awk '
  1==1 #default print action
  { print "some␣stuff" }
'
```

8_awk/each_line_example2.sh

```
Program1:
line␣1
line␣2

Program2:
line␣1
some␣stuff
line␣2
some␣stuff
```

and

```
#!/bin/bash

# function generating the output
output() {
  echo "line␣1"
  echo "line␣2"
  echo "line␣3"
}

echo "Program1:"
output | awk '1==1'

echo
echo "Program2:"
output | awk '
  1==1 #default print action
  { print "some␣stuff" }
'
```

8_awk/each_line_example3.sh

```
Program1:
line␣1
line␣2
line␣3

Program2:
line␣1
some␣stuff
```

```
line␣2
some␣stuff
line␣3
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 counter-intuitive 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 "`;`"[2]. Other line breaks are (usually) ignored. This means that e.g.[3]

```
# the echo is just here to make awk do anything -> see footnote
echo | awk '
  {
    print "some␣message"
    print "other␣message"
  }
  {
    print "third␣message"
  }
'
```

and

```
echo | awk '{ print "some␣message"; print "other␣message" }
  { print "third␣message" }'
```

and

```
echo | awk '{ print "some␣message"; print "other␣message" }; { ↙
    ↪print "third␣message" }'
```

are all equivalent.

[2]This is not entirely correct, see section 1.6 of the `gawk` manual [3] for details

[3]We already said that the `awk` rules are 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*.

## 8.5 Strings in awk

Strings in `awk` all have to be enclosed by double quotes, e.g.[4]

```
# inside awk action block -> see footnote
print "This␣is␣a␣valid␣string"
```

Multiple strings may be concatenated, just by leaving white space between them

```
#!/bin/bash
echo | awk '{ print "string1"   "␣"   "string2" }'
```

8_awk/vars_stringconcat.sh

```
string1␣string2
```

`awk` per default honours special sequences like "`\t`"(Tab) and "`\n`"(Newline) if used within strings:

```
#!/bin/bash
echo | awk '
  { print "test\ttest2\ntest3" }
'
```

8_awk/vars_stringspecial.sh

```
test    test2
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.

  ```
  var="value"
  # or
  var =    "value"
  ```

- Such a statement counts as an action, so we need multiple of these to be separated by a line break or "`;`":

  ```
    varone="1"; vartwo="2"
  ```

- In order to use the value of a variable no `$` is required:

  ```
  print var        # => will print "value"
  ```

[4]For some examples in this chapter the enclosing script is left out for simplicity. 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.

- `awk` is aware of floating point numbers and can deal with them properly

```
#!/bin/bash
echo | awk '{
  var="4.5"
  var2=2.4
  print var "+" var2 "=" var+var2
}'
```

8_awk/vars_fpaware.sh

```
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.

```
#!/bin/bash
echo | awk '{
  floatvar=3.2
  stringvar="abra"      #cannot be converted to number
  floatstring="1e-2"    #can be converted to number

  # calculation
  res1 = floatvar+floatstring
  res2 = floatvar + stringvar

  print res1 "␣" res2
}'
```

8_awk/vars_fpconvert.sh

```
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)

```
#!/bin/bash
echo | awk '
  { N=4; A="blub" }
  { print N }
  { print "String␣" A "␣has␣the␣length␣" length(A) }
'
```

8_awk/vars_global.sh

```
4
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 60. This includes the `C`-like convention of 0 being “false” and 1 being “true”:

```
#!/bin/bash
echo | awk '{
  v=3
  u=4

  print v "-" u "=" v-u

  v+=2
  u*=0.5

  print v "%" u "=" v%u


  # exponentiation is ^
  print v "^" u "=" v^u

  # need to enforce that comparison operatiors are
  # executed before concatenation of the resulting
  # strings. Not quite sure why.
  print v "==" u ":␣" (v==u)
  print v "!=" u ":␣" (v!=u)
  print v "!=" u "||" v "==" u ":␣" (v!=u||v==u)
  print v "!=" u "&&" v "==" u ":␣" (v!=u&&v==u)
}'
```

8_awk/vars_arithlogic.sh

```
3-4=-1
5%2=1
5^2=25
0
1
1
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!

```
#!/bin/bash
echo -e "some␣7␣words\tfor␣awk␣to␣process" | awk '
  {
    print "arithmetic:␣" 2*$2
    print $4 "␣" $1
  }

  {
    print "You␣gave␣me:␣␣" $0
  }
  '
```

8_awk/vars_fields.sh

```
arithmetic:␣14
for␣some
You␣gave␣me:␣␣some␣7␣words␣␣␣for␣awk␣to␣process
```

This lookup also works indirectly:

```
#!/bin/bash
echo -e "some␣words␣for\tawk␣to␣process" | awk '
  {
    v=5
    print $v
  }'
```

8_awk/vars_fields_indirect.sh

```
to
```

**NF** contains the number of fields in the current record. So the last field in a record can always be examined using `$NF`

```
#!/bin/bash
echo "some␣words␣for␣awk␣to␣process" | awk '
  {
    print "There␣are␣" NF "␣fields␣and␣the␣last␣is␣" $NF
  }'
```

8_awk/vars_fields_nf.sh

```
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 122.

**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.[5]

### 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:

```
#!/bin/bash

# define a shell variable:
A=laber

echo | awk '
  # define an awk variable and print it:
  { N=4; print N }

  # print something using the non-present shell variable A:
  { print "We␣have␣no␣clue␣about␣string␣A:␣\"" A "\"" }
'

# show that the shell knows A, but has no clue about N:
echo --$A--$N--
```

8_awk/awk_vs_shell_vars.sh

```
4
We␣have␣no␣clue␣about␣string␣A:␣""
--laber----
```

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 75,

[5]Be aware that some `awk` implementations like `mawk` furthermore have no support for changing `RS`.

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:

```
#!/bin/bash

# some data we have available on the shell
VAR="3.4"
OTHER="6.7"

# do calculation in awk and return packed data
RES=$(echo "$VAR␣$OTHER" | awk '{
  sum=$1 + $2
  product=$1*$2
  print sum "+" product
}')

# unpack the data on the shell again:
SUM=$(echo "$RES" | cut -f1 -d+)
PRODUCT=$(echo "$RES" | cut -f2 -d+)

# use it in an echo
echo "The␣sum␣is:␣$SUM"
echo "The␣product␣is:␣$PRODUCT"
```

8_awk/awk_vs_shell_getdata.sh

```
The␣sum␣is:␣10.1
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:

```
  awk -v "name=value" ' awk_source '
```

This is very useful in order to transfer `bash` variables to the `awk` program, e.g.

```
#!/bin/bash

VAR="abc"
NUMBER="5.4"
OTHER="3"

# ...

echo "data␣1␣2␣3" | awk -v "var=$VAR" -v "num=$NUMBER" -v ↙
    ↪"other=$OTHER" '
  {
    print $1 "␣and␣" var

    sum = $2 + $3
    print num*sum
    print $4 "␣" other
  }
'
```

8_awk/vars_from_shell.sh

```
data␣and␣abc
16.2
3␣3
```

**Exercise 8.3.** Take another look at your script from exercise 6.6 on page 86. 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.

```
#!/bin/bash
VAR="print"
echo "some␣test␣data␣5.3" | awk -v "var=$VAR" '
  var == "print" { print $2 }
  var == "noprint" { print "no" }
  $4 > 2  { print "fulfilled" }
'
```

8_awk/cond_comp.sh

```
test
fulfilled
```

- Regular expressions matching the current record

```
#!/bin/bash

{
  echo "not␣important"
  echo "data␣begin:␣1␣2␣3"
  echo "nodata:␣itanei␣taen␣end"
  echo "other␣things"
} | awk '
  # start printing if line starts with data begin
  /^data begin/ { pr=1 }

  # print current line
  pr == 1

  # stop printing if end encountered
  /end$/ { pr=0 }
'
```

8_awk/cond_regex_record.sh

```
data␣begin:␣1␣2␣3
nodata:␣itanei␣taen␣end
```

- Regular expressions matching the content of a variable (including `$0`, `$1`, ...)

```
#!/bin/bash
VAR="15"

echo "data␣data␣data" | awk -v "var=$VAR" '
  # executed if var is a single-digit number:
  var ~ /^[0-9]$/ {
    print "var␣is␣a␣single␣digit␣number"
  }

  # executed if var is NOT a single-digit
  var !~ /^[0-9]$/ {
    print "var␣is␣not␣a␣single␣digit"
  }

  $2 ~ /^.a/ {
    print "2nd␣field␣has␣a␣as␣second␣char"
  }
'
```

8_awk/cond_regex_var.sh

```
var␣is␣not␣a␣single␣digit
2nd␣field␣has␣a␣as␣second␣char
```

- Combination of conditions using logical AND (`&&`) or OR (`||`)

```
#!/bin/bash
VAR="15"

echo "data␣data␣data" | awk -v "var=$VAR" '
  var !~ /^[0-9]$/ && $2 == "data" {
    print "Both␣are␣true"
  }
'
```

8_awk/cond_combination.sh

```
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.

```
#!/bin/bash

{
  echo "data␣data␣data"
  echo "data␣data␣data"
  echo "data␣data␣data"
} | awk '
  BEGIN { number=0 } # optional: all uninitialised
                     # variables are 0
  { number += NF }
  END { print number }
'
```

8_awk/cond_begin_end.sh

```
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`:

```
#!/bin/bash
{
  echo "not␣important"
  echo "data␣begin"
  echo "1␣2␣3"
  echo "end"
  echo "other␣things"
} | awk '
  BEGIN {
    # initialise pr as 0
    # printing should only be done if pr==1
    pr=0
  }

  # start printing if line starts with data begin
  /^data begin/ { pr=1 }

  # stop printing if end encountered
  /end$/ { pr=0 }

  # print first two fields of current line
  # error here
  pr = 1 { print $1 "␣" $2 }
'
```

8_awk/cond_assign_error.sh

```
not␣important
data␣begin
1␣2
end
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

```
#!/bin/bask
awk '
  #your code here
'
```

- 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.** *(optional)* 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

```
␣␣Starting␣Davidson␣...
```

- A nice table is printed afterwards with the iteration index given in the first column

- The procedure is concluded with the lines

```
-----------------------------------------------------
␣␣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.

```bash
#!/bin/bash

{
  echo record1 word2
  echo record2 word4
  echo record3 word6
} | awk '
  BEGIN { c=0 }
  { c++ }
  { print c ":␣first␣rule" }
  /4$/ { next; print c "␣" $1 }
  { print c ":␣"  $2 }
'
```

8_awk/action_next.sh

```
1:␣first␣rule
1:␣word2
2:␣first␣rule
3:␣first␣rule
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.

```
#!/bin/bash

{
  echo record1 word2
  echo record2 word4
  echo record3 word6
} | awk '
  BEGIN { c=0 }
  { c++ }
  { print c ":␣first␣rule" }
  /4$/ { exit 42; print c "␣" $1 }
  { print c ":␣"  $2 }
  END { print "quitting␣..." }
'
echo "return␣code:␣$?"
```

8_awk/action_exit.sh

```
1:␣first␣rule
1:␣word2
2:␣first␣rule
quitting␣...
return␣code:␣42
```

**print** Print the strings supplied as arguments, followed by a newline character[6]. 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 [7]

```
#!/bin/bash
{
  echo 1 2 3 4
  echo 5 6 7 8
} | awk '
  $1 < 4 { printf $3 "␣" }
  $1 > 4 { printf $3 }
'
```

8_awk/action_printf.sh

```
3␣7
```

[6]Can be changed. See section 5.1.1 of the awk course notes [5] for details

[7]`printf` is much more powerful and allows fine-grained control of priting: See section 5.2 of [5] for more 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 34 for `bash`. We don't want to go through all of these here [8], just note that conditional branching can also be achieved inside an action block using the `if` control structure:

```
  if (condition) {
    action_commands
  } else {
    action_commands
  }
```

where `condition` may be any of the expressions discussed in section 8.7 on page 116. 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:

```
#!/bin/bash

# here we use DOUBLE quotes to have the shell
# insert the search pattern where awk expects it
awk "/$1/"
```

8_awk/ex_grep.sh

**Example 8.7.** Process some data from the `/etc/passwd`, where "`:`" or `,` are the field separators

```
#!/bin/bash
< /etc/passwd awk -v "user=$USER" '
  # set field separator to be : or , or many of these chars
  BEGIN {FS="[:,]+" }

  # found the entry for the current user?
  $1 == user {
    # print some info:
    print "Your username:        " $1
    print "Your uid:             " $3
    print "Your full name:       " $5
    print "Your home:            " $6
    print "Your default shell:   " $7
  }
'
```

8_awk/ex_passwd.sh

[8]See section 6.2. of the awk course notes [5] 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.[9]

```
#!/bin/bash
awk '
  # change the record separator to anything
  # which is not an alphanumeric (we consider
  # a different word to start at each alphnum-
  # eric character)
  BEGIN { RS="[^[:alnum:]]+" }
  # now each word is a separate record

  $0 == prev { print prev; ret=1; next }
  { prev = $0 }
  END { exit ret }
'
```

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.** *(demo)* 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.

[9] 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

```
␣␣Excited␣state␣1␣(singlet,␣A")␣␣␣␣␣␣␣␣␣␣␣␣␣␣[converged]
␣␣------------------------------------------------------
␣␣Term␣symbol:␣␣1␣(1)␣A"␣␣␣␣␣␣␣␣␣␣␣␣␣␣R^2␣=␣␣7.77227e-11

␣␣Total␣energy:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-7502.1159223236␣a.u.
␣␣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. The references in brackets point to relevant chapters of the *Introduction to awk programming* course notes [5] as well as the `gawk` manual [3] where more information about these topics can be found.

- Formatted printing ([5, chapter 5], [3, §5.5]): Controlling the precision of floats printed
- Control structures and statements ([5, chapter 6], [3, §7.4]) in `awk`: Loops, `case`, ...
- `awk` arrays ([5, chapter 7], [3, §8])
- `awk` string manipulation functions ([5, §8.1.2], [3, §9.1.3]): Substitutions, substrings, sorting
- Writing custom `awk` functions ([5, §8.2], [3, §9.2])
- Reading records with fixed field length ([5, §4.3], [3, §4.6]): Fields separated by the number of characters, not a regex.
- Reading or writing multiple files ([5, §5.3], [3, §4.9])
- Executing shell commands from within `awk` programs ([3, §4.9])
- Creating `awk` code libraries ([3, chapter 10])
- Arbitrary precision arithmetic using `awk` ([3, chapter 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 manyfold 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 small things 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

```
cat file | program
```

because of input redirection:

```
< file program
```

### 9.1.2 Useless use of `ls *`

We already said that

```
for file in $(ls *); do
  program "$file"
  # or worse without the quotes:
  program $file
done
```

is a bad idea because of the word-splitting that happens after command substitution. The better alternative is

```
for file in *; do
  program "$file"
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

```
RESULT=$(< file some_program)

# check if we got something
if [ "$RESULT" ];then
  do_sth_else
fi
```

we can just write

```
if <file some_program;then
  do_sth_else
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

```
grep word1 | grep word2 | grep word3
```

where the command

```
grep "word1.*word2.*word3"
```

is both more precise and faster, too.

Also `grep` already has numerous builtin flags such that e.g.

```
grep word | wc -l
```

are unnecessary, use e.g.

```
grep -c word
```

instead.

### 9.1.5 When `grep` is not enough . . .

. . . then do not use it!

```
grep regex | awk '{commands}'
```

can be replaced by

```
awk '/regex/ {commands}'
```

and similarly

```
grep regex | sed 's/word1/word2/'
```

can be replaced by

```
sed '/regex/s/word1/word2/'
```

### 9.1.6 testing for the exit code

It feels awkward to see

```
program
if [ "$?" != "0" ]; then
  echo "big␣PHAT␣error" >&2
fi
```

where

```
if ! program; then
  echo "big␣PHAT␣error" >&2
fi
```

is much nicer to read and feels more natural, too.

# Appendix A

# Setup for the course

This appendix summarises the required setup for working on the exercises and running the example scripts.

## A.1 Installing the required programs

All exercises and example scripts should run without any problem on all LinuX systems that have a recent `bash`, `sed` and the GNU `awk` implementation (`gawk`) installed. On non-Linux operating systems like Mac OS X it may still happen, that examples give different output or produce errors, due to subtle differences in the precise interface of the Unix utility programs.

### A.1.1 Debian / Ubuntu / Linux Mint

```
# Install as root:
apt-get install bash sed gawk git bsdutils findutils coreutils
```

### A.1.2 Mac OS X

For example using homebrew[1]

```
brew install bash gnu-sed gawk git findutils coreutils
```

[1] `https://brew.sh`

## A.2 Files for the examples and exercises

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.

# 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 [7] for more details.

- `bash` arrays
- Brace expansion
- Tilde expansion
- Coprocesses

# Appendix C

# Supplementary information

## C.1 The `mtx` file format

The main idea of the `mtx` file format is to be able to store matrix data in a plain text file without storing those matrix entries which are zero. This is achieved by only storing a selection of the matrix components and defaulting all other component values to 0.

The `mtx` files we use in this course[1] 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 entries, which are explicitly set in the file. We will refer to this number as $d$.
- All following lines — the explicitly provided entries — have the structure
  - Row index (starting at 1, i.e. 1-based)
  - Column index (1-based)
  - Value

  where the individual columns are again separated by one or more `<space>` or `<tab>` chars. The number of lines in this latter block and the number $d$ provided on the first non-comment line have to agree in a valid `mtx` file.

All matrix components, which are not listed in the latter block, default to a value 0.

[1]We will only use a subset of the full format, which can be found under `http://math.nist.gov/MatrixMarket/formats.html#mtx`

Some examples

- Consider the file

```
 1 %%MatrixMarket matrix coordinate real symmetric
 2 3 3 9
 3 1 1 1
 4 1 2 1
 5 1 3 1
 6 2 1 2
 7 2 2 2
 8 2 3 2
 9 3 1 3
10 3 2 3
11 3 3 3
```

resources/matrices/3.mtx

The first line is a comment line, which we can ignore. The second line tells us that the matrix represented is a $3 \times 3$ matrix and that all nine entries are provided in the Matrix Market file. Lines 3 to 11 then list the values. Overall this file represents the matrix

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}.$$

- The file

```
%%MatrixMarket matrix coordinate real
3 3 9
1 1 1
1 2 0
1 3 0
2 1 0
2 2 2
2 3 0
3 1 0
3 2 0
3 3 3
```

describes a $3 \times 3$ matrix as well, namely the diagonal matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}.$$

If we want to avoid storing the zeros, we can use the equally valid `mtx` file

```
%%MatrixMarket matrix coordinate real
3 3 3
1 1 1
2 2 2
3 3 3
```

Notice, how the last value in the first non-comment line has changed as well.

# 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] Michael F. Herbst. Introduction to awk programming 2016, August 2016. URL `https://doi.org/10.5281/zenodo.1038522`.

[6] Michael F. Herbst. Introduction to awk programming 2016 course website, August 2016. URL `https://michael-herbst.com/teaching/introduction-to-awk-programming-2016/`.

[7] Mendel Cooper. Advanced bash-scripting guide, March 2014. URL `http://www.tldp.org/LDP/abs/html/`.

# List of Commands

| | |
|---|---|
| `apropos` | Search in manpage summaries for keyword |
| `cat` | Concatenate one or many files together |
| `cd` | Change the current working directory |
| `chmod` | Change file or directory permissions (see section 1.3 on page 7) |
| `cut` | Extract columns from input |
| `echo` | Print something to output |
| `grep` | Filter input by pattern |
| `help` | Access help for `bash` builtin commands |
| `info` | Access the Texinfo manual for commands |
| `less` | View input or a file in a convenient way |
| `ls` | List the content of the current working directory |
| `man` | Open manual page for a command |
| `mkdir` | Create a directory |
| `pwd` | Print the current working directory |
| `return` | Quit processing a function (section 6.2 on page 79) or a sourced script (section 6.4 on page 90). |
| `rmdir` | Delete empty folders |
| `rm` | Delete files |
| `sort` | Sort input according to some parameters |
| `tac` | Concatenate files and print lines in reverse order |
| `tee` | Write input to file and output |
| `touch` | Change the modification time or create a file |

| | |
|---|---|
| `uniq` | Take a sorted input and discard double lines |
| `wc` | Count characters, lines or words on input |
| `whatis` | Print a short summary describing a command |