Advanced `bash` scripting

(block course)

Solutions to the exercises

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

Interdisziplinäres Zentrum für wissenschaftliches Rechnen
Ruprecht-Karls-Universität Heidelberg

6^{th} – 10^{th} November 2017

Contents

Solutions to the exercises

Solution to 1.6

A few things you should have observed:

- Using the `-L` flag we can change the language of the manual page shown. Most notably `man -LC command` will always show the manual page in English. Sometimes the content of the manpages is different depending on the language used. Most often the English manpage offers the best documentation.
- Different sections contain documentation about different topic, i.e

  section 1 Executable programs or shell commands

  section 2 System calls (needed for Unix programming)

  section 3 Library calls (needed for Unix programming)

  section 4 Special files and device files

  section 5 File formats and conventions of special files in the system

  section 6 Games

  section 7 Miscellaneous

  section 8 System administration commands

  section 9 Kernel routines (needed for Unix programming)
- For us the most important is the first section, i.e. the section documenting executables and shell commands
- By prepending the section number as first argument to `man`. E.g. try `man 2 mkdir` vs `man 1 mkdir`. Here `man 1 mkdir` gives the documentation for the `mkdir` command, the other for the system call.

Solution to 1.7

`grep`ing in Project Gutenberg

- The two solutions are

```
< pg74.txt grep hunger | wc -l
< pg74.txt grep -c hunger
```

  where the second one should be preferred, since it does the counting already in `grep`. This means that we need to call one program less ⇒ Usually better for performance.
- The options do the following:

  `-A n` Add n lines of input after each matching line

  `-B n` Add n lines of input before each matching line

  `-n` Print line numbers next to each matching input line as well

  `-H` Print file name next to each matching input line as well

`-w` A line only is displayed if exactly the keyword exists. Usually it is sufficient if the search string is *contained* in the line only.

- Run the command

```
< pg74.txt grep -nwA 1 hunger | grep -w soon
```

in order to find the numbers 8080 and 8081.

Solution to 1.8

Possible solutions are:

- Here we need to invert the file first in order for `head` to select the last 10 lines (which are now the first 10). Then another inversion using `tac` gives back the original order, i.e.

```
< resourches/digitfile tac | head | tac
```

- The trick is to use `tail -n +2`, i.e.

```
tail +n2 resources/matrices/3.mtx
```

- We can use the `-v` flag of `grep` in order to invert the result, i.e. now all non-matching lines are printed:

```
< resources/matrices/3.mtx grep -v %
```

- Use `cut` to extract the third field and `sort -u` to get a sorted list of the values with all duplicates removed. Now piping this to `wc -l` gives the number of lines in the output of `sort -u`, i.e. the number of distinct values:

```
< resources/matrices/3.mtx grep -v % | cut -d " "  -f 3 | sort -u | wc -l
```

- Now we need `sort` without the `-u`. We get the smallest as the first in the sorted output:

```
< resources/matrices/3.mtx grep -v % | cut -d " " -f 3 | sort | head -n1
```

- Running the equivalent command

```
< resources/matrices/bcsstm01.mtx grep -v % | cut -d " " -f 3 | sort | head -n1
```

gives the result 0. Looking at the file we realise, that there is actually another, negative value, which should be displayed here. The problem is that `sort` does lexicographic ordering by default. To force it into numeric ordering, which furthermore includes the interpretation of special strings like `1E-09`, we need the flag `-g`. The correct result is displayed with

```
< resources/matrices/bcsstm01.mtx grep -v % | cut -d " " -f 3 | sort -g | head -n1
```

- Running

```
< resources/matrices/lund_b.mtx grep -v % | cut -d " " -f 3 | sort -g | head -n1
```

gives an empty output. This happens since the file contains lines like

```
9 8  5.5952377000000e+01
```

where there are two spaces used between 2nd and 3rd column. The problem is that `cut` splits data at *each* of the delimiter characters — `<space>` in this case. In other words it considers the third field to be empty and will take the data `5.5952377000000e+01` to be in field 4. For us this means that there are empty lines present in the output of `cut`, which sort first and are printed by head.

- Using `awk`, we would run

```
< resources/matrices/lund_b.mtx grep -v % | awk '{print $3}' | sort -g | head -n1
```

which gives the correct result.

Solution to 2.1

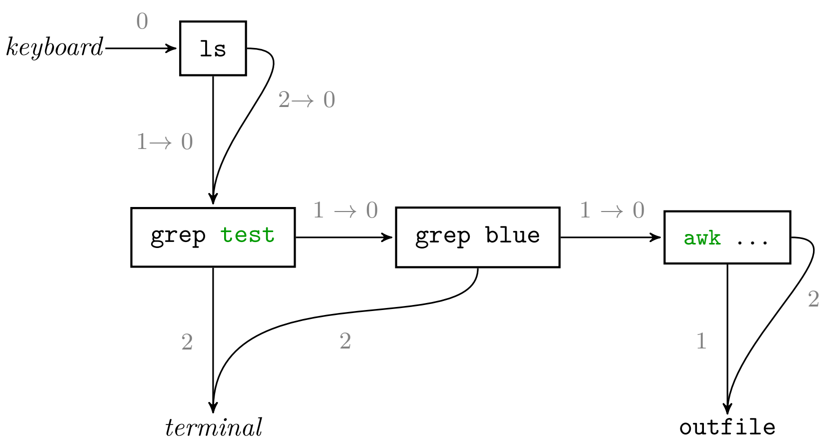
One way of doing this could be:

`cd !$` Enter
Ctrl + R `i` End ⟵ ⟵ ⟵ `blue` Enter
`ls|!:3-4` Enter
`mkdir !:2_red !:3_blue` Enter

This makes $5 + 11 + 9 + 22 = 47$.

Solution to 2.2

Using the same kind of redirection diagrams as in the notes, we get



where `awk ...` denotes the `awk '{print $2}'` command.

Solution to 2.3

Exploring `tee` for logging:

- The commandline proposed does not work as intended. Error output of `some_program` will still be written to the terminal and error messages of `tee` and `grep` both reach the `log.summary` file.
  This commandline, however, does work exactly as intended

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

  here both *stdin* and *stderr* of `some_program` reach `tee` and get subsequently filtered.

- Each time the program executes, both `tee` as well as the normal output redirector `>` will cause the logfiles to be filled from scratch with the output from the current program run. In other words all logging from the previous executions is lost.

  We can prevent this from happening using the `-a` (append) flag for `tee` and the redirector `>>`. Hence we should run

```
some_program |& tee -a log.full | grep keyword >> log.summary
```

Solution to 2.4

Some notes:

- Running `< in cat > out` is exactly like copying the file `in` to `out` as mentioned before.
- Running `< in cat > in` gives rise to the `in` file to be empty.
  This is because the shell actually opens the file handles to read/write data before calling the program for which input or output redirection was requested. This means that in fact the file handle to write the output to `in` is already opened before `cat` is called and hence `in` is already at the time `cat` looks at it (because the non-appending output file handle deletes everything). Overall therefore no data can be read from `in` and thus the `in` file is empty after execution.
- *stdin* is connected to the keyboard, *stdout* and *stderr* are connected to the terminal. Therefore everything we type(*stdin* of `cat`) is copied verbatim to the terminal (*stdout* of `cat`). The shell just seems to “hang” because `cat` waits for our input via the keyboard and thus blocks the execution of further commands. Ctrl + D sends an “EOF” character and hence signals that there is no more input to come. This quits `cat` and returns to the command propmt.

Solution to 2.5

- For the first case, pressing Ctrl + D signals `cat` that end-of-file, i.e. EOF, has been reached. This causes `cat` to quit processing and since no error has occurred since startup `0` is returned. Since no data was read by `cat`, also no output will be produced at all.
- Ctrl + C *aborts* the program and hence returs a non-ero exit code. Again no data was read so no data is returned.

Solution to 2.6

- `true` is a program that — without producing any output — always terminates with return code 0.
- `false` is a program that produces no output and always terminates with return code 1.

For the first set of commands the exit code is

- 0, since `false` returns 1 and hence `true` is executed, which returns 0.
- 0, since `true` triggers the execution of `false`, which in turn triggers the execution of `true`
- 1, since `false` returns 1, so nothing else is executed and the return code is 1.
- 0, since `false` causes `true` to be executed, which returns 0. So the final `false` is not executed and the return code is 0.

Running the commandlines in the shell, we get

- 0
- 0
- 1
- 1
- 0

In a pipe sequence, the return code is solely determined by the last command executed. In other words the return code of all other commands in the pipe is lost[1]

Solution to 2.7

This problem is meant to be a summary of the different types of syntax containing `&` and `|`.

- A usual pipe: The output of `echo test` on *stdout*, i.e. "test" gets piped into `grep test`, which filters for the string "test". Since this string is contained in `echo`'s output, we see it on the terminal and the return code is 0
- Recall that `&` sends the command to its LHS into the background. So the `echo` happens, which we see on the screen. At the same time `grep test` is executed, which does not have its *stdin* connected to a file or another program's *stdout*. In other words it has its *stdin* connected to the keyboard and it waits for the user to input data (The terminal "hangs".). Depending on what we type (or if we type anything at all) the return code of `grep` is different.
- A pipe where both the *stdout* as well as the *stderr* are piped to `grep`. The effect is the same as in the first example, since `echo` produces no output on *stderr*. I.e. we get "test" on the terminal and return code 0.
- We print once again "test" onto the terminal by executing `echo test`. Since this

[1] Not entirely true ... there exists a special array variable `PIPESTATUS` which actually captures the return code of each command of the pipe. How to use it is, however, out of the scope of this course.

is successful (zero return code) `grep test` is also executed. Similar to the second case, *stdin* of `grep` is connected to the keyboard and waits for user input. The exit code of `grep` — and hence the whole commandline — depends on what is typed.

- The `echo test` prints “test” onto the terminal and since this is successful, nothing else happens.

Solution to 2.8

Possible solutions are

- The question asks explicitly to just search for the word “the”, so we need to use `grep -w`:

```
< pg1661.txt grep -w the && echo success || echo error
```

- We need to provide `grep` with the `-q` argument as well:

```
< pg1661.txt grep -wq the && echo success || echo error
```

  This code executes a lot quicker, since `grep` can use a different algorithm for the search: Once it found a single match, it can quit the search and return 0.

- We need to use `grep` twice. Otherwise we get a “0” printed if there is no match:

```
<pg1661.txt grep -wq Heidelberg && <pg1661.txt grep -wc Heidelberg || echo "no matches"
```

- The results are

word	output
Holmes	460
a	2287
Baker	42
it	1209
room	168

- We can use the command `wc -w pg1661.txt` or equivalently `< pg1661.txt wc -w` to achieve this task.

Solution to 2.9

- Since the return code of the commands to the left of `&&` or `||` determine if the command to the right is executed, we best look at the command list left-to-right as well:
  - The directory `3/3` does not exist and hence the first `cd` gives return code 1.
  - Therefore `cd 4/2` is executed (`||` executes following command if preceding command has non-zero return code)
  - The command list to the left of the first `&&`, i.e. `cd 3/3 || cd 4/2` has return code 0 (the last command executed was `cd 4/2`, which succeeded)
  - Hence `cd ../4` is executed which fails since the directory `4/4` does not exist below `resources/directories`

  - In other words the list `cd 3/3 || cd 4/2 && cd ../4` has return code 1 and thus `cd ../3` gets executed
  - This succeeds and thus the command to the right of `&&` is executed, i.e. we `cat` the file
- We need to suppress the error messages of the failing `cd` commands. These are `cd 3/3` and `cd ../4`. In other words the shortest commandline would be

```
cd 3/3 2>/dev/null || cd 4/2 && cd ../4 >/dev/null || cd
    ↪../3 &&  cat file
```

- We now first make the directory `3/3`. So the execution changes slightly:
  - `cd 3/3` now succeeds and thus `cd 4/2` is not executed; we are in directory `resources/directories/3/3`.
  - The last command executed was the succeeding `cd 3/3`, such that the return code of the command list `cd 3/3 || cd 4/2` is zero.
  - We attempt to execute `cd ../4`, which fails as the dir `resources/directories/3/4` does not exist.
  - Hence we execute `cd ../3`, which is successful and "changes" the directory to `resources/directories/3/3`.
  - Finally the `pwd` command is also executed, since `cd ../3` was successful.

Solution to 2.10

We easily find out that the commands

```
  kill time fg history pwd exit
```

have documentation which can be accessed using `help` command. This means that they are shell builtins.

Solution to 3.2

This would give a quine:

```
#!/bin/bash
cat $0
```

3_simple_scripts/sol/quine.sh

Solution to 3.3

The solution for the generalised version is:

```
#!/bin/bash

# first print everything non-comment
< "$1" grep -v "%"  > "$2"
```

```

# now everything comment, note the append operator >>
< "$1" grep "%"  >> "$2"
```

3_simple_scripts/sol/comment_move.sh

or alternatively

```
#!/bin/bash

# First copy everything *from* the second line
< resources/matrices/3.mtx tail -n +2 > output.mtx

# Then only the *first* line, again note the append.
< resources/matrices/3.mtx head -n 1 >> output.mtx
```

3_simple_scripts/sol/comment_move_alternative.sh

Solution to 3.4

Since `cat` takes data on *stdin* and copies it to *stdout* without modification, we can cache all data a script gets on *stdin* in a variable `CACHE` using the simple line

```
CACHE=$(cat)
```

Once this has been achieved we just use `echo` to access this data again and grep inside it twice:

```
#!/bin/bash

# Store the keyword we get as first arg:
KEYWORD=$1

# Read every data cat can get on stdin into the
# variable CACHE.
# Since cat's stdin gets fed from the script's stdin,
# this effectively reads the stdin of the script into
# the variable CACHE
CACHE=$(cat)

# Now echo the data again, i.e. transfer the data from the
# variable to stdin of grep.
# Grep for the keyword and store it in the cache again.
CACHE=$(echo "$CACHE" | grep "$KEYWORD")
#
# The above two commands can be done at once using:
# CACHE=$(grep "$KEYWORD")

# We need the so-called quoting here, i.e. the " character
# before and after the parameter expansion for reasons
# explained in the next section.

# Print the first line of the results using head:
echo "$CACHE" | head -n1
```

```

# Print the last line of the results using tail:
echo "$CACHE" | tail -n1

# Now print an empty line
echo

# and now all the matches:
echo "$CACHE"
```

3_simple_scripts/sol/grep_print.sh

Solution to 3.8

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

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

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

3_simple_scripts/sol/ex_quoting.sh

Solution to 3.9

When using `echo "$VAR"| wc -l` the results are

- 2 (correct)
- 1 (correct)
- 1 (wrong, since string is empty)

On the other hand `echo -n "$VAR"| grep -c ^` gives

- 2 (correct)
- 1 (correct)
- 0 (correct)

Therefore this method should be preferred.

Solution to 3.10

A short excerpt of an output of `ls --recursive`:

```
./resources/directories/5:
1␣␣2␣␣3␣␣4␣␣6

./resources/directories/5/1:
file

./resources/directories/5/2:
file

./resources/directories/5/3:
file

./resources/directories/5/4:
file

./resources/directories/5/6:
file
```

It shows the following features:

- Each subdirectory is denoted by the relative path to it
- For each subdirectory we get a list of files it contains.
- Most notably the path of the subdirectories always ends in a "`:`"

If we now assume that no file or directory contains a "`:`" in its name, we can grep for "`:`" in order to get a list of all subdirectories. Since by our assumption no file or dir contains a "`:`" we can use "`cut -d: -f1`" in order to get rid of the tailing "`:`" in the output. A final grep of exactly this output achieves the desired filtering function. Overall we get the script

```bash
#!/bin/bash

# filter in the output of recursive ls
# for a pattern
#
# overall prints those directory paths that match the pattern
ls --recursive | grep ":" | cut -d: -f1 | grep "$1"
```

3_simple_scripts/sol/recursive_ls.sh

Solution to 3.11

The solution makes use of the fact that `grep -n` separates the line number and the text of the matching line by a "`:`". So by sending the output of `grep` to a `cut -d: -f1` we can just extract the numbers of the matching lines.

```bash
#!/bin/bash
FILENAME="$1"
```

```
KEYWORD="$2"

# Grep in the file for each of the keywords
# use the -w flag to only match words
# and the -c flag to only count the matches
# Since grep will return with exit code 1 if no match
# was found, this aborts the script prematurely in this case.
COUNT_KEYWORD=$(grep -cw "$KEYWORD" "$FILENAME") || exit 1

# Grep in the file for each of the keywords again
# now use the -n flag to get the line number of the matches
# use the -w flag to only match words
#
# if one considers the output of grep -n, one notices, that
# the line numbers and the text of the line are
# separated by :
# so using cut we can only extract the line numbers:
LINES_KEYWORD=$(grep -wn "$KEYWORD" "$FILENAME" | cut -d: -f1)
# now each of the former variables contains a list of
# line numbers with matching text

# Now just print the data as requested
echo $COUNT_KEYWORD $LINES_KEYWORD

# The exit 0 is not needed, since a successfully executed script
# always will exit with return code 0
```

3_simple_scripts/sol/grepscript.sh

Solution to 4.1

Here we use the positional parameters `$1` to `$3` and the `[` command in order to achieve our goal:

```
#!/bin/bash

# print the first 3 arguments in reverse:
echo "$3 $2 $1"

# if there is help, we print some help statement:
[ "$1" == "-h" -o "$2" == "-h" -o "$3" == "-h" ] && echo "You asked for some help"

# alternative:
[ "$1" == "-h" ] || [ "$2" == "-h" ] || [ "$3" == "-h" ] && echo "-h passed"
```

4_control_io/sol/arg_reverse.sh

Solution to 4.2

We use `test` to determine the file type and take appropriate action:

```
#!/bin/bash

# $1 is a file and an executable:
[ -f "$1" -a  -x "$1" ] && echo "File␣$1␣is␣executable."

# $1 is just a plain file:
[ -f "$1" ] && echo "File␣$1␣has␣size␣$(<␣"$1"␣wc␣-c)␣bytes."

# $1 is a directory:
[ -d "$1" ] && cd "$1" && ls
```

4_control_io/sol/test_file.sh

Solution to 4.4

The solution requires two nested `while` loops. We use `((ICNT++))` and `((JCNT++))` to increase the counter variables `ICNT` and `JCNT` by one in each iteration.

```
#!/bin/bash

# check if user wants help:
if [ "$I" == "-h" -o "$2" == "-h" -o "$3" == "-h" ]; then
  echo "The␣script␣needs␣two␣integers␣and␣optionally␣a␣file␣name"
  exit 0
fi

# store parameters as I, J and File
I="$1"
J="$2"
FILE="$3"

# check if I and J are not negative:
if [ $I -lt 0 -o $J -lt 0 ]; then
  echo "Both␣$I␣and␣$J␣need␣to␣be␣non-negative␣integers." >&2
  exit 1
fi

ICNT=1  # counter for I

# loop over directories:
while [ $ICNT -le  $I ];do
  # create directory ICNT
  mkdir $ICNT

  JCNT=1  # counter for J
  # loop over files:
  while [ $JCNT -le  $J ];do
    # name of the file to generate
    NAME="$ICNT/$JCNT"

    # if the file exists, we throw an error and exit
    if [ -f "$NAME" ]; then
      echo "The␣file␣$NAME␣already␣exists."
```

```
      exit 1
    fi

    # if user specified a file copy it:
    if [ -f "$FILE" ]; then
      cp "$FILE" "$NAME"
    else
      # else just create a new file
      touch "$NAME"
    fi

    # increase JCNT by one.
    ((JCNT++))
  done

  # increase ICNT by one.
  ((ICNT++))
done
```

4_control_io/sol/while_files.sh

Solution to 4.5

A solution only implementing the 1-argument and 2-argument version of `seq` is:

```
#!/bin/bash

if [ "$1" == "-h" ]; then
  echo "Some␣help"

  # exit the shell with return code 0
  exit 0
fi

# set the default for the 1-argument case:
FROMNUMBER=1  # the number at which the sequence starts
TONUMBER="$1" # the number until the sequence goes

if [ "$#" == 2 ]; then
  # overwrite defaults for 2-argument case:
  FROMNUMBER="$1"
  TONUMBER="$2"
fi

# check the assumptions are not violated:
if ! [ $FROMNUMBER -le $TONUMBER ]; then
  # assumption is violated => exit
  echo "$FROMNUMBER␣is␣not␣less␣or␣equal␣to␣$TONUMBER" >&2
  exit 1
fi

N=$FROMNUMBER
while [ $N -lt $TONUMBER ];do
```

```
  echo $N
  ((N++))
done
echo $N
```

4_control_io/sol/seq2.sh

If the 3-argument version should be supported as well we arrive at:

```
#!/bin/bash

if [ "$1" == "-h" ]; then
  echo "Some␣help"

  # exit the shell with return code 0
  exit 0
fi

#------------
# checking:

# set the default for the 1-argument case:
FROMNUMBER=1   # the number at which the sequence starts
TONUMBER="$1"  # the number until the sequence goes
STEP=1

if [ "$#" -gt 3 ];then
  # we can only consume up to 3 args:
  echo "Extra␣argument" >&2
  exit 1
fi

if [ "$#" -eq 3 ]; then
  # third arg not zero -> 3-arg case
  # overwrite defaults accordingly
  FROMNUMBER="$1"
  STEP="$2"
  TONUMBER="$3"
elif [ "$#" -eq 2 ];then
  # third arg is zero
  #     (otherwise we did not get here)
  # but second is set, hence:
  # overwrite defaults for 2-argument case:
  FROMNUMBER="$1"
  TONUMBER="$2"
  STEP=1
fi

# one arg case is default, but what if $1 is also empty
if [ -z "$1" ] ; then
  echo "Need␣at␣least␣one␣arg" >&2
  exit 1
fi

# check the assumptions are not violated:
```

```
if ! [ $FROMNUMBER -le $TONUMBER ]; then
  # assumption is violated => exit
  echo "$FROMNUMBER␣is␣not␣less␣or␣equal␣to␣$TONUMBER" >&2
  exit 1
fi

#-------------
# do the seq:

N=$FROMNUMBER
while [ $N -lt $TONUMBER ];do
  echo $N

  # do the increment STEP times:
  # using a loop running STEP times:
  I=0
  while [ $I -lt $STEP ]; do
    ((N++))  # increment our number
    ((I++))  # also increment the I
  done
done
echo $N
```

4_control_io/sol/seq3.sh

Solution to 4.6

A script which has the required functionality is:

```
#!/bin/bash

# The directory where the Project Gutenberg books are located:
DIR="resources/gutenberg"

if [ "$1" == "-h" ]; then
  echo "Please␣supply␣an␣argument␣with␣the␣pattern␣to␣search␣for."
  exit 0
fi

if [ -z "$1" ]; then
  echo "Please␣provide␣a␣pattern␣as␣first␣arg"
  exit 1
fi

#-------------------------------------------------------

# go through all gutenberg files
# they all end in txt
for book in $DIR/*.txt; do
  # grep for the pattern
  #    we need the " because $book is a path
  #    and because $1 is user input
  # and store the count in the variable
```

```
  MATCHES=$(< "$book" grep -ic "$1")

  # suppress books without matches:
  [ $MATCHES -eq 0 ] && continue

  # find out the number of lines:
  LINES=$(< "$book" grep -c ^)

  # print it using tabs as separators
  echo -e "$book\t$MATCHES\t$LINES"
done
```

4_control_io/sol/book_parse.sh

Solution to 4.7

One way to achieve the required substitutions is to exploit word splitting. Recall that word splitting takes place at all `<tab>`, `<newline>` or `<space>` characters.

- For the first part we need a commandline that performs word splitting on the content of `VAR` and inserts a `<newline>` between each word. This can be done e.g. by executing

```
for i in $VAR;
  echo $i
done
```

- For the second part we need to insert a `<space>` between all the words after word splitting. This is the effect of

```
echo $VAR
```

  Note that we deliberately leave `$VAR` unquoted here.

Solution to 4.8

We need to use the `while`-`case`-`shift` paradigm in order to parse the commandline in the required way

```
#!/bin/bash

QUIET=0   # are we in quiet mode -> 1 for yes
FILE=     # variable to contain the file

while [ "$1" ]; do
  case "$1" in
    -h|--help)
      echo "Help!!"
      exit 0
      ;;
    -q|--quiet)
      QUIET=1
      ;;
```

```
    -f)
      shift
      FILE="$1"
      ;;
    *)
      echo "Unknown argument: $1" >&2
      exit 1
  esac
  shift
done

# check whether the file is valid
if [ -f "$FILE" ]; then
  echo "File: $FILE"
else
  echo "Not a valid file: $FILE"
  exit 1
fi

# if we are not quiet:
if [ "$QUIET" != "1" ];then
  echo "Welcome to this script"
fi

# exit the script
exit 0
```

4_control_io/sol/commandline_parsing.sh

Solution to 4.13

We use a few calls to `read` in order to read single lines from the script's *stdin*. Then we output the relevant lines to *stdout* or *stderr*.

```
#!/bin/bash
# read first line of stdin
read line

# read second line of stdin
read line

# read third line of stdin
# and print to stdout
read line
echo $line

# read fourth line on stdin
# and print to stderr
read line
echo $line >&2
```

4_control_io/sol/read_third.sh

Solution to 4.14

We use read together with the `-p` (prompt) flag in order to ask the user for two numbers and then use the code from the exercise on page 13 to do the counting and printing:

```
#!/bin/bash

# Ask for two numbers:
read -p "Enter a first number:  " N
read -p "Enter a second number: " M

if [ $N -lt $M ]; then
  START=$N
  END=$M
else
  START=$M
  END=$N
fi

C=$START
while [ $C -lt $END ]; do
  echo $C
  ((C++))
done
echo $C
```

4_control_io/sol/read_seq.sh

Solution to 4.15

The final version of the script could look like this

```
#!/bin/bash

# the default values for from and to
FROM=1
TO="end"

# while-case-shift to parse arguments:
while [ "$1" ]; do
  case "$1" in
    --help)
      echo "Script can take two arguments --from and --to."
      echo "Each have a line number following"
      echo "The script then moves the --from line to the --to line"
      exit 0
      ;;
    --from)
      shift
      FROM=$1
      ;;
    --to)
      shift
```

```
      TO=$1
      ;;
    *)
      echo "Unknown argument: $1" >&2
      exit 1
  esac
  shift
done

if [ "$TO" != "end" ] && [ "$TO" -le "$FROM" ]; then
  echo "The --to line (= $TO) needs to be larger that the --from line (= $FROM)." >&2
  exit 1
fi

# line count
LC=0

# line cache (for the line that should be moved)
CACHE=

# var to keep track if cache is filled or not
# just needed to spot errors more quickly
CACHEFILLED=n

# while read line to read stdin line-by-line
while read line; do
  # increase line count
  ((LC++))

  # if the current line is the from line
  # just store the line in a cache
  if [ $LC -eq $FROM ]; then
    # fill the cache:
    CACHE=$line
    CACHEFILLED=y

    # no printing of this line
    # just continue to next line
    continue

  # if TO is not "end"
  # and it is equal to the current line number
  elif [ "$TO" != "end" ] && [ $LC -eq $TO ]; then
    # check first if we have something in the cache:

    if [ "$CACHEFILLED" != "y" ];then
      # this means some error
      echo "Expected cache to be filled in line $LC" >&2
      echo "This is not the case, however." >&2
      exit 1
    fi

```

```
    # print the cached line
    echo "$CACHE"

    # reset state of the cache
    # just done to spot errors more quickly
    CACHE=""
    CACHEFILLED=n
  fi

  # print current line:
  echo "$line"
  # note that quoting is needed such that
  # characters like tab are kept and not
  # removed by word splitting
done

# we still have something in the cache?
if [ "$CACHEFILLED" != "n" ]; then
  if [ "$TO" == "end" ]; then
    # just print it after everything:
    echo "$CACHE"
    exit 0
  fi

  # if we are getting here this means that
  # the CACHE is still filled even though
  # TO is a number and not "end"
  # so TO is too large:
  echo "The argument supplied to --to (= $TO) is not correct." >&2
  echo "We got less number on stdin than the value given to --to " >&2
  exit 1
fi

exit 0
```

4_control_io/sol/swap_lines_general.sh

Solution to 4.16

The solution just takes 5 lines of bash code:

```
#!/bin/bash
CACHE=
while read line; do
  # insert line by line into the CACHE, but
  # in reverse order.
  # quoting is important here to not loose any
  # newlines due to word splitting
  CACHE=$(echo "$line"; echo "$CACHE")
done
# print the result: Again quoting is needed
```

```
echo "$CACHE"
```

4_control_io/sol/tac.sh

Solution to 4.17

We need to use a slightly modified version of `while read line`, where we pass multiple arguments to `read`:

```
#!/bin/bash
# read line by line and extract the first second
# and third column to BIN, BIN2 and COL.
# Extract all the other columns to TRASH
while read BIN BIN2 COL TRASH; do
  # just print the third column
  echo $COL
done
```

4_control_io/sol/mtx_third.sh

Now if we run our script, redirecting the file `resources/matrices/lund_b.mtx` to its *stdin*

```
    < resources/matrices/lund_b.mtx 4_control_io/sol/mtx_third.sh
```

we realise that it can deal with the multiple spaces which are used in some lines to separate the columns. In other words, compared to `cut` it gives the correct result when the third column of the `mtx` files is to be extracted.

Solution to 4.18

We can achieve exactly what is asked for in a `bash` three-liner:

```
#!/bin/bash

# search for all files using find
# and process them line by line using
# while read line:
find . -type f | while read file; do
  # now grep inside the files
  # we use -n -H in order to keep an overview
  # which file and which lines did match

  grep -n -H "$1" "$file"
done
```

4_control_io/sol/grep_all.sh

Solution to 4.19

Since the directories are separated by a "`:`" in `PATH`, a good `IFS` to use is `:`.

```bash
#!/bin/bash

# we change the field separator to :
OIFS="$IFS"
IFS=":"

# if the user did not provide a command as first arg
# we complain:
if [ -z "$1" ]; then
  echo "Please provide a command as first arg" >&2
  exit 1
fi

# now make use of the new IFS and go through all
# directories in PATH
for dir in $PATH; do
  # does an executable $dir/$1 exist?
  if [ -x "$dir/$1" ];then
    # yes -> we are done
    echo "$dir/$1"
    exit 0
  fi
done
IFS="$OIFS"

# there still has not been an executable found:
exit 1
```

4_control_io/sol/which.sh

Solution to 5.1

The return codes are

- `1` because the assignment `B=0` inside the arithmetic evaluation returns zero, so running `((B=0))` is equivalent to running `((0))`, which is `C`-false. Hence the return code is 1.
- `0` because we just do a simple `echo` of the last value of the arithmetic evaluation `((B=0))`, which is 0. So the command is equivalent to `echo 0`, i.e. it prints "`0`" onto the terminal and exits with return code 0 as well.
- `0`: Here we take the output of `echo $((B=0))` — which is "`0`" — and grep for "`0`" within it. This character is of course is found and hence the return code is 0 again.
- `0`, since `-1` is nonzero, i.e. `C`-true.
- `1`, since 0 is, well zero, which is interpreted as `C`-false.
- `1`, since the last subexpression, i.e. `0` is, well zero.
- `0`, since the last subexpression `3` is nonzero.
- `0`: By just running

```bash
for((C=100,A=99 ; C%A-3 ; C++,A-- )); do echo "C: $C"; echo
```

```
↪"A: $A";done
```

on a shell, we get the output

```
C: 100
A: 99
```

which means that the loop is only run once.

If we look at the 3 fields of the C-like for loop, we see that `A` is initialised to 99 and `C` to 100. After each iteration `C` gets increased by one and `A` gets decreased by one. The iteration stops if `C%A-3` is equal to 0 (C-false), i.e. if

$$\mathrm{C}\%\mathrm{A} = 3$$

This is the case *after* the first iteration, since this gives `C` equal to 101 and `A` equal to 99.

Now we know that the loop body `((B=(B+1)%2))` is only executed once. Since `B` has not been set, it defaults to zero. Executing the statement under arithmetic evaluation hence assigns `B` with

$$(\mathrm{B}+1)\%2 = 1\%2 = 1$$

which is not C-false. Therefore the final `((B))` returns `0`, which is also the return code of the whole expression.

- `1`: `((B=1001%10))` gives rise to no output, such that the first statement

  ```
  ((B=1001%10)) | grep 4
  ```

  fails. Note that `B` is assigned with $1001\%10 = 1$, however.

  We continue with the second statement

  ```
  ((C=$(echo "0"|grep 2)+4, 2%3 ))
  ```

  the command substitution `echo "0"|grep 2` gives rise to no output, hence the resulting string is interpreted as zero. This means that `C` gets assigned to 4. The return code of the statement is determined by `2%3`, which is 2, i.e. the return code is 0.

  We proceed to execute the final statement

  ```
  echo $((4-5 && C-3+B)) | grep 2
  ```

  `4-5` is `-1` and hence C-true and `C-3+B` gives $4-3+1=2$, hence also C-true. In other words `4-5 && C-3+B` is true and `$((4-5 && C-3+B))` is the string "`1`". This means, however, that `grep` cannot find the character 2 in the output and overall the return code of this last expression is `1`.

Solution to 5.2

If one runs the code provided here on the shell, one realises, that for proper integer numbers the result of `echo $((A+0))` and the result of `echo $A` is identical. Exactly this behaviour was used in the following script:

```
#!/bin/bash

# store the first argument in A
A=$1

# check whether it is an integer by the trick
# we just learned about:
if [ "$(( A ))" == "$A" ]; then
  # compute the cube and echo it
  echo "$((A*A*A))"
else
  echo "Argument $1 is not a valid integer." >&2
  exit 1
fi
```

5_variables/sol/cube.sh

Solution to 5.3

One fairly naive solution is

```
#!/bin/bash
N=$1

# check if input is a positive number.
# note that this also checks whether the input is actually an integer
# since strings that cannot be converted to an integer properly are
# interpreted as zero in the following arithmetic evaluation:
if (( N <= 0 )); then
  echo Please provide a positive number as first argument
  exit 1
fi

# have a loop over all integers C less than or equal to N
C=1
while (( ++C <= N )); do
  S=1 # integer we use to test divisibility
  isprime=1  # flag which is 1 if C is a prime, else
             # it is 0
  while (( ++S, S*S <= C )); do
    # loop over all S from 1 to sqrt(C)
    if (( C%S==0 )); then
      # S divides C, hence C is not a prime
      isprime=0

      # break the inner loop: No need to
      # keep looking for divisors of C
      break
    fi
  done

  # if C is a prime, print it
```

```
  (( isprime==1 )) && echo $C
done
```

5_variables/sol/primes.sh

Solution to 5.4

The first version making use of a temporary file can be achieved like this

```
#!/bin/bash

# check that the argument provided is not zeros:
if [ -z "$1" ]; then
  echo "Please provide a pattern as first arg" >&2
  exit 1
fi

# delete the temporary file if it is still here:
rm -f tEMPorary_FIle

# create an empty temporary file
touch tEMPorary_FIle

# call book_parse.sh and analyse resulting table line-by-line
4_control_io/sol/book_parse.sh "$1" |  while read FILE MATCH NUMBER; do
  # read already splits the table up into the 3 columns

  # calculate the xi value:
  XI=$(echo "$MATCH/$NUMBER" | bc -l)

  # echo the xi value followed by a tab and the
  # filename to the temporary file
  echo -e "$XI\t$FILE" >> tEMPorary_FIle
done

# sort the temporary file:
#   -n    numeric sort
#   -r    reverse sort: largest values first
sort -nr tEMPorary_FIle | \
  # print the three higest scoring books
  head -n 3 tEMPorary_FIle

# remove temporary file again:
rm tEMPorary_FIle
```

5_variables/sol/book_analyse.sh

If we want to omit the reading and writing to/from disk, we have to do everything in one pipe. One solution for this could be

```
#!/bin/bash

# check that the argument provided is not zeros:
```

```
if [ -z "$1" ]; then
  echo "Please provide a pattern as first arg" >&2
  exit 1
fi

# call book_parse.sh and analyse resulting table line-by-line
4_control_io/sol/book_parse.sh "$1" |  while read FILE MATCH NUMBER; do
  # read already splits the table up into the 3 columns

  # calculate the xi value:
  XI=$(echo "$MATCH/$NUMBER" | bc -l)

  # echo the xi value followed by a tab and the
  # filename to stdout of the loop
  echo -e "$XI\t$FILE"
done | \
  # sort stdout of the loop
  sort -nr | \
  # filter the first three matches
  head -n 3 | \
  # format the output a little:
  while read XI FILE; do
    echo -e "$FILE  \t(score:\t$XI)"
  done
```

5_variables/sol/book_analyse_notemp.sh

Solution to 5.5

We first parse the arguments and check whether there is anything to do (if there are no numbers supplied, we are done). Then we build up the expression for bc in BCLINE and echo it to bc to get the result.

```
#!/bin/bash

MEAN=n  # if y the mean should be calculated
  # else the sum only

# first arg has to be -s or -m:
case "$1" in
  -m)   MEAN=y
    ;;
  -s) MEAN=n
    ;;
  *)
    echo "Expected -s (for sum) or -m (for mean) as first arg" >&2
    exit 1
esac
shift # remove first arg

if [ -z "$1" ]; then
  # if new first arg, i.e. original second arg is empty
```

```
  # we have no numbers on the commandline
  # hence the result is 0 in both cases:
  echo 0
  exit 0
fi

# We build up the expression for bc in this variable:
# note that we know that $1 is nonzero and we can hence
# initialise BCLINE with it
BCLINE=$1

# count how many numbers we were given:
COUNT=1

# remove the arg we dealt with:
shift

# go over all other arguments
# one by one:
for num in $@; do
  # build up BCLINE
  BCLINE="$BCLINE+$num"
  ((COUNT++))
done

# amend BCLINE if we are caculating the MEAN:
if [ "$MEAN" == "y" ]; then
  BCLINE="($BCLINE)/$COUNT"
fi

# calculate it with bc
# and print result to stdout
echo "$BCLINE" | bc -l
exit $?
```

5_variables/sol/sum_mean.sh

Solution to 5.6

We have to take care to exclude both the first comment line as well as the first non-comment line from being manipulated at all. Apart from these lines all other, however, have to be touched. This script uses a so-called firstrun flag and as well as the `while read line` paradigm to achieve this:

```
#!/bin/bash

NUM=$1

if [ -z "$NUM" ]; then
  echo "Need␣a␣number␣as␣first␣arg." >&2
  exit 1
fi
```

```

# read the comment line and copy to stdout
read line
echo "$line"

# initialise a firstrun flag (see below)
FIRSTLINE=1

# read all remaining data from stdin using grep
# ignore all other comment lines but parse the
# non-comment ones:
grep -v "%" | while read ROW COL VAL; do
  # if this is the first non-comment line
  # then it is special, we have to copy it as is
  if (( FIRSTLINE )); then
    FIRSTLINE=0
    echo "$ROW $COL $VAL"
    continue
  fi

  # for all other rows:
  echo "$ROW $COL $(echo "$NUM*$VAL" | bc -l)"
done
```

5_variables/sol/mtx_multiplier.sh

Solution to 5.7

One solution is:

```
#!/bin/bash

# read stdin line by line:
while read line; do
  # var containing the reversed line:
  LINEREV=""

  # do the reversal in a loop from
  # I=0 to I= length of line -1
  for ((I=0; I < ${#line}; ++I)); do
    # the substring expansion
    #   ${line:I:1}
    # extracts exactly the (I+1)th
    # character from line
    LINEREV="${line:I:1}$LINEREV"
  done
  echo "$LINEREV"
done
```

5_variables/sol/rev.sh

Another solution is:

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

```

# read stdin line by line:
while read line; do
  # do the reversal in a loop from
  # I=0 to I= length of line -1
  LENGTH=${#line}
  for ((I=0; I < LENGTH; ++I)); do
    # Use again a substring expansion
    # but instead we use the index expression
    #      LENGTH-I-1
    # to access the characters from the RHS
    # of the line to the LHS of the line
    echo -n "${line:LENGTH-I-1:1}"
  done
  echo
done
```

5_variables/sol/rev_other.sh

Solution to 6.2

One solution is

```
#!/bin/bash

DIRECTORY=$1
echo "Directory:             largest file"
echo "-----------------------------------"
for f in $DIRECTORY/*; do
  # Only go through directories => skip the '$subdir'
  # if it is not a directory
  [ ! -d "$f" ] && continue

  ( # Subshell for cd
    cd "$f"

    MAXSIZE=0
    MAXFILE="<No file found>"
    for file in *; do
      # This time skip if '$file' is not
      # a valid file.
      [ ! -f "$file" ] && continue

      # The filesize can be determined using
      # wc -c == number of bytes
      SIZE=$(wc -c "$file" | cut -f1 -d " ")
      if [ "$SIZE" -gt "$MAXSIZE" ]; then
        MAXSIZE="$SIZE"
        MAXFILE="$file"
      fi
    done

    # Print the findings
```

```
    echo "$f:    $MAXFILE"
  )
done
```

6_functions_subshells/sol/largest_file.sh

Solution to 6.3

The script contains the following problems:

- Line 10: We alter the `ERROR` flag, which is checked later on to determine if the script execution is to be aborted. This change becomes lost because it happens in a subshell. We should use grouping `{ ... }` instead.
- Line 31: The accumulation of matching lines happens within the implicit subshell started by the pipe. So after the `done`, `MATCHING` is empty again. It is better to fill `MATCHING` directly by a command substitution.
- Line 39: Better use `echo -n "$MATCHING"| grep -c ^` instead of `wc -l` (See exercise).

A better version of the script would be

```
1  #!/bin/bash
2  # initial note:
3  #    this script is deliberately made cumbersome
4  #    this script is bad style. DO NOT COPY
5  KEYWORD=$1
6  
7  ERROR=0    # Error flag
8  [ ! -f "bash_course.pdf" ] && {
9    echo "Please run at the top of the bash_course repository" >&2
10   ERROR=1
11 }
12 
13 # change to the resources directory
14 if ! cd resources/; then
15   echo "Could not change to resources directory" >&2
16   echo "Are we in the right directory?"
17   ERROR=1
18 fi
19 
20 [ $ERROR -eq 1 ] && (
21   echo "A fatal error occurred"
22   exit 1
23 )
24 
25 # List of all matching files
26 # VERSION1: making minimal changes:
27 MATCHING=$(ls matrices/*.mtx gutenberg/*.txt | while read line; do
28   if < "$line" grep -q "$KEYWORD"; then
29     echo "$line"
30   fi
31 done)
```

```

# VERSION2: Even more simple and more reliable
MATCHING=$(for line in matrices/*.mtx gutenberg/*.txt; do
  if < "$line" grep -q "$KEYWORD"; then
    echo "$line"
  fi
done)

# count the number of matches:
COUNT=$(echo -n "$MATCHING" | grep -c ^)

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

6_functions_subshells/sol/subshell_exercise_corrected.sh

Solution to 6.4

We use the function `list_files` that deals with a directory and all subdirectories recursively. A little care has to be taken when printing the paths such that the "/" appears at the right places.

```
#!/bin/bash

list_files() {
  #$1: prefix to append when listing the files
  DIR="$1"

  # deal with all files in current directory:
  for file in *; do
    # file is a regular file => list it
    if [ -f "$file" ]; then
      # print prepending prefix
      echo "$DIR$file"
    elif [ -d "$file" ]; then
      # file is a directory:
      # recursively call this fctn:
      (
        # go into subshell
        # this keeps track of
        # the working directory
        cd "$file"
        list_files "$DIR$file/"
      )
    fi
    # do nothing for all other types of
    # files
  done
```

```
}
```

6_functions_subshells/sol/find_file.sh

Solution to 6.5

Instead of using one single line with all commands, we use functions to split the tasks up into logical parts and name these parts sensibly.

```
#!/bin/bash

# check that the argument provided is not zeros:
if [ -z "$1" ]; then
  echo "Please␣provide␣a␣pattern␣as␣first␣arg" >&2
  exit 1
fi

calculate_xi() {
  # analyse the output from book_parse.sh
  # calculate the xi values and print a table
  # of xi values followed by a tab and the filename to stdout

  while read FILE MATCH NUMBER; do
    # read already splits the table up into the 3 columns

    # calculate the xi value:
    XI=$(echo "$MATCH/$NUMBER" | bc -l)

    # echo the xi value followed by a tab and the
    # filename to stdout of the loop
    echo -e "$XI\t$FILE"
  done
}

filter_3_largest() {
  # filter the output of calculate_xi such that only the 3
  # books with the largest xi values are passed from stdin
  # to stdout

  # sort stdin and filter for first 3 matches
  sort -nr | head -n 3
}

print_results() {
  # Take a table in the format produced by calculate_xi and
  # print the rows is a formatted way

  while read XI FILE; do
    echo -e "$FILE␣␣\t(score:\t$XI)"
  done
}

#----------------------------------------------------------
```

```
4_control_io/sol/book_parse.sh "$1" | \
  calculate_xi | filter_3_largest | print_results
```

6_functions_subshells/sol/book_analyse_fun.sh

Solution to 6.6

After the `subtract` operation has been implemented as well, we arrive at

```
#!/bin/bash

# global variable SEL to make selection between
# addition and multiplication
SEL=

#--------------------------------------------------

add() {
  # add two numbers
  # $1: first number
  # $2: second number
  # echos result on stdout
  echo $(($1+$2))
}

multiply() {
  # multiply two numbers
  # $1: first number
  # $2: second number
  # echos result on stdout
  echo $(($1*$2))
}

subtract() {
  # Subtract two numbers
  # $1: first number
  # $2: number subtracted from first number
  # echos result on stdout
  echo $(($1-$2))
}

operation() {
  # selects for add or multiply depending on
  # SEL
  # $1: first operand for operator (add or multiply)
  # $2: second operand for operator (add or multiply)
  # echos the result on stdout

  # this will call add if $SEL == "add"
  # or it will call multiply if $SEL == "multiply"
  # or subtract if $SEL == "subtract"
  local FIRST=$1
```

```
  local SECOND=$2
  $SEL $FIRST $SECOND
}

calculate3() {
  # it calls operation with 3 and $1
  # such that we either add, subtract or multiply (depending on SEL) 3 and $1
  # echos the result on stdout

  operation $1 3
}

map() {
  # $1: a command

  local COMMAND=$1
  shift

  # loop over all arguments left on the commandline
  # and execute the command in COMMAND with this
  # arguement
  for val in $@; do
    $COMMAND $val
  done
}

usage() {
  echo "$0  [ -h | --help | --add3 | --multiply3 ]  <arguments> "
  echo "Script to do some operation to all arguments"
  echo
  echo "Options:"
  echo "--add3          adds 3 to all arguments"
  echo "--multiply3     multiplies 3 to all arguments"
  echo "--subtract3     subtracts 3 from all arguments"
}

#--------------------------------------------------

# $1 selects method

case "$1" in
  --help|-h)
    usage
    exit 0
    ;;
  --add3)
    SEL=add
    ;;
  --multiply3)
    SEL=multiply
    ;;
  --subtract3)
```

```
    SEL=subtract
    ;;
  *)
    echo "Unknown␣argument:␣\"$1\"" >&2
    echo "Usage:␣" >&2
    usage >&2
    exit 1
esac

# remove the first arg we dealt with
shift

# deliberatly no quotes below to get rid of linebreak
# in the results:
echo $(map calculate3 $@)
```

6_functions_subshells/sol/functional.sh

It takes very little effort to add extra operators, since the script only needs to be changed at two places: We need to add the function and we need to add an extra case in order to get `SEL` set accordingly.

One could go even further: The functions `add`, `multiply` and `subtract` are very similar. So one could use the tool `eval` in order to write a generating function which automatically defines these aforementioned functions. Then we arrive at

```
#!/bin/bash

# global variable SEL to make selection between
# addition and multiplication
SEL=

#--------------------------------------------------

generator() {
  # function to generate a function that takes
  # two numbers and echos the result of applying
  # an operation to these numbers on stdout
  #
  # $1: name of the function to generate
  # $2: operator to use in the operation
  NAME=$1
  OP=$2

  eval "${NAME}()␣{␣ONE=\$1;␣TWO=\$2;␣echo␣\$((ONE␣$OP␣TWO));␣}"
}

generator "add" "+"        # generate add function
generator "multiply" "*"   # generate multiply
generator "subtract" "-"   # generate subtract

operation() {
  # selects for add or multiply depending on
  # SEL
  # $1: first operand for operator (add or multiply)
```

```
  # $2: second operand for operator (add or multiply)
  # echos the result on stdout

  # this will call add if $SEL == "add"
  # or it will call multiply if $SEL == "multiply"
  # or subtract if $SEL == "subtract"
  local FIRST=$1
  local SECOND=$2
  $SEL $FIRST $SECOND
}

calculate3() {
  # it calls operation with 3 and $1
  # such that we either add, subtract or multiply (depending on $SEL) 3 and $1
  # echos the result on stdout

  operation $1 3
}

map() {
  # $1: a command

  local COMMAND=$1
  shift

  # loop over all arguments left on the commandline
  # and execute the command in COMMAND with this
  # arguement
  for val in $@; do
    $COMMAND $val
  done
}

usage() {
  echo "$0  [ -h | --help | --add3 | --multiply3 ]  <arguments> "
  echo "Script to do some operation to all arguments"
  echo
  echo "Options:"
  echo "--add3          adds 3 to all arguments"
  echo "--multiply3     multiplies 3 to all arguments"
  echo "--subtract3     subtracts 3 from all arguments"
}

#--------------------------------------------------

# $1 selects method

case "$1" in
  --help|-h)
    usage
    exit 0
    ;;
```

```
  --add3)
    SEL=add
    ;;
  --multiply3)
    SEL=multiply
    ;;
  --subtract3)
    SEL=subtract
    ;;
  *)
    echo "Unknown argument: \"$1\"" >&2
    echo "Usage: " >&2
    usage >&2
    exit 1
esac

# remove the first arg we dealt with
shift

# deliberatly no quotes below to get rid of linebreak
# in the results:
echo $(map calculate3 $@)
```

6_functions_subshells/sol/functional_generator.sh

Note, however, that `eval` is a dangerous command and should never be used on anything that contains data, which the user of your script can set. In other words: Only use it if you know what it does and how it works!

Solution to 6.7

In order to make the script from the other exercise sourcable, we just need to insert the code

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

before the `case` statement, e.g. in line 80 (of the version not using the `generator`). The script, which is sourcable, can be found in `6_functions_subshells/sol/functional_sourcable.sh`. Note that it still can be executed normally and runs as expected.

If we want to use `functional_sourcable.sh` in the script `6_functions_subshells/source_exercise.sh`, we need to change it slightly:

```
#!/bin/bash

# check first if sourced script exists:
if [ ! -f 6_functions_subshells/sol/functional_sourcable.sh ]; then
  echo "This script only works if executed from the top directory" >&2
  echo "of the tarball containing the solution scripts." >&2
  echo "Please change the working directory accordingly and" >&2
  echo "execute again." >&2
  exit 1
```

```
fi

# source the other script
. 6_functions_subshells/sol/functional_sourcable.sh

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

# multiply 6 and 7 and print result to stdout:
multiply 6 7
```

6_functions_subshells/sol/source_exercise_amended.sh

Due to the relative path to the sourced script we used in this modified version of `6_functions_subshells/ source_exercise.sh`, the script only works if executed from the top directory of the tarball, which contains the solution scripts.

Solution to 7.2

The matching part:

- `..` matches any string that contains any two character substring, i.e. any string with two or more letters. This is everything except `g` and the empty string.
- `^..$` matches a string with exactly two characters, i.e. `ab` and `67`.
- `[a-e]` matches any string that contains at least one of the characters `a` to `e`, i.e. `ab` and `7b7`.
- `^.7*$` matches any string which starts with an arbitrary character and then has zero or more 7s following. This is `g`, `67`, `67777`, `7777` and `77777`.
- `^(.7)*$` matches any string which has zero or more consecutive substrings consisting of an arbitrary character and a 7. This is `67`, `o7x7g7`, `7777` and the empty string. Note that e.g. `77777` does not match: If we "use" the pattern `.7` three times we get `^.7.7.7$` and `77777` has one character too little to be a match for this.

Solution to 7.3

The crossword:

	`a?[3[:space:]]+b?`	`b[^eaf0-2]`
`[a-f][0-3]`	a3	b3
`[[:xdigit:]]b+`	3b	bb

Solution to 7.4

a) `ab*c` or `c$` or `c`

b) `ab+c` or `bc$` or `bc`

c) `^a.*c` or `c$`

d) `^␣*q` or `q..`

e) `^a|w` or `....` or `r|w`

Solution to 7.5

- A single digit can be matched by the regex `[0-9]`.
- The list of digits we get by running

```
< resources/digitfile grep -o '[0-9]'
```

  is just the list of all digits which are contained in the file `resources/digitfile` in exactly the order they occur.
- We can run

```
< resources/digitfile grep -o '[0-9]' | sort -n | uniq -c
```

  to get the required table

```
␣␣␣␣␣␣1␣0
␣␣␣␣␣␣7␣1
␣␣␣␣␣␣2␣2
␣␣␣␣␣␣3␣3
␣␣␣␣␣␣3␣4
␣␣␣␣␣␣7␣5
␣␣␣␣␣␣1␣7
␣␣␣␣␣␣3␣9
```

  In other words there is 1 zero, 7 ones, 2 twos, . . . , 3 nines.

Solution to 7.6

- Regexes for the parts:
  - sign: "`[+-]`"
  - prefactor: "`[01]\.[0-9]*`"
  - exponent: "`[0-9]+`"
- So altogether the scientific numbers need to match:

```
([+-]?)([01]\.[0-9]*)e([+-]?)([0-9]+)
```

  where the parenthesis `( )` are only provided to show the individual parts, i.e.

```
[+-]?[01]\.[0-9]*e[+-]?[0-9]+
```

  would be valid as well. Executing this on the `digitfile` gives

```
$ < resources/digitfile grep -E '[+-]?[01]\.[0-9]*e[+-]?[0-9]+'
```

```
1.759e+15
1.5e+5da␣is␣a␣scientific␣number
-1.34e+04
```

- Introducing the fault tolerance implies:
  - We replace the plain requirement for “e” by the bracket expansion “[eEdD]”.
  - Instead of “[01]\.[0-9]*”, we require a number with an optional decimal part, i.e. “[0-9]+(\.[0-9]*)?”

  Hence overall

```
[+-]?[0-9]+(\.[0-9]*)?[eEdD][+-]?[0-9]+
```

```
$ < resources/digitfile grep -E
  ↪'[+-]?[0-9]+(\.[0-9]*)?[eEdD][+-]?[0-9]+'
```

```
1.759e+15
-9.3e-5
19e-5␣is␣not␣properly␣formatted␣either.
1.5e+5da␣is␣a␣scientific␣number
-1.34e+04
```

Solution to 7.7

By running

```
< resources/matrices/bcsstm01.mtx grep -E
  ↪'[+-]?[0-9]+(\.[0-9]*)?[eEdD][+-]?[0-9]+'
```

we can easily verify that the proposed pattern gives indeed the values in the third column. As usually we get the largest of these values by piping the result to `sort -r -n | head -n1`:

```
< resources/matrices/bcsstm01.mtx grep -E
  ↪'[+-]?[0-9]+(\.[0-9]*)?[eEdD][+-]?[0-9]+' | sort -r | head
  ↪-n1
```

Solution to 7.8

The whole problem can be solved using the command lines

```
<  resources/chem_output/qchem.out head -n48 > file
< file sed -r '/Q-Chem/d; s/[A-Z]\.-?//g; s/,/\n/g' | sed
  ↪'s/^[[:space:]]*//; /^$/d' | sort
```

The sed commands in more detail:

- `/Q-Chem/d`: Delete all lines containing `Q-Chem`

- `s/[A-Z]\.-?//g`: Replace all initials by nothing. Since sed tries to match as much as possible, the `-?` makes sure that first names with a "`-`" are removed completely as well. E.g.

```
T.-C.
```

  gets replaced by the empty string by the means of two substitutions in this step.

- `s/,/\n/g`: All commas get replaced by a line break.
- `s/^[[:space:]]*//`: Replace leading whitespace by nothing, i.e. remove it.
- `/^$/d`: Remove empty lines.

Note that we need two sed invocations here because sed does take proper note of the extra line break we introduce with the substitution `s/,/\n/g`. This can be explained as follows:

sed processes all rules for each line going from top to bottom, right to left. So even though we introduce new line breaks by the substitution, sed considers the resulting string still as a logical line and all regexes are applied to the logical line instead of the actual lines. Using such the procedure, which was suggested by the exercise, we cannot deal with this in any other way but piping it to another sed, which now honours the new line breaks.

Note that a careful inspection of the problem reveals that the one-liner

```
< file sed -r '/Q-Chem|^$/d; s/( *[A-Z]\.-? ?|, *$)//g; s/,/\n/g'
```

does the trick as well, just using a single sed.

Solution to 8.2

One possible solution is:

```bash
#!/bin/bash

if [ ! -r "$1" ]; then
  echo "Cannot read file: $1" >&2
  exit 1
fi

< "$1" awk '{ print $2 " + " $3 " = " $2+$3 }'
```

8_awk/sol/print_add.sh

If we execute this like

```
< resources/matrices/3.mtx 8_awk/sol/print_add.sh
```

or like

```
< resources/matrices/lund_b.mtx 8_awk/sol/print_add.sh
```

it prints the correct results, thus dealing well with the multiple separators in `resources/matrices/lund_b.mtx`.

Solution to 8.3

We use `echo` in order to transfer the numbers to `awk`, let `awk` do the computation and print the result on *stdout* (straight from `awk` itself):

```
#!/bin/bash

# global variable SEL to make selection between
# addition and multiplication
SEL=

#--------------------------------------------------

add() {
  # add two numbers
  # $1: first number
  # $2: second number
  # echos result on stdout
  echo "$1 $2" | awk '{ print $1+$2 }'
}

multiply() {
  # multiply two numbers
  # $1: first number
  # $2: second number
  # echos result on stdout
  echo "$1 $2" | awk '{ print $1*$2 }'
}

subtract() {
  # Subtract two numbers
  # $1: first number
  # $2: number subtracted from first number
  # echos result on stdout
  echo "$1 $2" | awk '{ print $1-$2 }'
}

operation() {
  # selects for add or multiply depending on
  # SEL
  # $1: first operand for operator (add or multiply)
  # $2: second operand for operator (add or multiply)
  # echos the result on stdout

  # this will call add if $SEL == "add"
  # or it will call multiply if $SEL == "multiply"
  # or subtract if $SEL == "subtract"
  local FIRST=$1
  local SECOND=$2
  $SEL $FIRST $SECOND
}

calculate3() {
  # it calls operation with 3 and $1
```

```
  # such that we either add, subtract or multiply (depending on SEL) 3 and $1
  # echos the result on stdout

  operation $1 3
}

map() {
  # $1: a command

  local COMMAND=$1
  shift

  # loop over all arguments left on the commandline
  # and execute the command in COMMAND with this
  # arguement
  for val in $@; do
    $COMMAND $val
  done
}

usage() {
  echo "$0  [ -h | --help | --add3 | --multiply3 ]  <arguments> "
  echo "Script to do some operation to all arguments"
  echo
  echo "Options:"
  echo "--add3          adds 3 to all arguments"
  echo "--multiply3     multiplies 3 to all arguments"
  echo "--subtract3     subtracts 3 from all arguments"
}

#--------------------------------------------------
# make script sourcable:
return 0 &>/dev/null
#--------------------------------------------------

# $1 selects method

case "$1" in
  --help|-h)
    usage
    exit 0
    ;;
  --add3)
    SEL=add
    ;;
  --multiply3)
    SEL=multiply
    ;;
  --subtract3)
    SEL=subtract
    ;;
  *)
```

```
    echo "Unknown␣argument:␣\"$1\"" >&2
    echo "Usage:␣" >&2
    usage >&2
    exit 1
esac

# remove the first arg we dealt with
shift

# deliberatly no quotes below to get rid of linebreak
# in the results:
echo $(map calculate3 $@)
```

8_awk/sol/functional_awk.sh

Solution to 8.4

The BEGIN rule initialises the variable `c` to zero, which is the default anyway. Therefore it can be omitted.

```
#!/bin/bash
awk '
  # BEGIN { count=0 }
  { count=count + 1 }
  END { print count }
'
```

8_awk/sol/awk_wc.sh

Solution to 8.5

One possible solution is:

```
#/bin/bash
awk '
  # initialise inside_table
  # the flag we use to keep track whether we are inside or outside
  # of a Davidson table
  BEGIN { inside_table=0 }

  # whenever we encounter the "  Starting Davidson", we
  # change the flag to indicate that we are inside the table.
  /^[[:space:]]*Starting Davidson/ { inside_table=1; itercount=0 }

  # save current iteration number in itercount
  # but only if the first field contains a digit.
  # and we are not in the "Guess" step
  $1 ~ /[0-9]/ && inside_table == 1 && $0 !~ /Guess/ {
      itercount=$1 }

  # here the last stored itercount is the actual number
  # of iteration steps performed
```

```
  # we print it and reset the flag to 0
  /^[[:space:]]*Davidson Summary:/ { inside_table=0; print itercount }
'
```

8_awk/sol/davidson_extract.sh

Solution to 8.9

The `uniq` command can be implemented like this:

```
#!/bin/bash
awk '
  $0 != prev { print $0; }
  { prev=$0 }
'
```

8_awk/sol/awk_uniq_2line.sh

or alternatively one line shorter:

```
#!/bin/bash
awk '
  $0 != prev {print $0; prev=$0 }
'
```

8_awk/sol/awk_uniq.sh

A solution, which implements `uniq -c` is

```
#!/bin/bash
awk '
  # initialise prev on first run:
  prev == "" { prev=$0; c=1; next }

  # current line is same as previous:
  # increase counter by one:
  $0 == prev { c++ }

  # current line is not same as previous:
  # print the statistics for previous line and
  # reset prev and c
  $0 != prev {print c " " prev; prev=$0; c=1 }

  # print statistics for the last set of
  # equal lines:
  END { print c " " prev }
'
```

8_awk/sol/awk_uniqc.sh

Solution to 8.10

One possible solution is:

```
#/bin/bash
awk '
  # initialise inside_block
  # the flag we use to keep track whether we are inside or outside
  # an excited states block
  BEGIN { inside_block=0 }

  # whenever we encounter the "  Excited state ", we
  # change the flag to indicate that we are inside the table.
  # also we store the state number, which sits in the third field
  /^ *Excited state [0-9]+/ { inside_block=1; state_number=$3 }

  # if we find the "Term symbol" line inside the block, we store
  # the term symbol which sits in $3 $4 and $5
  inside_block==1 && /^ *Term symbol/ { term_symbol=$3 " " $4 " " $5 }

  # if we find the "Excitation energy" line, we store the excitation energy
  # and print the table, since we do not care about the rest of the
  # block. Next we reset the inside_block flag for the next block to come.
  inside_block==1 && /^ *Excitation energy/ {
    excitation_energy=$3

    # print the data tab-separated (for analysis with e.g. cut)
    print state_number "\t" term_symbol "\t" excitation_energy

    inside_block=0
  }
'
```

8_awk/sol/exstates_extract.sh

Licensing and redistribution

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

Michael F. Herbst. Advanced bash scripting 2017. November 2017. URL http://doi.org/10.5281/zenodo.1045332.