UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Viktoria Plemakova

Teaching advanced Bash scripting using semi-automated assessment environment

Bachelor's Thesis (9 ECTS)

Supervisor: Artjom Lind, MSc

Tartu 2016

Edasijõudnud Bash skriptimise õpetamine kasutades pool-automatiseeritud hindamiskeskkonda

Lühikokkuvõte: Bash on Unix'i käsuinterpretaator ning skriptimiskeel. Käesolev töö keskendub ülesannetekogu ning testimismehhanismi implementeerimisele progressiivse Bash skriptimise õpetamiseks. Antud hetkel puuduvad paljudes ülikoolikursustes materjalid, mis kasutavad õpetamisel Linux'i keskkonda. Antud lähenemine üritab muuta skriptimise õppimise efektiivsemaks ning erineda programmeerimisainetele omastest lähenemistest. Põhiosa teadustööst keskendub ülesanneteplokkide struktuurile, kolme ploki implementeerimisele (*for* ja *while* tsüklid, tingimuslaused) ja kontrollskriptile. Kokkuvõttes vajab antud süsteem veel testimist, kuid on üliõpilastele Bash skriptimise õpetamiseks paljulubav automaatõppesüsteem. Tulevikus on võimalik täiendada olemasolevat tulemust uute ülesannete juurde tegemisega ning testskripti muutmisega. Selleks, et mõista, millised on õpilaste nõrkused Bash skriptimise alal, tuleks läbi viia täiendavaid katseid.

Võtmesõnad: Unix, Linux, Bash, skriptimine, automaattestid

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia

Teaching advanced Bash scripting using semi-automated assessment environment

Abstract: Bash is a Unix command-line interpreter as well as a scripting language. This thesis focuses on implementing exercise sets and a testing mechanism for teaching progressive Bash scripting. Right now there is a lack of such material in several university courses that use Linux environment in teaching. The approach attempts to make learning scripting more effective and make it different from the approaches used in several programming courses. The main part of this thesis concentrates on the structure of the exercise sets, implementation of three sets (for and while loop, conditionals) and the validation script. In the end, the presented system lacks real life testing but shows great promise as an automated learning system for teaching university students Bash scripting. In the future, the existing work could be improved by generating more exercises and modifying the tester. To get an idea of students' weaknesses concerning Bash, an intensive testing should be carried out.

Keywords: Unix, Linux, Bash, scripting, automated testing

CERCS: T120 Systems engineering, computer technology

Contents

List of Figures

List of Listings

1 Introduction

Bash is one of the core components of many Linux distributions and therefore it is vital to know how to use it, what benefits it has and how to best pass this information to the students. However, teaching Linux fundamentals is often not compulsory in neither some high school nor university curricula. Yet, some university courses expect students to have basic Bash skills. As the academical background is different for each student, some courses such as System Administration, Operating Systems, Distributed Systems and Data Security at the University of Tartu would benefit from a material on Bash.

At the moment, the University of Tartu does not offer any courses that would entirely focus on teaching command line and Bash scripting. Creating such course that would also have good tutorials and exercise sets is time consuming. The exercises created for this bachelor thesis could serve as a basis for creating an online course in the future for students to learn Bash at their own pace. This could be achieved by introducing automated grading system which would give learners feedback on their progress and speed up the learning progress.

Above all, this thesis aims to help students that need to enhance their Bash skills. The exercises presented in this thesis were created to teach students how to use Bash scripting to their advantage. Being able to write good scripts is in essence far more than executing simple commands. Writing a Bash script allows students to combine simple commands that they already know like `ls`, `cp` or `grep` with loops, conditionals etc. to create a far more complex solution. Moreover, scripting gives users the control over the customization of the outputs. In brief, these exercises are designed in order to teach students to take simple commands and combine them in a way that creates something more than a simple command alone could produce.

It is crucial that students start writing scripts from the beginning. To achieve that, the exercises created for this thesis cannot be skipped. In order to fully complete all the exercises, the previous ones have to be done first as they contain certain information which makes completing the task possible. Final results can be read from the summary.

In Chapter 2 Bash shell and scripting are introduced. In addition, Chapter 2 gives an overview of different approaches that are used when teaching Bash to university students. Chapter 3 focuses on explaining the structure of the exercises created for this thesis and describes the techniques used to create the tasks. The assessment of students' scripts is discussed as well. Chapter 4 concentrates on the solution design. In Chapter 5 results are presented.

2 Related Work

This chapter describes the Bash shell and scripting in Bash. In addition, some methods that have been used for teaching Bash command line and scripting at various universities are introduced.

2.1 Bash

In section 2.1.1 the overview of the Bash shell is given and in section 2.1.2 Bash shell scripting is described.

2.1.1 Overview of Bash shell

Bash (short for 'Bourne-Again SHell') is a Unix command-line interpreter that is used as a standard shell on many Linux distributions [5] as well as on Apple's OS X. Bash was initially developed by Brian Fox of the Free Software Foundation as a GNU Project's shell and now it is maintained by Chet Ramey of Case Western Reserve University [6].

Bash is similar to the original Bourne shell (`sh`) which means that most of the scripts written in `sh` can be executed in Bash shell as well [7]. In addition to some features from the Korn shell and the C shell [8], Bash contains improvements over the previous shells that ease the use of command line and shell scripting [6].

Bash features [6, 9, 10], for example, include:

1) command history. Every entered command is recorded which lets users to execute it again or run a modified version of the last command. This feature also allows to go back and forth in the command history, search for a specific command and save the command history;
2) command line editing. This feature allows users to move back and forth through the command line and edit any mistakes without actually erasing half of the line. Users may also change editing key bindings for their liking;
3) job control. Processes can be suspended and restarted as well as moved to the foreground or background;
4) aliases;
5) prompt customisation;
6) directory stack;
7) file and directory name completion. Using the TAB key users can complete command or file names having typed only several letters. If there are more than one completion available then a list of all possibilities is provided;

8) additional variable expansions;

9) shell arithmetic.

2.1.2 Bash shell scripting

A Bash shell script is a plain text file (usually with a `.sh` extension) which contains shell commands. Any command that is generally executed on the command line can be added to a script. It does not change how a certain command performs, whether it is run from the command line or interpreted from a script. Apart from commands, a shell script can contain conditionals, variables, loops and functions as well.

Shell scripting is widely used in system administration. Having a script for certain tasks saves time as there is no need to retype same commands every time a user needs something to be done. For example, a script can be used to perform a daily backup, clean up log files [11] or install a program.

Writing a Bash script

There are a few things to keep in mind when writing a Bash script and these will be explained using the example script found in listing 1.

```
1 #!/bin/bash
2 # Following line will display "Hello, world!" in the terminal
3 echo "Hello, world!"
```

Listing 1: Sample "Hello, world!" script

The header of the script should contain the character sequence that is called the shebang (`#!`). The hash and the exclamation mark are followed by the absolute path to the interpreter program which could be a shell, a programming language, or a utility [11]. In listing 1, it is the path to Bash (`/bin/bash`).

Unless Bash is set as the default shell, shebang is needed in the heading of the script [11]. It is still advised to add a valid shebang line to the beginning of every script. Executing a Bash script under another shell might cause problems as the script might include some "shell-specific constructs" according to the Advanced Bash-Scripting Guide [11].

The specified interpreter executes all the lines in the script that follow the shebang line and ignores any comments (lines starting with a hash character). In

the example script, the second line is ignored because it is a comment. In the last line, the `echo` command is executed and then the script exits.

Permissions and executing a script

To run a script, it has to be executable. In order to achieve that, we need to give *execute* permission to the script using the `chmod` (change mode) command like shown in listing 2. The script also needs *read* permission to allow the shell to read it [11].

```
viktoria@Byron:~$ ls -l helloworld.sh
-rw-rw-r-- 1 viktoria viktoria 0 apr    20 14:32 helloworld.sh
viktoria@Byron:~$ chmod u+x helloworld.sh
viktoria@Byron:~$ ls -l helloworld.sh
-rwxrw-r-- 1 viktoria viktoria 0 apr   20 14:32 helloworld.sh
```

Listing 2: Making the script executable

First line in listing 2 shows that the script is not executable yet. The command on the second line of the example allows only the owner to execute the file `helloworld.sh`. Last line in listing 2 shows that the execute permission bit was granted to the file. To allow anyone to run the script `+x` or `a+x` should be used in the command where `a` means *all*. Finally, the script can be run by typing `./helloworld.sh` in the same directory where it was created or by using the absolute path to the script.

Users cannot run the script by just using its name because the directory which contains the script is not included in the `$PATH` variable by default for security reasons [11]. However, it is possible to move the script to the `/usr/local/bin` to make it available for everybody and run the script by just typing its name [11]. Alternatively, users can create a `bin` directory in their home directory and add it to the `$PATH` variable. After moving the script into the newly created directory, it can be executed by its name in any directory.

2.2 Teaching Bash

In section 2.2 various approaches of teaching Bash at other universities and the university of Tartu are provided and discussed.

2.2.1 Workstation Software course

University of Tartu used to have a course called Workstation Software (*Tööjaamade tarkvara* in Estonian) [12] taught by Marti Taremaa. One of the course objectives was to prepare students for other courses where Unix knowledge is a prerequisite [13]. The course included topics like general Unix history, command line, shell programming (`bash\sh`), X Window System, `awk` and Tcl\Tk.

Each lab material included some code examples, relevant references to different manuals and some exercises to practice. Figure 1 contains an example of a lab task where students are asked to check when did they last log in, what does the `last reboot` command do and compare it to the output from the `uptime` command. There were three homework assignments during the course: a Bash or `sh` script, an `awk` script and a Tcl\Tk application. All three assignments had to be submitted to pass the course [12].

> **Harjutus.** Vaata käsuga `last kasutajanimi`, millal oled viimati sellesse arvutisse sisse loginud (kui oled). Mida näitab käsk `last reboot`? Võrdle saadud infot käsu `uptime` tulemusega.

Figure 1: Example exercise from the Workstation Software (WS) course lab [1]

For each assignment students had to choose one task from a given list of about 30–40 different options [2], thus every student had a unique exercise to solve. This possibly ensured that students did not copy each other's scripts and gave them an opportunity to pick an assignment that seemed interesting and/or easy for them. Nonetheless, students were allowed to use some code from other sources as long as they mentioned that in their work [2]. Assignments assumed that scripts were non-interactive unless some task stated differently [2]. Students had to show and explain their homework to teaching assistants to get it graded and perform any necessary edits if needed [2].

Figure 2 shows an example of a shell scripting homework assignment. The script should take in a file extension and a name of a subdirectory as arguments. Then it moves all the files with the given extension from the current directory to the right subdirectory. It should also be possible to provide more than one pair of arguments. Other available assignments were of seemingly similar difficulty.

12. Koostada programm, mis liigutaks etteantud kataloogis olevad failid etteantud laiendite järgi alamkataloogidesse, jättes ülejäänud puutumata. Nii laiendid kui ka alamkataloogide nimed peaks saama paaridena ette anda.

```
N:\\>sort_files.sh txt tekstidokumendid java programmeerimine
```

Figure 2: Example homework assignment from the WS course [2]

The course was discontinued after the autumn semester in 2009 although it would have been a great prerequisite for several other courses where the course work depends on knowing how to use Linux systems. The Workstation Software course provided students with necessary and sufficient Unix skills which is why having a similar course anew would be useful.

2.2.2 Treasure hunt

Unix is considered complex for some newcomers which is why Matthieu Moy [14] finds it important to teach it in a way that does not scare new users away. His way of introducing Unix to beginners is letting them explore the command line and applications by completing a series of exercises where one task has to be solved before being able to access the next one (a so-called treasure hunt). Users are provided with a textbook [15] and a wiki [16]. The idea behind this method seems to be that the learning progress would be entertaining to newcomers which in order would help them learn new things better through this experience.

The exercises in the learning game include, for example, some essential Unix commands and tools, the ability to use a text editor and more. As mentioned before, users have to solve one exercise to see the next one. It seems a reasonable restriction: unless user understood certain topics s/he cannot move on to the next task. When using/learning certain commands, it is important to understand what it does and why it works like that. For that, the author's textbook [15] provides some smaller exercises to complete before proceeding to the main task of the level.

Good. This concludes part E of the treasure hunt.

The next step is located in directory F1 of this directory (in the same archive as `step-E13.tar.gz`).

This is the only file whose name starts with an 'x', and ends with a character different from 'z'. The solution is very simple once you know what a wildcard is (the booklet will teach you if you do not know already).

Figure 3: Treasure hunt task example [3]

The main topics of the textbook [15] cover some basic Unix commands and tools, and Bash. The author claims that the textbook aims to be better over other existing Unix tutorials. Even though, one could find a beginner's guide to Unix on the internet and the content would not differ that much from the textbook. Naturally, not every source provides exercises, but in general, there are as detailed introductions to Unix as this textbook.

Shell scripting was mentioned as a bonus level as it was considered an advanced topic. The author's goal was to introduce new users to Unix, so making this topic optional is in some way understandable. Scripting can sometimes get complicated and, in addition, is a large topic on its own. After just learning some day-to-day commands, shell scripting might seem a bit overwhelming, most likely particularly for those who struggled with the treasure hunt exercises. However, there still could have been some examples in the main part of the course to show what can be done when users become more experienced and comfortable with Unix commands. There were, though, links to tutorials and books such as Advanced Bash-Scripting Guide [11] on the wiki page [17].

The topics pointed out in the Matthieu Moy's paper might be enough for beginners when introduced to Unix for the first time. The method itself perhaps is more attractive to some users, but teaching it this way might only work with simpler topics. Despite seemingly user-friendly approach there were still those who did not complete the set of exercises as mentioned in the paper. Therefore, author's solution still does not show that learning basics is easy for everybody. However, the idea of restricting learners from accessing the next task before they have completed the previous one, is reasonable and is also one of the approaches that will be used in current thesis.

2.2.3 Linuxgym

Linuxgym [18] is a software that was developed by Andrew Solomon et al. in 2002 [4] to automate assessment of Unix command line and scripting assignments. It is a virtual machine that imitates Linux operating system. Students log into Linuxgym Live machine to access exercises and their solutions are evaluated without leaving the system. Linuxgym Online was developed along with Linuxgym Live — a website primarily for teachers to monitor students' progress.

Solomon [4] observed that teaching larger groups of students makes it more difficult to provide each student with sufficient feedback as well as expect them to practice Unix skills for a certain number of hours. Moreover, approaches used for assessing student's skills were not effective enough. This was a problem at both the undergraduate and postgraduate level. He defined three main points to consider when teaching Unix to students:

a) to grade students using an approach that accurately determines their individual Unix command formulation skills;

b) to grade students in a manner that closely replicates the way that they will use their Unix skills in the real-world; and

c) to encourage students to practice and develop their Unix skills online [4].

Based on that the author's solution was to introduce a completely new approach to teaching and grading Unix skills — Linuxgym.

Prior to introducing Linuxgym the author's institution offered an undergraduate subject called Command-Line Interface. This course concentrated on teaching basic Unix commands such as `ls`, `mkdir`, and `cp`. During the exam students had to run a script that saved both input and output of his/her commands into a text file. In addition, students had to write an explanation of their solution. Both files were then read by the teaching assistants. In the end, this marking approach, however, was too time consuming for the teaching staff.

Solomon noted that a multiple choice and short answer tests are frequently used to grade such skills as well. These type of tests are used, for example, in the Linux Professional Institute exams. The tests assess students' ability to read and understand scripts before editing them. The flaw of such grading is that it is not formative. As an alternative, author mentioned the converse assessment where students have to choose a command to solve a task. Since there are many ways to perform a task then this approach is better than multiple choice questions where a student might not be familiar with the right possibility.

In the postgraduate programme, author's university offered a course on scripting with Bash and Perl. The assessment was similar to the Workstation Software course mentioned in subsection **??** — students were assessed by two assignments

were each required students two write a script. The first task was assigned in the middle of the semester because students needed certain skills to write the script. Unfortunately the teaching staff marked the assignment by the time students had to turn in the second task. This way students did not get feedback on time and could not use it to improve their skills and solve the second assignment better.

After the introduction of Linuxgym both aforementioned courses chose to use it instead of the old marking approaches. Linuxgym allows users to `ssh` into the course machine and access instructions file in their home directory. The file contains ten exercises that students have to complete within an hour. For instance, figure 4 shows one task that the file could contain:

> **Question 1:** There is a file in your home directory called 7sidd10.txt. Append the contents of /usr/local/unixexam-data/gutenberg/7myrt10.txt to the end of 7sidd10.txt.

Figure 4: Example of a Linuxgym exercise [4]

The directory `/usr/local/unixexam-data/gutenberg/` contains a set of various files. The names of the files mentioned in figure 4 are selected randomly from that directory. Some scripting tasks do not contain any randomly generated data like in the previous example. Yet, each student's script is tested against different data-files. The outputs of the student's script are compared to the outputs that the teacher expects.

After completing each question students can mark their solution using the `mark_question` script. There is no limit on how many times students can mark their solutions and the number of attempts for each question is not recorded. Listing 3 shows the marking process for the task showed in figure 4:

```
[solomon]$ mark_question 1
QUESTION 1 ..... WRONG:( - Try again.
[solomon]$ cat /usr/local/unixexam-data/gutenberg/7myrt10.txt
↪  >> 7sidd10.txt
[solomon]$ mark_question 1
QUESTION 1 ..... RIGHT - Well done!
```

Listing 3: Example of a Linuxgym automated marking [4]

After the time is up, students cannot access the system anymore. For each user a file is created in the teacher's directory and it contains marks for all ten question. A file containing student's ID and total score is saved along with the previous file.

To prove the effectiveness of Linuxgym students were asked to answer two questionnaires as well as participate in end of semester and online forum discussions. The survey targeted the students of the previously mentioned postgraduate course. During the semester students had to solve seven Linuxgym tests, answer two multiple choice tests and complete one scripting assignment. The first questionnaire focused on the overall quality of the course and the second one on students' opinion of Linuxgym as a tool.

The answers to the questionnaires were more positive after the introduction of Linuxgym, mainly there was a drop in the number of negative comments concerning the course. The author noted that students who previously failed the course performed better after Linuxgym was introduced. Solomon believed that it was mainly due to the fact that students could get instant feedback and practice their skills in a real Linux environment and not on paper. There were still some students who preferred the old marking system over Linuxgym but the majority stated that Linuxgym was more motivational and effective.

Based on the author's paper the introduction of Linuxgym indeed improved the quality of the courses that focused on Unix command line and scripting. The main improvement was not using the paper based tests since it only encouraged students to memorize certain commands and in the end they had little understanding of Linux systems. Linuxgym, on the other hand, allowed students to practice whenever they had time which possibly encouraged them to solve tasks more often and made learning process more effective. Having multiple Linuxgym tests throughout the semester made students learn new topics progressively.

Introducing automatic grading system in Linuxgym saved a lot of time for the teaching staff as well as left students satisfied with quick feedback. Students could get an idea of what might be wrong with their solution and even sort out

the problem before going to their teacher. In this thesis automated validation of exercises is proposed as well.

3 Solution Design

Last chapter described Bash shell scripting and how it is taught at various universities. This chapter concentrates on what approaches will be used by the author when creating sets of exercises on Bash scripting. Additionally, this chapter includes how the students' shell scripts will be evaluated.

3.1 Exercises

In section 3.1.1 the general overview of the exercise sets is given. In section 3.1.2 and section 3.1.3 the structure of the exercises and the validation process are described.

3.1.1 General overview

For Bash scripting, students could be provided with tasks as it is done in many other programming courses. In the end of each topic there would be a couple of homework assignments, e.g. a number of tasks that are focused on the use of the `for` loop. These assignments are usually not connected to each other. The goal of this thesis, however, is to generate sets of exercises for different Bash scripting topics where in each exercise set the tasks are chained — one task is connected to the previous and the next one.

Thus, completing every task is compulsory. Some output of a previous exercise or a working script from earlier are needed in order to complete the next tasks successfully. For instance, students might have an exercise where the script's output would be a text file containing some data. In the next task, they need to sort the data and use `grep` to get certain lines from the file. Unless students have solved the task where the data was generated, they do not have the right file to complete the next task.

It means that a student cannot move on to the next task if s/he has not finished a previous one. This type of approach ensures that students do not skip certain topics that they find complicated. It also makes sure that students actually understand the topic and practice their skills by writing scripts until they succeed.

Students' shell scripts are evaluated by using a standalone script that checks the output. If the output is right then students get access to the following exercise otherwise not. Some tasks may require that students use some command in their script or on the contrary avoid using specific commands. In that case, the validation script also checks the content of students' code.

3.1.2 Exercise structure

Mostly the top-down method will be used when creating the exercises. The top-down method implies that first off the final and the most advanced exercise is created and then it is broken down into simpler tasks. This helps to reach the most basic topics that need to be covered by a set of exercises. Since tasks in each set are connected then moving step-by-step allows students to prepare for more complicated assignments.

Some exercises are meant to be implementations of common Unix commands or command line utilities such as `grep`. This way students not only practice writing scripts on certain topics but also get better understanding of how those commands work. Moreover, in the end students will have their own implementations which they could modify further for their liking and use in the future.

For example, a task to write a script that every five seconds lists the contents of a directory. This could be useful, for instance, when copying a large file to another location and there is a need to monitor the progress. One of the shortest solutions is to use the `watch` command as shown in listing 4.

```
viktoria@Byron:~$ watch -n 5 ls -lh
```

Listing 4: The `watch` command

`watch` [19] executes any specified command every n seconds. First, `watch` displays a line which contains the interval, executed command, and date as shown in listing 5. Then follows the first screenfull of the output. If the interval option `-n` is not set like in listing 4 then `watch` executes the command every two seconds. `watch` has also the option `-d` or `--differences` among others to highlight any differences made between sequential command executions. The input command is executed until `watch` is terminated.

```
Every 5,0s: ls -lh                      Tue May 10 16:35:54 2016

total 24K
-rwxrw-r-- 1 viktoria viktoria  314 apr   29 00:46 jama.sh
-rwxrwxr-x 1 viktoria viktoria   31 apr   19 00:42 myscript.sh
-rwxrw-r-- 1 viktoria viktoria  783 mai    3 14:37 unix_for.sh
-rwxrw-r-- 1 viktoria viktoria  267 mai    5 19:28 unix_ifelse.sh
-rwxrw-r-- 1 viktoria viktoria 1005 mai    3 19:55 unix_while.sh
-rwxrw-r-- 1 viktoria viktoria   65 mai    7 22:11 watchtest.sh
```

Listing 5: Sample `watch` command output

The `watch` command is easy to use and would do the job but newcomers might not be familiar with this command yet. However, the same functionality can be achieved by writing a simple `while` loop and using a couple of common Unix commands like shown in listing 6. For students to write a similar script they have to be familiar with the `while` loop and know how to use commands like `ls`, `date`, `sleep` and `clear`.

```
1 #!/bin/bash
2 while true
3 do
4   clear; date; ls -l; sleep 5
5 done
```

Listing 6: The `watch` command implementation

`sleep` [20] delays the execution of commands for a specific amount of time which could be seconds (default), minutes, hours or days. The command is executed as

```
sleep NUMBER[SUFFIX]
```

where suffix could be *s*, *m*, *h* or *d*. Therefore, in listing 6 the program is delayed for 5 seconds after `ls -l` is executed. The `date` command displays by default current date and time and `clear` clears the terminal screen. In listing 6 `date`

imitates the title line like in the output of the `watch` command which helps to ensure that updates are actually made every 5 seconds. `clear` command removes the old output from the terminal screen before `ls -l` is executed again.

The exercises preceding the example task should, therefore, concentrate on the usage of

a) `while` loop;

b) `date`, `sleep` and `clear` commands.

Now, if students were given a task to implement the `watch` command using the `while` loop and `date`, `sleep`, `clear` commands, they would know how to do it.

3.1.3 Exercise validation

In the subsection 3.1.1 a brief overview of the assessment process was given. The complete design of the validation system consists of three main files.

The first part is the validator — a Python script which takes student's script as an input and reviews it. First file that this script reads is the `exec.txt` which contains all of the information regarding the exercises, i.e. what exercise precedes the one that is being checked, how many points is possible to gain, the name of the script to be checked and arguments. This information is later needed for testing the student's solution.

The validation script then refers to the final important file `synt.txt` which contains all the restrictions for the exercises. This is the second part of the whole validation process. This file sets the rules for the testing. For example, let the exercise A1 be the same as the sample task provided in subsection **??** — implementing the `watch` command by using the `while` loop and `date`, `sleep` and `clear` commands. The restrictions file would be checked for the commands that the student's implementation has to include. While the validator does check for certain commands which can and cannot be used, it does not deduct points for doing extra work as in using more advanced commands and programming solutions, e.g. regular expressions.

The next step would be to check the output of the solution. If the output of the student's script matches the one included in the testing file then the student gets another point added to the final grade. If the output is wrong no points are added to the final score.

Finally, the student gets feedback from the validator in the terminal. The feedback contains information about the points the student earned and lost as well as which requirements were met or not. Additionally, the student sees whether the final score was enough to pass this exercise and get access to the next one. The student has to earn the maximum score to move on — losing points means that

the student has not mastered necessary commands or constructs that the current exercise tried to teach.

4 Solution Implementation

Last chapter showed how the exercise sets are constructed and the validation of students' scripts is made. This chapter focuses on describing the exercise sets that were made for this thesis. In Chapter 4 validation process and the validation script is discussed in more depth as well.

4.1 Exercise sets

Author of the thesis created three sets of exercises for the purpose of testing the proposed solution to teaching advanced Bash scripting. Task definitions and sample solution can be found in appendix A. Each set of exercises contains a few subsequent exercises which are designed to be from easier to complicate on the level of difficulty. Some of the existing tasks could be made simpler or on the contrary more difficult in the future. The exercises assume that users are familiar with some basic everyday Unix commands.

First exercise set was created to practice writing `for` loops. This set of exercises takes a practical real world problem of transforming a set of 300 image files into a short video. Some of the files might be corrupted or empty files with similar names and extensions, so students are first supposed to sort find the right files. The output of this exercise set should be a video. The idea is to show how useful the `for` loop is and that it can be used for more than just printing lines from a file or similar. The student does not have to do a lot of unnecessary manual work but rather write a small script that automates the process.

The second set of exercises is focused on the usage of the `while` loop. By solving all of the exercises students get their own implementation of the `grep` tool. The implementation is a simplified version of the real command, i.e. without regular expressions and should work only with text files. Students again start from simpler exercises and move onto more complicated ones. The tasks include, for example, reading input from files, using command line arguments, counting lines and searching for a substring in text files. This set of exercises is intended to show students that is possible to write your own customized implementations of some common commands or tools.

Finally, a set of exercises was created that concentrates on the conditional statements. This set helps to create a script that checks if entered commands are executed successfully by referring to the exit code and making decisions based on that. Right now this is the longest of the three exercise sets and consists of six tasks. Besides, by the end of the set, students should be able to check whether background tasks are executed successfully. Here the importance of creating and reading log files is showed as well.

Student receives an instructions file for each exercise. The exercise has to

be completed according to the instructions for the validator to check it. The instructions state how the solution file should be named, what arguments should it take and how many points the exercise is worth. There is also general information about how the solution is marked and how to access following exercises.

4.2 Script validation

Along with the exercise sets the validation script was developed. Two versions of the script were written — one in Python 3.4 and the other one in Bash. Both codes can be found in appendix B. Both scripts were written while using the `while` loop exercise set in appendix A.2 for testing purposes but it was written by keeping in mind that it should be universal enough to make it work for other kinds exercise sets. The following discussion is written about the Python version of the validator.

The validation script itself is run from the command line as follows

```
$ python3 tester.py A1
```

where the argument given for the tester is the name of the exercise to be validated. The letter in the name corresponds to the set and the number to the exercise, i.e. A1 means that student wants to check the first exercise from set A. The script then collects all the necessary data that it needs — the execution parameters (`exec.txt`), necessary syntax (`synt.txt`) that should be present in student's script and expected output in a separate text file. The script uses dictionaries to keep the information of execution parameters and syntax.

4.2.1 Data collection

The text file `exec.txt` contains the data in the following form:

```
A1:A0;3;solution-A1.sh;99;test1_in.txt
A2:A1;3;solution-A2.sh;99;test1_in.txt
A3:A2;3;solution-A3.sh;99;test1_in.txt;test2_in.txt
...
```

Listing 7: The `exec.txt` file contents

The file contains essential parameters for every exercise. The name of the exercise and its parameters are separated by a colon. The parameters are in turn separated by semicolons. For example, exercise A1 in listing 7 has the following parameters:

1) A0 corresponds to the previous exercise on which A1 depends;
2) 3 shows what is the maximum amount of points for this exercise;
3) `solution-A1.sh` is the name of the student's script that the validator needs to check;
4) 99 is the first argument that is passed to the student's script;
5) `test1_in.txt` is the input data file.

The exercises usually get one or more arguments which could be of any format. Since the exercise A1 in listing 7 is concentrated on implementing `grep`, then the arguments are a string and a text file. In a different task it could have been, for example, a directory name, path or command as well.

The file `synt.txt` is of similar construction as the previous one:

```
A1:while=yes;grep=no
A2:while=yes;grep=no
A3:while=yes;grep=no;$
...
```

Listing 8: The `synt.txt` file contents

Again exercise names are separated from parameters by a colon. Since some exercise might have restrictions then they are written down in this file. For example, in listing 8 `while=yes` means that students should use the `while` loop in their solution and `grep=no` shows which command cannot be used.

4.2.2 Validation and feedback

After the tester has read the necessary data, the student's solution is executed to get the output. The Python's subprocess module [21] is used for the execution of the student's Bash script. The output is retrieved by using the `subprocess.check_output` method which gets the following command as an input:

```
bash file arg1 arg2...
```

There `file` is the name of the solution file and `arg1`, `arg2` and so on are arguments.

If the execution is successful, the validator starts checking the output of the solution and whether it corresponds to the requirements file. Students get one point for generating right output and one point for each correctly implemented

requirement. The feedback is printed to the terminal after checking is done as shown in listing 9.

```
$ python3 tester.py A1
Solution: OK
Script must have WHILE: OK
Script must not have GREP: OK

Total:   3/3
Great job! You can now access next task.

$ python3 tester.py A2
Solution: NOK
Script must have WHILE: OK
Script must not have GREP: OK

Total:   2/3
You need 1 more point(s) to access next exercise. Try again!
```

Listing 9: Positive and negative feedback

After the feedback is printed, the tester appends the date and time of the attempt as well as points for the task into a separate text file.

4.2.3 Validator implementation in Bash

The validator also has a working version written in Bash. However, it was more time consuming and complicated to write the script in Bash than in Python which only proves that writing longer Bash scripts needs a lot of practice and skill. Moreover, rewriting the script to suit different types of exercises seems more difficult than in Python. Nevertheless, the fact that the validation script could be written already in two different scripting languages, shows that the idea behind the tester's structure is good enough to use and develop the tester further.

5 Conclusion

In the process of this thesis three sets of exercises each set focusing on one of the following topics were created: `for` loop, `while` loop, and conditional statements. Each set consists of at least four tasks. The aim was to create chained exercises with increasing difficulty. For assessment and marking the automatic testing script was implemented for one set of exercises in both Python and Bash languages. Retrofitting the validation script for new exercises can be done quickly in Python but is not impossible in Bash as well. While the exercise set lacks a real life testing group, it did achieve its purpose of being a starting point for future implementations of Bash scripting related material. It also shows promise as a possible candidate for a Bash e-course if needed. Therefore students who need to take subjects where the Linux command line interface is used often, could practice their Bash command line and scripting skills online while still getting feedback.

For future work the tester should be completed for all of the exercises which also includes updating parameters and syntax requirements files as well as generating more testing data. Exercise testing could use some randomness in testing data as well. The tester should include an exception control for all different errors made by students. More exercises could be added to cover more important topics in Bash scripting since it is a wide topic.

References

[1] "MTAT.03.011 Tööjaamade tarkvara. 3. praktikum – *Shell*: muutujad. Utiliidid, käsukonveierid, arhiveerimine," 2009. [Online]. Available: https://courses.cs.ut.ee/2009/tjt/Main/Lab3 [Accessed: 11.05.2016].

[2] "MTAT.03.011 Tööjaamade tarkvara. Esimene individuaalülesanne: *shell*iskript," 2009. [Online]. Available: https://courses.cs.ut.ee/2009/tjt/Main/ProblemsSh [Accessed: 11.05.2016].

[3] "Unix Training Ensimag," retrieved from: https://gitlab.com/unix-training/unix-training [Accessed: 12.05.2016].

[4] A. Solomon, D. Santamaria, and R. Lister, "Automated Testing of Unix Command-line and Scripting Skills," in *Information Technology Based Higher Education and Training, 2006. ITHET'06. 7th International Conference on.* IEEE, 2006, pp. 120–125.

[5] M. Garrels, "Bash Guide for Beginners." [Online]. Available: http://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf [Accessed: 20.04.2016].

[6] "The bash FAQ." [Online]. Available: ftp://ftp.cwru.edu/pub/bash/FAQ [Accessed: 19.04.2016].

[7] M. T. Jones, "Evolution of shells in Linux," *IBM developerWorks*, 2011.

[8] C. Ramey and B. Fox, *Bash Reference Manual*, 2014.

[9] C. Ramey, "BASH — The Bourne-Again Shell," Available at http://tiswww.case.edu/php/chet/bash/bash-intro.html, [Accessed: 08.05.2016].

[10] R. Toal, "Introduction to Bash." [Online]. Available: http://cs.lmu.edu/~ray/notes/bash/ [Accessed: 19.04.2016].

[11] M. Cooper, "Advanced bash-scripting guide," 2014.

[12] "MTAT.03.011 Tööjaamade tarkvara," 2009. [Online]. Available: https://courses.cs.ut.ee/2009/tjt/Main/HomePage [Accessed: 11.05.2016].

[13] Marti Taremaa, "MTAT.03.011 Tööjaamade tarkvara. 1. loeng - Sisututvustus, tööjaamad, UNIX ajalugu," 2009. [Online]. Available: https://courses.cs.ut.ee/2009/tjt/uploads/Main/loeng1.pdf [Accessed: 20.04.2016].

[14] M. Moy, "Efficient and Playful Tools to Teach Unix to New Students," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 93–97.

[15] O. Alphand, N. Berthier, B. Cassagne, C. Cassagne, G. Funchal, M. Moy, S. Nieuviarts, and F. Perronnin, *Initiation à Unix — L'environnement de travail à l'Ensimag*, 2015. [Online]. Available: http://www-verimag.imag.fr/~moy/IMG/pdf/poly-intro-unix.pdf

[16] "EnsiWiki — Stage Unix de rentrée." [Online]. Available: http://ensiwiki.ensimag.fr/index.php/Stage_Unix_de_rentr%C3%A9e [Accessed: 20.04.2016].

[17] "EnsiWiki — Script Shell." [Online]. Available: http://ensiwiki.ensimag.fr/index.php/Script_Shell [Accessed: 20.04.2016].

[18] A. Solomon, "Linuxgym: Software to Automate Formative Assessment of Unix Command-line and Scripting Skills," *SIGCSE Bull.*, vol. 39, no. 3, pp. 353–353, Jun. 2007.

[19] "Ubuntu Manpage: watch - execute a program periodically, showing output fullscreen." [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/watch.1.html [Accessed: 10.05.2016].

[20] "Ubuntu Manpage: sleep - delay for a specified amount of time," Available at http://manpages.ubuntu.com/manpages/trusty/man1/sleep.1.html, [Accessed: 10.05.2016].

[21] "17.5. subprocess — Subprocess management — Python 3.5.1 documentation." [Online]. Available: https://docs.python.org/3/library/subprocess.html [Accessed: 12.05.2016].

[22] "ImageMagick: Convert, Edit, Or Compose Bitmap Images." [Online]. Available: http://www.imagemagick.org/script/index.php [Accessed: 12.05.2016].

A Exercise sets

A.1 For loop

A.1.1

1 POINT. You will find a directory named `images` in your training directory. It is filled with about 300 image files of .tga (TARGA Image File) format. Write a script that lists all the .tga files in the directory. Your script should be named `solution-A1.sh` and take the image format as an argument.

```
#!/bin/bash
ls -l ./* | grep -e ".tga$"
```

Listing 10: Sample solution of A.1.1

A.1.2

2 POINTS. The directory also includes some dummy files, i.e. empty or corrupted files which have similar names as real images in the directory. Modify your previous script that it would check the type of the listed files using the `file` command. Your script should be named `solution-A2.sh` and take the image format as an argument.

```
#!/bin/bash
for line in $(ls ./* | grep -e ".tga$"); do file "$line"; done
```

Listing 11: Sample solution of A.1.2

A.1.3

3 POINTS. Modify your previous script that it would put the output of the last exercise into a text file called `output.txt`. Your script should be named `solution-A3.sh` and take the image format as an argument.

```
#!/bin/bash
for line in $(ls ./* | grep -e ".tga$"); do file "$line"; done
 ↪  > output.txt
```

Listing 12: Sample solution of A.1.3

A.1.4

4 POINTS. Modify your previous script. From the output file that you now have, get the names of the real `.tga` files and save them in a variable called `names`. Your script should be named `solution-A4.sh` and take the image format as an argument.

```
#!/bin/bash
for line in $(ls ./* | grep -e ".tga$"); do file "$line"; done
 ↪  > output.txt

names=$(for line in output.txt; do grep -e "Targa" ${line} |
 ↪  cut -d':' -f1; done)
```

Listing 13: Sample solution of A.1.4

A.1.5

5 POINTS. Modify your previous script. Iterate over the names and convert them from .tga to .jpg. Use the `convert` command from Imagemagick [22] package for this. Your script should be named `solution-A5.sh` and take the image format as an argument.

```
#!/bin/bash
for line in $(ls ./* | grep -e ".tga$"); do file "$line"; done
 ↪  > output.txt

names=$(for line in output.txt; do grep -e "Targa" ${line} |
 ↪  cut -d':' -f1; done)

for image in $names; do convert "$image" "${image%.*}.jpg";
 ↪  done
```

Listing 14: Sample solution of A.1.5

A.1.6

6 POINTS. Modify your previous script. Use your .jpg pictures to put them together into a video. Search for the `avconv` command to get more information. Your script should be named `solution-A6.sh` and take the image format as an argument.

A.2 While loop

A.2.1

2 POINTS. You have a text file called `task_001.txt` in your training directory. Write a script that takes the file as an argument, reads the file and prints out all the lines it contains. Use the `while` loop for reading lines. Your script should be named `solution-B1.sh`.

```
#!/bin/bash
input_file=$1
while read -r line; do
  echo "$line"
done < $input_file
```

A.2.2

3 POINTS. Modify your previous script that it additionally takes one random word as an argument. Loop over the file and print out only the lines that contain the given word. You cannot use `grep` for this. Your script should be named `solution-B2.sh`, the given word has to be the first argument and the input file the second argument.

```
#!/bin/bash
word=$1
input_file=$2

while read -r line; do
  if [[ "$line" == *"$word"* ]]
  then
    echo "$line"
  fi
done < $input_file
```

A.2.3

3 POINTS. Modify your previous script that it first prints the line number where the given word was found and then the line itself. For example:

```
4  This here is a test sentence
```

You cannot use `grep` in your code. Your script should be named `solution-B3.sh`, the given word has to be the first argument and the input file the second argument.

```bash
#!/bin/bash
word=$1
input_file=$2

counter=0
while read -r line; do
  counter=$(( $counter + 1 ))
  if [[ "$line" == *"$word"* ]]
  then
    echo "  $counter  $line"
  fi
done < $input_file
```

Listing 15: Sample solution of A.2.3

A.2.4

3 POINTS. Modify your script that it could take several text files as arguments. You can use `task_001.txt` and `task_002.txt` for testing. The script should print in which file it is currently searching as follows:

```
------------------------------
Searching in file task_001.txt:
------------------------------
12 This here is a test sentence
```

You cannot use `grep` in your code. Your script should be named `solution-B4.sh`, the given word has to be the first argument like before.

```bash
#!/bin/bash
word=$1 # first argument is the word that we search
shift   # shift to the next argument

files=$* # arguments without the first one, i.e. input files
for f in $files
do
  counter=0
  echo "------------------------------"
  echo "Searching in file ${f}:"
  echo "------------------------------"

  while read -r line; do
    counter=$(( $counter + 1 ))
    if [[ "$line" == *"$word"* ]]
    then
      echo "  $counter  $line"
    fi
  done < $f
done
```

Listing 16: Sample solution of A.2.4

A.3 Conditionals

A.3.1

1 POINT. Write a simple script that takes a command as an argument and runs it. Your script should be named `solution-C1.sh`.

```
#!/bin/bash
cmd=$1
eval $cmd
```

Listing 17: Sample solution of A.3.1

A.3.2

3 POINTS. Modify your previous script to check whether the command it takes as an argument runs successfully or not. For that check the exit code: `$?`. Print out `OK` or `NOK` accordingly. Your script should be named `solution-C2.sh`.

```
#!/bin/bash
cmd=$1
eval $cmd

if [ $? -le 0 ]
then
  echo OK
else
  echo NOK
fi
```

Listing 18: Sample solution of A.3.2

A.3.3

3 POINTS. Modify your previous script that it could take more than one argument and provide feedback to each command given like in the previous task. Your script should be named `solution-C3.sh`.

```
#!/bin/bash
cmd=$*
for c in $cmd
do
  eval $c
  if [ $? -le 0 ]
  then
    echo ${c}: OK
  else
    echo ${c}: NOK
  fi
done
```

Listing 19: Sample solution of A.3.3

A.3.4

4 POINTS. Modify your previous script that it would send command outputs to a log file. Name your log file `script.log`. You should see only `OK` or `NOK` printed out in the terminal. Check if your log file is created and contains the right output! Your script should be named `solution-C4.sh`.

```
#!/bin/bash
cmd=$*
LOG=/tmp/script.log

for c in $cmd
do
  eval $c 2>&1 >> $LOG
  if [ $? -le 0 ]
  then
    echo ${c}: OK
  else
    echo ${c}: NOK
  fi
done
```

Listing 20: Sample solution of A.3.4

A.3.5

5 POINTS. Modify your previous script that it would run each command in the background and write exit code into a temporary file named after the given command, for example `ls.tmp`. Wait for each background process to finish and then check exit codes from the files. Print `OK` or `NOK` as previously.Your script should be named `solution-C5.sh`.

```
#!/bin/bash
cmd=$*
LOG=/tmp/script.log

for c in $cmd; do
(eval $c 2>&1 >> $LOG; echo $? > /tmp/${c}.tmp) &
done

wait

for f in $(find /tmp/ -name '*.tmp'); do
  if [ $(cat $f) -le 0 ]
  then
    echo ${f}: OK
  else
    echo ${f}: NOK
  fi
done
```

Listing 21: Sample solution of A.3.5

A.3.6

5 POINTS. Modify your previous script that it would add the exit codes from all of the temporary files. If the exit code is 0, print `OK`, otherwise `NOK`. Your script should be named `solution-C6.sh`.

```
#!/bin/bash
cmd=$*
LOG=/tmp/script.log

for c in $cmd; do
(eval $c 2>&1 >> $LOG; echo $? > /tmp/${c}.tmp) &
done

wait

sum=0
for f in $(find /tmp/ -name '*.tmp'); do
  num=$(cat $f)
  sum=$(( $sum + $num ))
done

if [ $sum -le 0 ]
then
  echo OK
else
  echo NOK
fi
```

Listing 22: Sample solution of A.3.6

B Tester codes

B.1 Bash implementation

```
#!/bin/bash

STDERR_LOG=tester_stderr.log
FILE=$(grep 'file' $1 | cut -d' ' -f2-)
ARGS=$(grep 'arg' $1 | cut -d' '  -f2-)
SYNT=$(grep 'synt' $1 | cut -d' ' -f2)
PTS=$(grep 'pts' $1 | cut -d' '  -f2)
ID=$(grep 'id' $1 | cut -d' '  -f2)
EXP=$(cat $2)
SYNT_POS=$(grep 000 $SYNT | cut -d\  -f2-)
SYNT_NEG=$(grep 111 $SYNT | cut -d\  -f2-)
RES_OUT=$(echo $(basename $0) | cut -d'.' -f1)'-'${ID}.txt

echo $(date) >> $STDERR_LOG
RES=$(bash $FILE $ARGS 2>>$STDERR_LOG)

FIN_PTS=$PTS

if [ "$RES" == "$EXP" ]; then VALID_CODE=OK; else
 ↪  VALID_CODE=NOK; FIN_PTS=$((FIN_PTS-1)); fi;

P=$(grep $SYNT_POS $FILE | wc -l);
if [ $P -ge 1 ]; then VALID_SYN_POS=OK; else VALID_SYN_POS=NOK;
 ↪  FIN_PTS=$((FIN_PTS-1)); fi;
```

Listing 23: The validation script in Bash

```
23 N=$(grep $SYNT_NEG $FILE | wc -l);
24 if [ $N -ge 1 ]; then VALID_SYN_NEG=NOK;
    ↪  FIN_PTS=$((FIN_PTS-N)); else VALID_SYN_NEG=OK; fi;
25
26 echo 'Solution : ' $VALID_CODE
27 echo 'Script must have : ' $SYNT_POS ' : '  $VALID_SYN_POS
28 echo 'Script must not  have : ' $SYNT_NEG ' : ' $VALID_SYN_NEG
29
30 echo 'Total ' $FIN_PTS '/' $PTS
31
32 echo $ID':'$FIN_PTS >> $RES_OUT
```

Listing 24: The validation script in Bash

B.2 Python implementation

```
#!/usr/bin/python3.4

import subprocess
import sys
from _datetime import datetime


def populate_dictionary(in_file):
    dic = {}
    with open(in_file, 'r') as f:
        for line in f:
            split_line = line.strip().split(':')
            dic[split_line[0]] = ';'.join(split_line[1:])
    return dic

# argument for the tester, e.g. A1
exercise_id = sys.argv[-1]
# read parameters for execution
definitions = populate_dictionary('exec.txt')
# what syntax should/shouldn't be in the script
syntaxes = populate_dictionary('synt.txt')

with open(exercise_id + '_exp.txt', 'r') as f:
    expected_output = f.read().strip()
```

Listing 25: The validation script in Python

```
split_definition = definitions[exercise_id].split(';')
dependency = split_definition[0]  # previous exercise
points = int(split_definition[1])  # max points
file = split_definition[2]  # student's script to check
arguments = split_definition[3:]  # arguments for the above

split_syntax = syntaxes[exercise_id].split(';')
positive_syntax = [cmd.split('=')[0] for cmd in split_syntax if 'yes' in cmd]
negative_syntax = [cmd.split('=')[0] for cmd in split_syntax if 'no' in cmd]

with open(file, 'r') as f:
    student_script = f.read().strip()

# get student's result
result = subprocess.check_output(['bash', file] + arguments, universal_newlines=True).strip()
final_points = 0

# compare student's result with expected output
if result in expected_output:
    print("Solution: OK")
    final_points += 1
else:
    print("Solution: NOK")
```

Listing 26: The validation script in Python

```
48  # check if student's script has required/forbidden syntax
49  for cmd in positive_syntax:
50      if cmd in student_script:
51          print("Script must have " + cmd.upper() + ": OK")
52          final_points += 1
53      else:
54          print("Script must have " + cmd.upper() + ": NOK")
55
56  for cmd in negative_syntax:
57      if cmd not in student_script:
58          print("Script must not have " + cmd.upper() + ": OK")
59          final_points += 1
60      else:
61          print("Script must not have " + cmd.upper() + ": NOK")
62
63  # print feedback
64  print("\nTotal:", '\t', str(final_points) + '/' + str(points))
65
66  if final_points < points:
67      print("You need " + str(points - final_points) +
68            " more point(s) to access next exercise. Try again!")
69  else:
70      print("Great job! You can now open next task.")
71
72  # write exercise's result to a file
73  with open(exercise_id + ".txt", "a") as f:
74      f.write(datetime.now().strftime("%c") + '\n')
75      f.write(exercise_id + ':' + str(final_points) + '\n')
```

Listing 27: The validation script in Python

Non-exclusive licence to reproduce thesis and make thesis public

I, Viktoria Plemakova (date of birth: 1st of January 1995),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Teaching advanced Bash scripting using semi-automated assessment environment

supervised by Artjom Lind

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 12.05.2016