Advanced **bash** scripting

(block course)

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

Interdisziplinäres Zentrum für wissenschaftliches Rechnen
Ruprecht-Karls-Universität Heidelberg

24^{th} – 28^{th} August 2015

Contents

List of Tables

Course description

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

This idea structures the whole course: In the first part we will revisit some basic concepts of a Unix-like operating system and review the set of Unix 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 (e.g. from files) using so-called regular expressions and programs like `awk`, `sed` or `grep`.

Learning targets and objectives

After the course you will be able to

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

Prerequisites

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

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

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

Compatibility of the exercises

All exercises have been tested on Debian 7 "Jessie" with the relevant packages installed. Everything *should* work on other Unix-like operating systems as well, but I cannot guarantee it. Especially Mac OS X can be rather different in some cases, which is why generally recommend using the computer cluster instead.

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 `http://blog.mfhs.eu/teaching/advanced-bash-scripting-2015/`. If you use any part of my work, please include a reference to this URL along with my name and email address.

Script examples

All example scripts in the repository (see appendix A on page 104) are published under the CC0 1.0 Universal Licence. See the file `LICENCE` 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 center 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 or X-like OSes, which derive from the traditional AT&T UNIX. Two of the most important modern OSes, Mac OS X and GNU/Linux, are included in this family. In other words: Unix' importance in academia and business has not changed very much over the years.

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

Write programs that do one thing and do it well.

For the Unix-like OSes this means that in theory

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

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

  - programs ("shell scripts") that combine the utilities to achieve a bigger task

- The OS is extremely modular:
  - All programs have a well-defined interface
  - It is easy to swap one program for a modified/enhanced version without breaking the rest of the OS

- The OS is standardised:
  - The functionality of the programs is (almost) identical for all OSes of the Unix-family.

1.1.1 Impact for scripting

On such a platform scripting becomes very helpful since

- all important functionality is available in the OS-provided utilities. So very little actual code has to be written to glue the utilities together.
- the utilities are not too specific for a particular job and can therefore be used flexibly throughout the script.
- documentation of their interfaces (commandline arguments) is available.

⇒ If one changes from one Unix-like OS to another or from one version of the OS to the next, no change in the functionality of the derived script is to be expected.

⇒ Scripts become reusable and portable.

1.2 The Unix utilities

Now let us briefly review some of the most important utility programs on a modern Unix-like OS. This list is not at all complete and in fact we will add more and more utilities to our toolbox during the course. See page 109 for a full list of commands introduced in this course.

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

1.2.1 Accessing files or directories

`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 columns from input, options

- `-d` delimiter: Character to use for the split
- `-f` fields: Which fields(columns) to output

`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

Example 1.1. In this example we will assume that the current working directory is the top level of the git repository [2]. 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.

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

```
< 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[3] books pg74 and pg76 that contain the word "hunger". One could run those two commands one after another

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

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

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

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

```
< 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 e.g.

```
< resources/matrices/3.mtx sort | uniq
```

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

```
< resources/matrices/3.mtx sort -u
```

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

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

```
sort -u resources/matrices/3.mtx
```

In some cases the latter command tends to perform somewhat better. Nevertheless I personally prefer the version `< resources/matrices/3.mtx sort -u` since this has a very suggestive syntax: The data flows from the producers (`< file`) on the RHS to the consumers on the LHS and on the way passes through all commands.

1.2.4 Other

`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 out how the commands `tail` and `head` work we could use the manpage

```
man tail
man head
```

The same works with man itself, try 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 19).

1.2.5 Exercises

Exercise 1.6. Exploring the `man` program: Read the manpage of `man`. Find out how

- one can get the manpage in a different language (i.e. German instead of English)
- the manpages are structured:
  - How many sections are there?
  - Which one is the most important for us?
- one can enforce an article to be from an appropriate section

Exercise 1.7. A first look at Project Gutenberg books:

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

Exercise 1.8. Looking at some matrices:

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

- Provide a sequence of commands for each of the columns in `3.mtx` that prints *how many distinct* values there are in this column. Ignore the comment line. Look at the file yourself and compare the values.
- Provide a sequence of commands for each of the columns in `3.mtx` that prints *the smallest* value in the third column of `3.mtx`. Again make your commands ignore the first comment line.
- Run the same sequence of commands as in 5. on `resources/matrices/lund\_b.mtx`. The result should surprise you. What goes wrong here?
- Do the same thing with `resources/matrices/bcsstm01.mtx`. Again be very careful and check the result properly. You need the right options for `sort` for this to succeed.
- Another tool that can be used to print certain columns in files is `awk`. The syntax is `awk '{print $n}'` to print the `n`th column. Use it instead of `cut` for the file `lund\_b.mtx`. How does it perform?

1.3 The Unix file and permission system

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

1.3.1 What are files?

- Convenience feature for programmers or users of the computer
- File: Virtual chunk of data.
- File path: Virtual location where user expects the file.
- File System: Provides lookup feature to translate file path to hard drive location
- Lookup mechanism incorporates extra information about the file:
  - Owner (Person who created the file)
  - Group (Group of people file is attributed to)
  - Permissions for file access
  - Time when time was created/accessed/modified
- All this information can be obtained using the `ls -l` command
- Some files are "special", e.g.
  - soft links: Files that point to a different file path
  - ⇒ OS performs look-up at the other file path
  - hard links: Duplicated entries in the lookup mechanism
  - ⇒ Two paths point to the same hard drive location

1.3.2 Unix paths

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

- Entities on the path are separated by "`/`"
- The last entity may be a file or directory, all the others are directories[4]
- *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

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

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

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

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

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

```
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` 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 and redirects.

2.1 Historic overview

2.1.1 What is a shell?

Back in the days:

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

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

Nowadays:

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

2.1.2 The Bourne-again shell

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

```
. /etc/bash_completion
```

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

2.2.2 Accessing the command history

Consider a sequence of commands

```
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, we could just type

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

to achieve the same thing as above.

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

- Now 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.

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

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

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

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

2.3 Redirecting command input/output

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

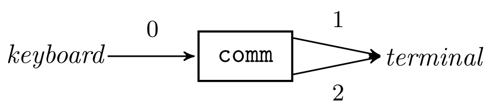
- *stdin* or file descriptor (fd) 0: The command reads all input from here
- *stdout* or fd 1: All normal output is printed here
- *stderr* or fd 2: All output concerning errors is printed here

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

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

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

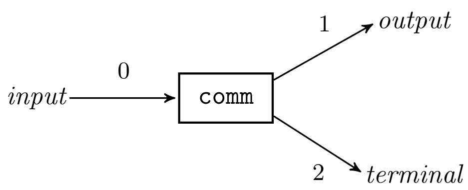
By default *stdin* is connected to the keyboard and both *stdout* and *stderr* are connected to the terminal. Running a `comm` in the shell hence gives a "redirection diagram" like



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

```
< 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

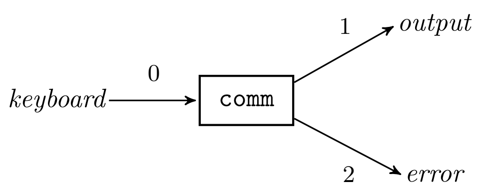
```
< input comm >>output
```

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`.

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



syntax	Comment
`>`	print *stdout* to file
`>>`	append *stdout* to file
`2>`	print *stderr* to file
`2>>`	append *stderr* to file
`\&>`	print *stdout* and *stderr* to file
`\&>>`	append *stdout* and *stderr* to file

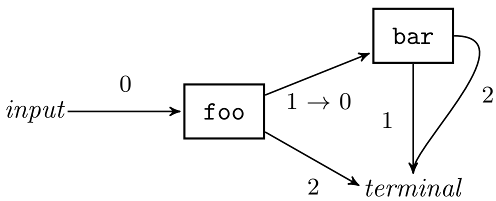
Table 2.2: Summary of the output redirectors of the `bash` shell. The versions with a single `>` always substitute the content of the file entirely, whereas the `>>` redirectors append to a file.

syntax	Comment
`|`	connect *stdout* $\rightarrow$ *stdin*
`|\&`	connect *stdout* and *stderr* $\rightarrow$ *stdin*

Table 2.3: Summary of the types of pipes

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

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



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

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

```
 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 redirection diagrams

```
ls |& ls | grep blubber | 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 keyword &> log.summary
  ```

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

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

Exercise 2.4.

- Create a file called `in` and write some random text to it.
- Run `< in cat > out`. What happens?
- Run `< in cat > in`. What happens here?
- Draw a redirection diagram for running plain `cat`. How can you explain that the terminal seems to "hang" if just `cat` is executed on the commandline.
  (Hint: Run `cat`, type something to the terminal and press Enter)

2.4 The exit status of a command

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

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

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

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

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

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

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

- 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 exectued 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 2.5. Go to the directory `resources/directories`. Explain the output of the following commands

- Run `cd 3/2 || cd 4/2 && cd ../4 || cd ../3 && cat file`
- Suggest the places at which we need to insert a `2>/dev/null` in order to suppress the errors from `cd`. Try to insert as little code as possible
- Run `mkdir -p 3/2; cd 3/2 || cd 4/2 && cd ../4 || cd ../2 && cat file; rmdir 3/2; rmdir 3`

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

```
ls /aXGTEar || true
true && false || true
false && false && false
false || true || false
```

Run the following commands on the shell

```
ls aXGTEar | true
ls aXGTEar |& true
ls | false
```

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

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

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

Exercise 2.8. The following commands are all valid `bash` shell syntax. What happens in each case? 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`

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

2.5 Tips on getting help

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

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

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

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

A word of warning about shell builtin commands:

- It is intentional that shell builtin commands act extremely alike external commands
- Examples for perhaps surprising shell builtins are `cd`, `test` or `echo`
- Some of these commands — like `test` or `echo` — are provided by the OS as well.
- The builtins get preference by the `bash` for performance reasons

⇒ The manpage for some commands (describing the OS version of it) do not always agree with the functionality provided by the `bash` builtin.

- Usually the `bash` has more features

⇒ Bottom line: Sometimes you should check `help command` even though you found something in the manpages.

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

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

Chapter 3

Simple shell scripts

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

3.1 What makes a shell script a shell script?

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

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

This line, starting with a hash(`#`) and a bang(`!`) — hence the name — tells the OS which program should be used to interpret the following commands. If a file with executable rights is encountered that begins with a shebang, the OS runs the specified program (in this case `/bin/bash`) and feeds the remainder of the file to the programs *stdin*[1]. In order to compose a shell script we hence need two steps

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

3.2 Shell variables

Shell variables are defined using the syntax

```
VAR=value
```

and 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

[1]Strictly speaking the shebang is not required, since a missing shebang causes the default shell to be used — which is good enough for many cases. It is nevertheless good practice to include the shebang

name	value
USER	name of the user running the shell
HOSTNAME	name of the host on which the shell runs
PWD	The current working directory
RANDOM	Random value between 0 and 32767
HOME	The user's home directory
PATH	Search path for commands
SHELL	Full path of the shell invoked
IFS	Internal field separator (see section 4.8 on page 48)

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

```
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 substiutions (see section 3.2.2 on page 23), arithmetic expanson(see section 5.1 on page 52 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 is not to be interpreted as part of a 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[2], but they can be interpreted as integers if a builtin command requires this (e.g. `test` — see section 4.2 on page 30)
- Variables can also be deleted again using

```
unset VAR
```

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

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

3.2.1 Special parameters

Next to the variables we mentioned above, the shell also has a few special parameters, that are set to their values automatically by the shell. Their expansion works exactly like for other variables. The most important special parameters are

- 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 first second`, we get

  ```
  The first: first
  The second: second
  ```

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

Example 3.1. If the script

```
#!/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 52.

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 the code

  ```
      VAR=$(ls /nonexistant)
  ```

  in a shell would still write the error message `ls` prints to *stderr* on the terminal.

- Inside the `$()` we have a so-called subshell (see also section 6.1 on page 61), where output redirection is possible. We could suppress the error message by running

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

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

  ```
  echo $(echo one;echo two)
  ```

  gives

  ```
  one
  two
  ```

- The return code of a command substitution is the return code of the command list provided, ie the code of the last command executed. So in order to still inform the user that something went wrong, we could use

  ```
      VAR=$(ls /nonexistant 2> /dev/null) || echo something wrong here
    \item Command substitutions may be nested:
      \begin{lstlisting}
      VAR=$(echo $(echo $(echo value))) # VAR contains "value"
  ```

Exercise 3.2. Write a `bash` quine[3], i.e. a script that produces its source code as output when executed. Hint: The solution has about 20 characters.

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

- Write a script that takes a matrix file as first argument (arg) and an output file as second argument. After the execution the output file should contain the same content as the matrix file but the (first) comment line should appear as the last line, ie. comment and data should be swapped.

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

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

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

Input on *stdin* is very volatile, once you used it in a script it is gone forever (see section 4.7.2 on page 44 for more details on this). If we need to use it multiple times nevertheless we need a temporary cache, like in this example.

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]

If we wanted to use these characters in a string as a value for a variable, we need to *escape* them, ie preceed 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 commandlne

```
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[4], 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*
- A word is now considered a separate entity: Either a command or an argument to a command

Example 3.5. When the shell encounters the command line

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

It first substitutes the commands and parameters to give for example (KEYWORD =search, 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 the respective parts of the commandline by either the single quote "'" or the double quote """, e.g.

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

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

- single quote "'": No special characters, but "'" survive
  - ⇒ """, "$", "#" are all nonspecial
  - ⇒ No parameter expansion or command substitution
  - ⇒ No word splitting
- double quote """: Only """, "$" 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)'
```

[4]See appendix B.3.2 on page 105 for more details how a commandline is parsed

```
echo "$EXAMPLE"
echo "\"some other example: "␣$EXAMPLE2

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

3_simple_scripts/quoting_example.sh

which is

```
abcdef123Di␣18.␣Aug␣12:43:59␣CEST␣2015␣next
"some␣other␣example:␣␣$ABC$NUM$(data)
'test'
```

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:
```

Whilest

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

gives

```
The␣first:
The␣second:␣-h
```

Omitted quoting or escaping is a very commen source of error. Some hints:

- When passing arguments to commands *always* quote it using double quotes (unless you have a reason not to)
  - ⇒ This avoids problems when variables are empty
  - ⇒ It does not hurt anything
- When initialising variables *always* quote the values using double quotes
  - ⇒ Same reason as above
- When a variable contains a file name be extra careful that you use double quotes everywhere you use its value
  - ⇒ Paths or filenames may contain spaces
- Use syntax highlighting in your editor[5]
  - ⇒ You will discover missing escapes or closing quotes much more quickly

[5]vi: `syntax on`, emacs: `font-lock-mode`

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

```
#!/bin/bash
# script to extract some information from directories
# $1: additional keyword to search for
#
cd Top Dir
ADDITIONAL=$(<output grep $1)
IMPORTANT=$(<output grep -i important)
cd Lower
FILE=$(<out1 grep -H $1; <out2 grep -H $2)
COUNT=$(echo '$FILE' | wc -l)

echo results:
echo "   important messages:" $IMPORTANT
echo '   other messages: $ADDITIONAL'
echo we found $COUNT more findings in
echo $FILE
```

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=$(echo line1; echo line2)`
- `VAR=$(echo line1)`
- `VAR=""`

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

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

You will learn in the next section that the `-n` flag prevents echo from printing an extra trailling `<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 80. 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 as well to verify that it works.

Exercise 3.10. Write a script that takes a keyword (which may contain spaces) as an argument. Use recursive `ls` (manpage) to find all directory paths below the working directory that match the keyword. Print these results followed by the number of matches.

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

- No other output on *stdout* should be produced by the script

- If the file cannot be read the script should exit with a status code 1, else with code 0 (see `help exit` if you do not know the exit command)
- Count the number of characters excluding comments (use the script `charcount.sh` for this) The shortest shell script (using only what we have covered so far) wins `:)`

Chapter 4

Control structures and Input/Output

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

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*. A few notes:

- If one wants to print to *stderr* one can use a special kind of redirector, namely `>&2`[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 27)

- The argument `-e` enables the interpretation of a few special escaps (see `help echo` and table 4.1)

[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`

4.2 The `test` program

`test` is a very important program that is used all the time in scripting. Is main purpose is to compare numbers or strings or to check certain properties about files. It is exetremely feature-rich and this section can only serve as a simple introduction. For more detailed information about `test`, consider `help test` and the `bash` manual [2].

Most tasks of the test program 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 before the closing “`]`” is important, else the command fails.
- `bash` can only deal with integer comparison and arithmatic. Decimal values cannot be compared on the shell (but there are other tools like `bc`, see 5.2 on page 56)
- 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"
  ```

  where `grep` is only executed if the file ”file” really does exist.
- There also exists the command `[[` in the `bash` shell, which is more powerful. We will talk about this command briefly when we introduce regular expressions in section 7.1.1 on page 80.

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

Exercise 4.1. Write a shell script that takes 5 arguments and prints them in reverse order If -h is entered anywhere a short description should be printed as well.

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

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

4.3 Conditionals: `if`

The simplest syntax of the `if` command is

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

- 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

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

4_control_io/ifexamples.sh

gives output

```
  VAR=4
  /export/home/mherbst/Dokumente/Lehre/2015.08_Bash_Kurs
```

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, and if its exit status is zero, the corresponding `then`-`list` is executed and the command completes.
- Otherwise, the `else`-`list` is executed.
- The exit status is the exit status of the last command executed, or zero if no condition tested true.

Example 4.3. The script

```
#!/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 "/nonexistant"

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

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 non-zero return code.

```
#!/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
```

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 exit always 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.

```
#!/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. Write a script that takes two integer values as args, I and J. The script should:

- create directories from 1 till I
- Put empty files named 1 till J in each of these directories
- Do something sensible 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`
- *optional* If a third argument is a file, the script should copy this file to all locations

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

- If called with a single argument, print all integers from 1 to this value
- If called with two arguments, print from the first arg to the second arg
- If called with three arguments, print from the first arg to the third in steps of the second.
- Your script should print help if one of the three arguments is `-h`
- Your script should print an error if the combination of arguments provided makes no sense. It should explain why the combination makes no sense.

4.5 Loops: `for`

Basic syntax:

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

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

```
#!/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

Since word splitting occurs right before the execution, i.e. basically after everything else, there is quite a 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=$(< resources/matrices/3.mtx grep 1)
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
1
1
1
1
2
1
```

- 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 charaters `*` and `?` are the special pattern characters. If replacement of `*` by zero or more arbitrary characters gives the name of an existing file, this replacement is done before execution. The same is true for `?` which is replaced by exactly one arbitrary character.[2] In the context of `for` loops this is usually encountered like this

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

```
#!/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: Unmatched 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
```

- Combination of all of these

A word of warning: The The paradigm

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

is extremely problematic, since files with spaces are not properly accounted for[3] 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
```

[3]The reason is that command substitution happens earlier than pathname expansion. Therefore the results of `$(ls)` go through word splitting before being executed. See appendix B.3.2 on page 105 for more details.

Exercise 4.6. Here we start a small project trying to recommend a book from Project Gutenberg based on a few 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 you're script also works if spaces in the pattern or in the files are encountered
  - Ignore the case of the pattern
  - You may assume all books of Project Gutenberg to be `.txt` files
  - Provide help if the argument is `-h`
  - Use proper error statements if something goes wrong or is not sensible.
- Change your script such that it prints the number of matching lines and the number of actual lines next to the script name. The fields of the table should be separated by tabs. A possible output could be

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

- *optional* Suppress the output of books without any match
- *optional* Extend your script such that arbitrary many keywords may be provided

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

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

4.6 Conditional: `case`

The `case` command has the following basic syntax:

```
case word in
  pattern) list ;;
  [ pattern) list ;; ]
  ...
esac
```

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

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

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

```
#!/bin/bash
VAR=$@     # all arguments assigned to VAR
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 specific
    # than pattern l* above
    ;;
  $1) echo "VAR is \$1"
    ;;
  *)  echo "VAR is something else"
    ;;
esac
```

4_control_io/caseexample.sh

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 in scripts. A very common paradigm is `while`-`case`-`shift`[4]

```
#!/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"
```

[4] no official name, but my own creation :)

```
            ;;
        # 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 approriate 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 --shou`

```
-f encountered, file: file
Unknown argument: --shou
```

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

- `-h`, `-q`, `-v`
- `--help`, `--quiet`, `--verbose`
- `-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 is a valid file, else throw an error and exit
- If `-f-` is not given, the file should be `resources/matrices/3 b.mtx`.
- Display all arguments that were given, unless `--quiet` or `-q` are given
- If `-v` or `--verbose` are given it should also print the first 10 lines of the file on `-f`

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.

Using the return code of read it is easy to check whether we were able to obtain *any* data from the user or not. We cannot check, however, whether all fields are filled or not.

```
#!/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

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

```
You␣entered␣"1",␣"2",␣"3"
```

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

```
You␣entered␣"",␣"",␣""
```

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

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

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

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

Example 4.10. Consider the script

```
#!/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. In fact we get exactly the same result as if only a single cat would be contained in the file. 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 doing the same it encounters the EOF character straight away, signalling that it has reached the end of the stream. Hence no extra output is produced.

The same thing occurrs if we use two other commands that keep reading until nothing is left, like two `grep`s after another:

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

the second `grep` is pointless. If subsequent `grep`s on *stdin* are desired, one usually uses a temporary caching variable to circumwent these problems:

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

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

```
#!/bin/bash
read FIRST
read SECOND
echo "$SECOND"
echo "$FIRST"
cat
```

4_control_io/swaplines.sh

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

Exercise 4.12. Write a simple script that outputs the third line provided on *stdin* to *stdout* and the fourth to *stderr*.

Exercise 4.13. Extend the script from the previous exercise:

- Use `read` to ask the user for two line numbers. Print those lines to *stdout* and *stderr*, respectively.
- Does the script work as expected? Why not?

Exercise 4.14. We want to write a more general version of exercise 3.3 on page 23.

- 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.
- If an error occurs, i.e. if the 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`.
- *optional* remove the restriction on the values of `--to` and `--from`

4.7.3 The `while read line` paradigm

Probably the most important application of the `read` command is the `while read line` paradigm[5]. 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

[5]Again not an offical name

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 it to read from a file

```bash
#!/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"
done < "$1"
```

4_control_io/addlinenumbers.sh

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

- we can pipe the output of a command to it

```bash
#!/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 loops output in a file by adding `> file` after the `done`

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

```
CACHE=$(echo "$CACHE"; echo "next line")
```

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

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

- Write a script that prints the second column (values are separated using one or more `<space>` or `<tab>` characters) of everything on *stdin*

Now modify your script such that it modifies the data of the `mtx` files of `resources/matrices` in the following way

- The script takes an argument on `$1`. If the argument is `-` or missing it should read *stdin*, else `$1` should be interpreted as a file and read from this file (This is a very common paradigm, see `man cat`, `man grep`)
- Lines is the `mtx` file starting with `%` are ignored
- Of all other lines the value in the second column is increased by one and the modified line is printed on *stdout*

Exercise 4.17. `find` is a really handy program to search for files and directories with uncountable options (see `man find`). You can find the most important options in table 4.3 on the next page. `find` per default searches through all directories and subdirectories and prints the relative paths of file matching the conditions to *stdout*. All options you provide are connected using a logical *and*. This can of cause be changed (see documentation).

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 keyword provided on `$1` and print to stdout on which line and in which file the match occurred
- *optional* The script should pass all options provided as arguments to the relative `grep` commands, ie an invocation `./grep_all -i word` should use `grep -i word` to filter the files

optional Write a second script `open_match` that passes all its arguments to `grep_all`. If only a single match is found the script should fire up your favourite editor at precisely this location. [6]

[6]For `vim` this can be achieved by executing “`vim file +line`”

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

4.8 Influencing word splitting: The variable IFS

In table 3.1 on page 21 we already mentioned the variable IFS.

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

Two very important use cases of manipulation of the IFS variable:

- Manipulation of the way for loops iterate:

```
#!/bin/bash
OIFS=$IFS
IFS="+"
VAR="4+5+6+7"

# before the for loop runs the value after the "in"
# is subject to word splitting
echo first loop
for number in $VAR; do
  echo $number
done
echo

# it is good practice to change IFS back to the original
# after you used the trick, otherwise all sorts of crazy
# erros can occurr
IFS=$OIFS

echo second loop
for i in 1 2 3 4; do
  # this works now as intuitively expected:
  echo $i
done
```

4_control_io/IFS_for.sh

```
first loop
4
5
6
7

second loop
1
2
3
4
```

- Influencing `read`:

```bash
#!/bin/bash

ARG="foo"
VAL="bar"
COMMENT="Some crazy comment"

# here we run code to determine the values of ARG, VAL, COMMENT

# store it for later usage in a more compact form
STORAGE="$ARG+$VAL+$COMMENT"

# ...

# unpack it again
OIFS=$IFS
IFS="+"
echo "$STORAGE" | {
  read ARG VAL COMMENT
  echo "The argument was $ARG"
  echo "The value was $VAL"
  echo "The comment was $COMMENT"

} # see next chapter why we need the { ... }
# ignore it for now
IFS=$OIFS
```

4_control_io/IFSread.sh

```
The argument was foo
The value was bar
The comment was Some crazy comment
```

Exercise 4.18. In this exercise we will write a script called `unpack_pair.sh`.

- It should take a single separator character on `$1`
- Whenever this character (and only this character) is encountered on *stdin*, it should be replaced by a `<newline>` character in *stdout*.

Exercise 4.19. The shell uses the following procedure to lookup the exact location where commands should 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 the first word gives a file that is executable, execute this file.
- Else keep searching in the next directory in `PATH`

Example: The commandline

```
vim testfile
```

has the first word `vim`. Consider

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

a lookup reveals that the file `/usr/bin/vim` exists and is executable. So this file is executed.
There exists a tool, called `which`, that does exactly this lookup when provided with a command as firs 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`
- If it exists print the full path and return 0
- Else return 1

4.9 Conventions when scripting

To conlude 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 philosopy [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.

As usually: There are many exceptions to each of the guidelines below. As a rule of thumb: Follow each guideline, unless there is a good reason not to.

[7] This is a slight simplification

4.9.1 Script structure

- Have a `shebang`. Dot.
- A block doing a task should have a comment explaining what happens, what goes in and what comes out.
- Whenever funny `bashisms` are used that could make code unclear, explain what happens.
- One script should only do one job only. Split complicated tasks into many scripts. This makes it easier to code and easier to reuse.
- Use shell functions (see section 6.2 on page 67) to structure your script. Have a comment what each function does.

4.9.2 Input and output

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

⇒ Can be summarised as "Design each script as a filter"

- Use `mktemp` for temporary files and clean the mess up afterwards (see section 6.3 on page 75)

4.9.3 Parsing arguments

- Each script should support the arguments `-h` or `--help` and explain what it does and at least the most important commandline arguments it supports.
- For each argument there should be a descriptive "long option" preceeded by two "`--`". There may be short options (preceeded by one "`-`").
- Do not worry about the long argument names. You can code tab completion (see section B.1.1 on page 105) for your script.

Chapter 5

Avanced parameter expansions and arithmetic expressions

In this chapter we will expand on two topics we already briefly touched: Arithmetic expansion and parameter expansion (in section 3.2 on page 20).

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 58). One may, however, also access or assign variables without the leading $

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

  ```
  7
  ```

- Note: Positional parameters are *not* available
- All common operators are available:

[1]The precise rules are more or less identical to the rules of the C programming language

- `+ -` addition, subtraction
- `* / %` multiplication, (integer) division, remainder
- `**` exponentiation
- `name++ ++name name-- --name` increment and decrement operators
- `+= -= *= /= %=` Infix assignment

```bash
#!/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
- 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 point seem a little strange at first, but they assure that arithmetic expressions can be used as a replacement for `test` in `while` or `if` constructs

```
#!/bin/bash

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

5_variables/arith_replacement.sh

```
I␣can␣be␣devided␣by␣3:␣3
I␣can␣be␣devided␣by␣3:␣6
I␣can␣be␣devided␣by␣3:␣9
I␣can␣be␣devided␣by␣3:␣12
I␣can␣be␣devided␣by␣3:␣15
I␣can␣be␣devided␣by␣3:␣18
I␣can␣be␣devided␣by␣3:␣21
I␣can␣be␣devided␣by␣3:␣24
I␣can␣be␣devided␣by␣3:␣27
```

```
I␣can␣be␣devided␣by␣3:␣30
I␣can␣be␣devided␣by␣3:␣33
I␣can␣be␣devided␣by␣3:␣36
I␣can␣be␣devided␣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
- The whole construct is replaced by the final value the `expression` gives rise to
- The return code of `(( ))` is not available.
- The expression may be used just like ordinary parameter expansion `${VAR }`

```bash
#!/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 up to this point. This incudes intermediate expressions, i.e.

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

5_variables/arith_intermediate_floats.sh

```
26
0
```

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.

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

```bash
((B=0))
$((B=0))
echo $((B=0)) | grep 0
for((C=100,A=99 ; C%A-3 ; C++,A-- )); do ((B=(B+1)%2)) ; done; ((B))
((B=1001%10)) | grep 4 || (( C=$(echo "0"|grep 2)+4, 2%3 )) && echo $((4-5 && C-3+B)) | grep 2
```

Exercise 5.2. For the arithmetic expansion an empty variable or a string that cannot be converted to a number counts as zero("`C`-false")

- How could that be exploited to test wheather a variable can be properly converted to a number?
- Write a script that calculates all primes between 1 and `N`, where `N` is a number supplied as the first argument to your script.
- Check that `N` is a sensible number before entering the routine.

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

```bash
# 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 mark powers.

```
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 88) or `python`. Most of the time it is, however, still sensible to use `bc`, since it is extremely, i.e. quick to start up.

Exercise 5.3. Now we want to extend our project to recommend books from Project Gutenberg. Recall that your script from exercise 4.6 on page 40 gives output of the form

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

where the second column was the number of matching lines and the third column was the number of actual lines in the file. Write a script that

- Takes keywords as arguments, which are then used to call the script from exercise 4.6 on page 40
- Parses the respective script outputs
- Calculates for each book the relative importance given as

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

- Sorts the books according to the relative importance
- Suggests 3 books for the user and gives their score.
- Be sure to deal properly with cases in which less than 3 books match

Try a few keywords, e.g. "Baker", "wonder", "the", "virgin", "Missouri, Kentucky". Any observations?

Exercise 5.4. 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
- The mean if `-m` is provided
- Give an error if something does not make sense or neither `-m` nor `-s` are given.

Exercise 5.5. *optional* Read about the `mtx` format in appendix C.1 on page 107.

- Write a script that takes a `mtx` file on *stdin* and a number on `$1`.
- The output should be again a valid `mtx` file where all entries are multiplied with said number.
- The comment in the first line (but not neccessarily any other) should be preserved

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

- *use-default*

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

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

- *use-alternate*

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

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

```
#!/bin/bash

A=
B=3

echo ${B:+"B works"}
echo ${A:+"A works"}
echo ${A:-"notA: "$B}

echo ${A:="defined"}
echo ${A:+"A works"}
echo ${A:-"notA: "$B}
```

5_variables/pexp_use.sh

```
B works

notA: 3
defined
A works
defined
```

- *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 chagacters starting from offset are printed. Both length and offset are arithmetic expressions

- *parameter length*

```
${#parameter}
```

Expands into the number of characters parameter currently has.

```
#!/bin/bash

VAR="some super long string"
LEN=${#VAR}
echo $LEN

# 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_length.sh

```
22
super long
super long
```

- *pattern substitution*

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

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

```
#!/bin/bash
VAR="some super long string"
PATTERN="s*e"
PATTERN2="?r"
REPLACEMENT="FOOOO"

# the longest match is replaced:
echo ${VAR/$PATTERN/$REPLACEMENT}
echo ${VAR/$PATTERN2/$REPLACEMENT}

# all matches are replaced
echo ${VAR//$PATTERN2/$REPLACEMENT}
```

5_variables/pexp_subst.sh

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

Exercise 5.6. Implement the `basename` program in `bash`:

- Take two arguments: A path on `$1` and an (optional) extension
- If the extension is *not* provided, strip the file extension encountered off the path
- If the extension *is* provided, remove only this file extension if any.

There also exists a handy `dirname` tool, which removes the last component of a path, i.e. everything after the last "`/`".

- *optional* Recode the `dirname` tool in `bash`

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; }
```

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

The syntax is e.g. useful for

- Unpacking data

```
#!/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 "Crazy␣header"
  echo
  echo "A␣first␣message␣to␣stderr" >&2
  echo "I␣want␣fish" | grep -w fish
  echo "lorem␣ipsum␣dolor␣sit␣amet"
  echo "This␣goes␣to␣the␣stderr" >&2
} > /tmp/some-file-here 2> /tmp/file-stderr

# print content
echo Everything on the first file:
echo -----------
cat /tmp/some-file-here
echo -----------
echo
echo "Everything␣on␣the␣second␣file:"
echo -----------
cat /tmp/file-stderr
echo -----------

# cleanup
rm /tmp/some-file-here /tmp/file-stderr
```

6_functions_subshells/group_write_file.sh

```
Everything␣on␣the␣first␣file:
-----------
Crazy␣header

I␣want␣fish
lorem␣ipsum␣dolor␣sit␣amet
-----------

Everything␣on␣the␣second␣file:
-----------
A␣first␣message␣to␣stderr
This␣goes␣to␣the␣stderr
-----------
```

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.

An example

```
#!/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/mherbst/Dokumente/Lehre/2015.08_Bash_Kurs
Hello from subshell: A: 5   B: 6
/export/home/mherbst/Dokumente/Lehre
Hello from main shell: A: 3   B: 6
/export/home/mherbst/Dokumente/Lehre/2015.08_Bash_Kurs
```

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

```
#!/bin/bash

#Here want to do some stuff in the PWD
echo "The list of files in the PWD:"
ls | head -n 4
(
```

```
  # do stuff in a different directory
  cd resources/matrices

  # and using a different IFS
  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:
0Administratives
0_preamble.tex
10_performance
1_intro_Unix

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

Some paths:
/export/home/mherbst/opt/paraview/current/bin
/opt/software/openmpi/gcc-4.8.2/1.6.5/bin
/export/home/mherbst/opt/tor/current/bin
/export/home/mherbst/opt/arxiv2bib/bin

word1:word2
```

6.1.3 Implicit subshells

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

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

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

```
C=0 # initialise counter
find -type f | while read line; do
  # subshell here!
  ((C++))
done
#not in subshell any more:
echo "We␣found␣$C␣files"
```

6_functions_subshells/subshell_pipes.sh

```
We␣found␣0␣files
```

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:

```
#!/bin/bash
COUNT=$(find -type f | {
  C=0
  while read line; do
    ((C++))
  done
  echo $C
})
echo "We␣found␣$COUNT␣files"
```

6_functions_subshells/subshell_pipes_correct.sh

```
We␣found␣575␣files
```

- Command substitutions: Usually less of a problem

```
#!/bin/bash
A=-1
echo $( ((A++)); echo $A )
echo $A
```

6_functions_subshells/subshell_commandsubst.sh

```
0
-1
```

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

```
#!/bin/bash

# initial note:
#    this script is deliberately made cumbersome
#    this script is bad style. DO NOT COPY

# keyword
KEYWORD=${1:-0000}

ERROR=0
[ ! -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!
```

Exercise 6.2. Rewrite your PATH-lookup script from exercise 4.19 on page 50 using the features from this section wherever it is sensible.

Exercise 6.3. Rebuild the `find -type f` command using `ls` and the features of the `bash` shell. Hint: Use subshells to keep track of the current directory level you are in.

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 excute `list` available 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
testfct() {
  echo blub  #write to stdout
  read test  #read from stdin
  read test2  #also read from stdin
  echo $test >&2 #write to stderr
  echo $test2  #write to stout
}

{
  echo line1
  echo line 2
} | testfct | grep 2
```

6_functions_subshells/fun_pipe.sh

```
line1
line␣2
```

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

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

```
1
2␣3
1␣2␣3␣4␣5
4
```

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

```
#!/bin/bash

return_test() {
  if [ "$1" == "a" ]; then
    echo "No␣thanks"
    return 1
  fi

  echo "Thank␣you"
}

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

VAR=b
if other_test "$VAR"; then
  return_test "$VAR"
  echo $?
fi

return_test "a"
echo $?
```

6_functions_subshells/fun_return.sh

```
Thank␣you
0
1
```

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

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

[2]Functions directly or indirectly called by the function, i.e. called functions, functions called from called functions, . . .

```
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.

Good practice when using functions:

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

Compare the two code snippets and decide for yourself what is more readable[3]

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

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

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

```
  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

```
#!/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
}

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
}

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

Exercise 6.4. Take another look at your script from the second Project Gutenber exercise (exercise 5.3 on page 57). Split the script up into a few sensible functions. A few ideas:

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

Exercise 6.5. Write the following functions:

- A function `map` that takes a function name as first argument and applies this function to all other arguments supplied. The results should just go to *stdout* and be separated by `<newline>`.
- A function `add` that adds the 2 arguments provided
- A function `multiply` that adds the 2 arguments provided
- A function `operation` that reads the global variable SEL and depending on its value calls `add` or `multiply` with the arguments provided
- A function `calculate3` that takes a single argument and calls `operation` with this argument and the number "3"

Write an encapsulalting script that

- allows to select between the calculation method (e.g. the argument `--add3` selects addition, the argument `--multiply3` multiplication)
- passes all other commandline arguments to `calculate3` (After the script knows they are sensible numbers)

How much efford does it take to add a third option that allows to subtracts 3 from all input parameters? *optional* Remove the explicit "3" and allow the user to set this value on the commandline.

6.2.1 Overwriting commands

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

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

```
#!/bin/bash

test() {
  echo "Hi␣from␣the␣test␣function"
}

VAR="blubber"
test -z "$VAR" && echo "VAR␣is␣zero"
```

6_functions_subshells/overwrite_fail.sh

```
Hi␣from␣the␣test␣function
VAR␣is␣zero
```

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

```
#!/bin/bash

C=0 # count to break at some point

[() { # overwrite the [ builtin

  # use test to end at some point
  if test $((C++)) -gt 100; then
    echo "$C"
    exit 0
  fi

  # 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

```
102
```

In scripts it is best to avoid this feature since it can make code very unintuitive and hard to understand. For customising your interactive `bash`, however, this can become very handy (see appendix B.1.2 on page 105).

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

```
#!/bin/bash

printer() { echo "1"; }

```

```
for((I=0;I<10;++I)); do
  printer
  printer() { echo "$I"; }
done
```

6_functions_subshells/overwrite_mostrecent.sh

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

Again this feature should be used with care.

6.3 Cleanup routines

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

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

# 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 variable

# 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. 6.5 on page 73). 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. This means that all variables and functions defined in `otherscript` are also available to the shell afterwards:

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

6_functions_subshells/sourcing.lib.sh

```bash
#!/bin/bash

PATH="$PATH:6_functions_subshells"
. sourcing.lib.sh  #lookup performed in PATH

echo $VAR
testfunction
```

6_functions_subshells/sourcing.script.sh

```
foo
Hey I exist
```

Note: In order to find otherscript the bash honors the environment variable PATH. As the example suggests this way libraries defining common or important functionality may be defined and used from different places.
A good trick is to make each script sourcable by default. Since the return statement is allowed in scripts which are sourced and not in scripts which are executed, one can realise a break between function definitions and "global code" that is only considered when a script is sourced:

```bash
#!/bin/bash

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
}

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
}

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"
}
```

```

#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
  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/source_sourcability.sh

Chapter 7

Regular expressions

Up to this point we covered most important features of the `bash` shell[1]. We will now discuss regular expressions, a syntax that is used in a lot of Unix tools in order to search or describe textual data.

7.1 Regular expression syntax

7.1.1 Matching regular expressions in plain bash

Before we introduce regular expressions, we will first need a tool to try them out with. `bash` already has a syntax understanding regular expressions or *regex*es, which gives us a good starting point:

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

- This command returns with exit code 0 when `string` can be described by the regular expression `regex`, else it returns 1.
- If `string` is described by `regex` one also calls this a "match" or says that `string` contains a match for `regex`.

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

7.1.2 Regular expression operators

It is best to think of regular expressions as a "search" string where some characters have special meaning. All non-special characters just stand for itself, e.g. the regex "`a`" just matches the string "`a`"[2].

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

[1] A list of things we left out can be found in appendix B.4 on page 106

[2] This is why for `grep`, we could just grep for a word not even knowing that it uses regexes by default

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`*` The *preceeding* regular expression should be repeated as many times as neccessary to find a match, e.g. "`ico*`' matches "`ic`", "`ico`" or "`icooooo`", but not "`icco`". The "`*`" applies to the *smallest* possible expression only.

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

`+` Similar to "`*`": The preceeding expression must occurr at least once

```
[[ "woo (rd" =~ wo+ \( ]]; echo $?      #true
[[ "oo (rd" =~ (wo)+ \( ]]; echo $?     #false
[[ "wo (rd" =~ (wo)+ \( ]]; echo $?     #true
```

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

There are a few things to note

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

7.1.3 Special bracket expansions and POSIX character classes

Both bracket expansion and complemented bracket expansion allow for a shorthand syntax 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]`

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

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

Note that POSIX character classes can to 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
  exit 0
fi
```

7.1.4 Getting help with regexes

Writing regular expressions takes certainly a little practice, but is exetremely 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 7.1. Describe in words what kind of strings are matched by these regular expressions

- `..|[a-e]`
- `.:*$|[^[:digit:]]+`
- `(.:)*|[^[:digit:]]+`
- `^[^[:space:]]+abacus [A-F]?`

optional Fill the following crossword:

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

Exercise 7.2. Give regular expressions that satisfy the following

	matches	does not match
a)	`arrp`, `whee`	`bla`, `kee`
b)	`qome`, `pyol`, `mhr`	`bedw`, `tnu`, `isa`
c)	`a=0.0345`, `a=.0500e-4`	`a=nf`, `a=0.0000`

Note: The art of writing regular expressons is to use the smallest number of characters possible to achieve your goal.

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. Care has to be taken, however, to properly quote or escape the characters which are special characters to the shell, such that they are actually passed on to `grep`. 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 neccessary things manually. Note that this can lead to constructs like

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

where a lot of backslashes are needed.

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

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

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 "'" or e.g. parameter expansion are needed in `sed_commands` double quotes should be used.

Overview of basic sed commands[6]:

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

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

[5]This is not fully correct, see `grep` manpage for details.

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

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

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

```
#!/bin/bash

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

echo ------

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

7_regular_expressions/sed_insertion.sh

```
blub
laber
blbl
laber
------
laber
blub
laber
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 `text`. Furthermore we can use the escape sequences `\1`, `\2`, ..., `\9` to refer to the matched text of 9 subexpressions in `regex` marked using `\( ... \)`. 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.*\([1-3]\)/2:\2 1:\1 full: &/'
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

-----
2:1␣1:l␣full:␣line1
␣␣␣␣␣␣line␣␣2
2:3␣1:L␣full:␣LiNE3

-----
line1
line␣2
LiNE3
```

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

7.3.1 Alternative matching syntax

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

- In front of a command, `/regex/` can also be expressed as `\c regex c`, where `c` is an arbitrary character.

- For the command `s`: `s c regex c text c` is equivalent to `s/regex/text/`.

```
#!/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 7.3. Use `grep` along with the standard Unix tools to achieve the following:

- Find out, how often do the digits 0 to 9 occurr in the file `resources/chem_output/qchem.out`. Hint: Use `grep -o` and a suitable regex.
- Remind yourself of the `mtx` format in appendix C.1 on page 107. Design a regular expression to match all "value" fields (the third column) in the `mtx` files of directory `resources/matrices`. Use it to find the largest value contained in all files.
- Amend your regex from the previous part such that it does not match lines which are commented out.

Exercise 7.4. 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` or which are empty.
- Remove all initials and bothering `.` or `-` symbols (Do not remove `-` on compound surnames!)
- Replace all `,` by `\n`, the escape sequence for a line break.
- Do cleanup: Remove unneccesary 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 recode the functionalty of one of the afforementioned programs.

8.1 Structure of an awk program

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

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

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

During execution awk goes from record to record and tries to match each condition to it. If a match is encountered the action code corresponding to the matching condition is executed.

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

Similar to the shell the # starts a comment in awk programs and <newline> and ";" may be both be used to terminate a rule line.

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_soure
  ...
'
```

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

Example 8.1. Just to give you an example for what we discussed in the previous sections, this is a shell script calling an inline **awk** program printing some nice messages[1]

```
#!/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:
  { print "Thank␣you" }
'
```

8_awk/basic_example.sh

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

8.3 Strings, variables and arithmetic in awk

Strings in **awk** all have to be in double quotes, e.g.

```
print "This␣is␣a␣valid␣string"
```

Multiple strings may be concatenated, just by leaving whitespace between them

[1]I will use sytax highlighting adapted for **awk** code for input files in this chapter.

```
#!/bin/bash
# the echo is just here to make awk do anything
echo | awk '{ print "string1"   " "   "string2" }'
```

8_awk/vars_stringconcat.sh

```
string1 string2
```

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"
```

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

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

- `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
- 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 conditon statements)

```
#!/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 52. 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

  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
```

Some variables in `awk` have special meaning:

`$0` contains the content of the current record. Note that the `$` 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 "some 7 words for 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 "some words for awk 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 8.6 on page 98.

`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.

8.3.1 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 8.2. Take another look at your script from exercise 6.5 on page 73. Use `awk` to make it work for floating-point input as well.

8.4 `awk` conditions

Each action block may be preceeded 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 (incuding $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 independant 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 8.3. Use `awk` to achieve the following

- Rebuild the command `wc -l`, i.e. count the number of line in a file.
- Rebuild the command `uniq`, i.e. discard duplicated lines in *sorted* input.
- *optional* Also rebuild the `uniq -c` functionality in the upper script.

Exercise 8.4. The file `resources/chem_output/qchem.out` contains the logged output of a quantum-chemical calculation. During this calculation 2 so-called Davidson diagonalisations have been performed. Say we wanted to extract how many iterations steps were neccessary 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.

8.5 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.

```
#!/bin/bash

{
  echo record1 word2
  echo record2 word4
  echo record3 word6
} | awk '
  { print NR ": first rule" }
  /4$/ { next; print NR " " $1 }
  { print NR ": "  $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 '
```

```
  { print NR ": first rule" }
  /4$/ { exit 42; print NR " " $1 }
  { print NR ": "  $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[2]. Just `print` (without an argument) is identical to `print $0`.

`printf` Formatted print. See next section for details.

8.5.1 Using printf

TODO

8.5.2 Conditions inside action blocks: if

`awk` also has analoguous control structures to the ones we discussed in chapter 4 on page 29 for `bash`. We don't want to go through all of these here[3], 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 8.4 on page 93. As usual the `else`-block may be omitted.

8.6 Further examples

Example 8.5. 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

[2]Can be changed. See section 5.3. of the `awk` manual [3] for detais

[3]See section 7.4 of the `awk` manual [3] for all the remaining ones.

Example 8.6. 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

Example 8.7. This program finds duplicated words in a document. If there are some, they are printed and the progrem returns 1, else 0.

```
#!/bin/bash
awk '
  # change the record separator
  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 8.8. Rebuild the `tac` command in `awk`. Notice that `print` does honor escapes like “`\n`”, “`\t`”, … by their special meaning.

Exercise 8.9. 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 achive 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.

- 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.

8.7 `awk` features not covered

This section is supposed to provide a quick overview of the features of `awk` we did not touch upon. For further reading about `awk` and any of the topics mentioned here I can recommend the `gawk` manual "GAWK: Effective AWK programming" [3]. It is both comprehensive and well-written. In the following list the paragraph numbers in brackets refer to appropriate sections of the `gawk` manual.

- Control structures and statements (§7.4) in `awk`: Loops, `case`, ...
- `awk` arrays (§8)
- `awk` string manipulation functions (§9.1.3): Substitutions, substrings, sorting
- Writing custom `awk` functions (§9.2)
- Reading records with fixed field length (§4.6): Fields separated by the number of characters, not a regex.
- Reading or writing multiple files (§4.9)
- Executing shell commands from within `awk` programs (§4.9)
- Creating `awk` code libraries (§10)
- Arbitrary precision arithmetic using `awk` (§15): Floating point computiation 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 and so even for `bash` scripts there are a few things worth noting when large amounts of data should be processed:

- Use the shell for as much as possible. Calling external programs is by far the most costly step in a script. So this should really only be done when the external program does more than adding a few numbers.

- 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 code.
  - 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 $(ls *); do
  command "$file"
  # or worse:
  command $file
done
```

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

```
for file in *; do
  command "$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 somecommand)

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

we can just write

```
if <file somecommand;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 unneccesary, use e.g.

```
grep -c word
```

instead.

9.1.5 When grep is not enough . . .

. . . than 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

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

where

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

is much nicer to read and feels more natural.

Appendix A

Obtaining the files

In order to obtain the example scripts and the files necessary 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.

Note that all exercises and example scripts should run without any problem on all LinuX systems. For other Unix-like operating systems like Mac OS X I cannot guarantee this, however.

Appendix B

Other `bash` features worth mentioning

B.1 `bash` customisation

TODO

B.1.1 Tab completion for user scripts

TODO

B.1.2 The `.bashrc`, `.profile`, `.bash_logout` configuration files

TODO

B.2 Making scripts locale-aware

TODO

B.3 `bash` command-line parsing in detail

B.3.1 Definitions

TODO

B.3.2 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

TODO expand on this

B.4 Notable `bash` features not covered

The following list gives some keywords for further exploration into scripting using the `bash` shell. See the bash manual [2] or the advanced bash-scripting guide [5] for more details.

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

Appendix C

Supplementary information

C.1 The `mtx` file format

The `mtx` files we use in this course[1] for demonstration purposes, follow a very simple structure

- All lines starting with "`%`" are comments
- The first non-comment line contains three values separated by one or more `<space>` or `<tab>` characters:
  - The number of rows
  - The number of columns
  - The number of non-zero entries
- All following lines — the non-zero entries — have the structure
  - Column index
  - Row index
  - Value

  where the values are again separated by one or more `<space>` or `<tab>` chars.

[1]We will only use a subset of the full format

Bibliography

[1] Eric S. Raymond. The Art of Unix Programming, September 2003. URL http://www.faqs.org/docs/artu/.

[2] Bash manual. URL https://www.gnu.org/software/bash/manual/.

[3] Arnold D. Robbins. GAWK: Effective AWK Programming, April 2014. URL https://www.gnu.org/software/gawk/manual/.

[4] Sed manual. URL https://www.gnu.org/software/sed/manual/.

[5] Mendel Cooper. Advanced bash-scripting guide, March 2014. URL http://www.tldp.org/LDP/abs/html/.

List of Commands

`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
`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
`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
	Search in manpage summaries for keyword
	Access help for `bash` builtin commands
	Access the Texinfo manual for command