Appendix A

The UNIX terminal

In this appendix a few practical details about using UNIX is explained. First of all, UNIX is an operating system developed at Bell Labs in the early 1970's. UNIX is the most successful operating system for computer professionals. Linux and MacOS X and several other versions are all implementations of the UNIX system. The main non-UNIX operating system, Microsoft Windows, now supports most Linux programs through its Ubuntu subsystem.

A.1 Starting a terminal

To start you need a window which is a **terminal**. We assume you use Linux or MacOS X when explaining how to start a terminal. Apart from this beginning step, the rest will be applicable also for Windows using the Ubuntu subsystem. Locate the part where you can search for commands, and type `terminal`. Then hit the ENTER or RETURN key on your keyboard. If this does not work, use a search engine to find out how a terminal is opened on your computer. When you have started the terminal, type the following command and hit ENTER:

```
pwd
```

You should then get output such as `/home/`*`yourname`*. The terminal is a window and inside the window a program called `bash` reads what you type and responds to your commands. Bash is a **command line interpreter**, a type of program also called a **shell**. Bash stands for **Bourne Again Shell**, and is a more sophisticated version of the first UNIX shell, written by the British mathematician Stephen Bourne. Both Linux and MacOS X use Bash. When you typed `pwd` and hit ENTER, Bash read what you typed and then executed the command to print the **working directory**. We will next see what a **directory** is.

A.2 Files and directories

All documents in computers and also in mobile phones are located in one of several directories. In UNIX, a document is also called a **file**. Almost everything is a file in UNIX, including directories, printers and USB memory sticks. This is for organizing all your documents to make it easier to work with computers. Every user on a UNIX computer has a **home directory** which is the root of a tree of directories which you own. The tilde character, ~, is an abbreviation of your home directory. Every person with an account on your computer has her or his own home directory. An analogy to home directories are rooms in a building (and the computer being the building). Inside a room there may be shelves to put documents on. One directory can contain other directories. Your home directory probably has a `Documents`, a `Downloads` and a `Desktop` directory. The Desktop directory is special because you can usually see the documents in this directory simply by looking at your computer display.

From now on, we will not write that you should hit ENTER after a command. Next type the command:

```
cd Desktop
```

Bash now uses your Desktop directory as its working directory. Verify this by giving the command `pwd` again. You should now make an experiment: give the following command:

```
ls
```

The output should be a list of all **files** which are also visible on your computer display. The command `ls` stands for *list* (as in the verb). Next give the command (called ”ls minus ell”):

```
ls -l
```

This option to `ls` tells `ls` to print more information about every file, such as how many bytes it is (5th column) and when it was most recently modified.

The command `cd` which you used before, can also be used without any argument, and the result then is to change directory to your home directory. Do that:

```
cd
```

Say you want to create a directory for the labs for this book. To create a directory `algorithms`, you type:

```
mkdir algorithms
```

Change directory to it and then create a new directory `labs` and then change directory to `labs`. We usually simply say ”cd to labs”. If all is well, you should be in the following directory now: `/home/`*yourname*`/algorithms/labs`.

Now type:

```
ls -a
```

You will see two file names: ”dot” and ”dotdot”, i.e. the file with the short name `.` and another with name `..` and these names are present in every directory. Files starting with a dot are only listed with the `-a` option. The file `.` is the name of the current directory and `..` of the parent directory. Type:

```
cd ..
```

Verify that you are in the directory `/home/`*yourname*`/algorithms`. Using the name `.` will be shown later.

To repeat the most recent command, use `!!` and to repeat the most recent command which begins with a string such as `abc` use `!abc` or the keyboard’s ”up-arrow” key, which also can be used to find previous commands.

A.3 Copying files

To copy a file, use the command `cp`. If you have a file `graph.scala`, and want to copy it before modifying it (as a backup), use:

```
cp graph.scala another-name.scala
```

Or any other name. So you copy from a file to another file. Now cd to your home directory. To make a copy of your directory `algorithms`, type:

```
cp -r algorithms algorithms-001
```

The `-r` stands for recursively, i.e. all directories in `algorithms` are copied as well.

Making backups of your files is very essential. A useful way is to store them on different computers. To copy files between computers is also useful when preparing work at home and then using it at a university computer, for instance. How to do that will be explained next.

A.4 Basic SSH commands

Secure Shell or **SSH** is a set of commands which make it possible to copy files between computers and to work remotely from a terminal on your computer in a terminal on a different computer. Everything you type, including passwords, is encrypted. First, to copy files from your computer to a university account, which for illustration we assume is `abc@login.student.lth.se`, you use the command:

```
scp -r algorithms abc@login.student.lth.se:
```

The first time you contact a remote computer you will be asked if you trust it is the correct computer. Normally it is safe to reply `yes`. You will then be requested to enter your password on the remote computer (which as said is safe to do, unless the computer you login to somehow has been hacked, of course). Notice the colon at the end. It tells `scp` that the string before is an account at a certain computer. Skipping the colon is equivalent to using the local version of copy, `cp`.

To copy a directory from the university, say, `algorithms/labs/lab1` to your computer, type:

```
scp -r abc@login.student.lth.se:algorithms/labs/lab1 .
```

Note the use of the dot: we tell `scp` to copy the remote `lab1` into the current directory (which has name `.`) and we also see that in addition to an account and computer, we can also specify a certain directory, which must be present in your home directory at the remote computer (i.e. the first part, `algorithms`, must be present).

To work from home on a remote computer, you can use:

```
ssh -Y abc@login.student.lth.se
```

Here no colon is used. The `-Y` option is useful if you want to open a window on the remote computer which should appear on your local display. Often doing so is slow, but with a fast Internet connection it can work to some extent. You may want to try to use an editor such as `atom`, although editors developed for terminals, such as `vi`, `emacs`, or `nano` probably are more suitable if your Internet connection is slower. You logout from the remote machine by typing `exit` or pressing CTRL-D.

To avoid having to enter passwords you can create a certain file and copy once to the remote computer as shown next. Enter the command on your local computer:

```
ssh-keygen -t rsa
```

The program will ask you to select a secret message which will be encrypted. It is recommended to enter a different message for the so called passphrase. When you have followed the instructions, you will see two new files in the directory `.ssh` in your home directory: `id_rsa` and `id_rsa.pub` and it is the latter which you should copy to the remote machine:

```
ssh-copy-id abc@login.student.lth.se
```

When you next login to the remote computer, it is possible that your local computer will ask if it should remember your secret message. You probably want to answer ”yes” and ”forever” or something similar. Now you should be able to both ssh and scp to the remote computer without having to enter a password.

SSH sessions are terminated after some time of no activity, which can be annoying, but by putting the following line in your SSH configuration file `~/.ssh/config`:

```
ServerAliveInterval 60
```

a message will be sent every minute to the SSH server at the remote computer and the session will be kept alive.

A.5 File name expansion

Suppose you want to cd to `Downloads` from your home directory. Then you can type `cd Dow` and TAB. If `Dow` is a unique file name prefix in this directory, Bash will expand it to `Downloads`.

Another form of file name expansion is the use of $*$ and `?`. The string `*.scala` refers to all files which end with `.scala`. The `?` matches one character, so `?.scala` refers to all files with one character followed by `.scala`, such as `a.scala`.

A.6 Removing files

To remove files, use the command `rm`:

```
rm doomed
```

To remove a directory, add the option `-r`. Many UNIX users learn "the hard way" that a command such as:

```
rm * .class
```

has a space after the $*$ which means that all files are matched and removed, after which you are told that no file named `.class` exists.

A.7 Scala and C programming

To compile a Scala file called `graph.scala`, type:

```
scalac graph.scala
```

And to run the `main` method the `object graph` type:

```
scala graph
```

As you might know, it is better to use `fsc` instead of `scalac` since it starts a server which reduces the compilation time. To instead compile a C program `graph.c`, type:

```
cc graph.c
```

And to run it, type:

```
./a.out
```

To use the large number of programming tools for C and C++, we refer the reader to [20].

A.8 Simple shell programming

Bash is a programming language with variables and control flow statements including `if` and `for` which make it possible to do shell programming. A shell program is also called a **script** and contains commands. One of the most useful shell programs for programming is to automate running and testing. Suppose you want to compile a Scala program and then run it if the compilation succeeded. This can be done as follows:

```
scalac graph.scala && scala graph
```

All programs return an integer to Bash[1] and by convention, zero means success and nonzero some kind of failure. When your program calls `exit(0)` or returns zero from its `main` method, then this is thus used by Bash. Normally Bash ignores the return value but in the command above, `scala graph` is only run if `scalac` returned zero.

Suppose you want to test that the output of your program is correct. This can be done by having a file called e.g. `correct` and saving your program's output to a file called e.g. `output`. The a program `diff` compares these files and returns, as expected, zero if they are identical, i.e. your program passed the test. First you need to create the file `correct`. Then type:

```
scalac graph.scala && scala graph > output && diff output correct
```

Then `diff` prints, as briefly as possible, the difference in the two files. If `diff` prints no output, the files are identical. The program `echo` simply prints the arguments as output. This can be used as follows:

```
scalac graph.scala &&
scala graph > output &&
diff output correct &&
echo PASS
```

By putting the && at the very end of the line, Bash continues with the command on the next line (that is, no blank is permitted to the right of &&).

If you put the previous command in a file, say `t`, you can then type:

```
sh t
```

This file `t` is an example of a small shell program. Suppose then you have several test cases (such as thousands) as pairs of input and correct files, named as: *testname*`.in` and *testname*`.correct`. You can do as follows to test all of them, and instead of typing this, you can download it from the book's home page at `concise-algorithms.net`.

[1] To be more accurate, programs return an integer to the parent, which normally is the program which started it, which normally is Bash, but any program can create new programs and check the return value.

```
for x in *.in
do
        echo testing $x
        y=$(basename $x .in)
        scala graph $x > output
        if diff output $y.correct
        then
                echo test failed for $x
                exit 1
        fi
done
```

This illustrates the use of control statements and shell variables. The `*.in` matches all filenames which end with `.in` and the for-loop will iterate through these, with the shell variable `x` assigned to one such input file in each iteration. The the loop prints out which input file is being tested. As we can see, we get the value of a shell variable by using a dollar sign. Without the dollar-sign, Bash cannot distinguish between a file named `x` and the shell variable `x`. The next statement looks a bit weird but assigns the test name without the `.in` part to shell variable `y`. In general, putting a command within `$()` produces a string which can be assigned to a variable, for instance. To produce the filename for the ”correct” file, we can use `$y.correct` which if the input file was `a.in` results in the filename `a.correct`. If `diff` returns a nonzero Bash will notice that in the if-statement and terminate the testing. In case you would like to continue with other tests, a hashtag is similar to a `//` comment in other languages, so by changing the script to contain `# exit 1`, the script will not exit at an error but continue with other test cases.

Finally, to see which commands a script executes, you can use:

```
sh -x t
```

This ends this short introduction to the UNIX terminal.