Unix-Like Data Processing 2017

v1-draft (2017-08-16)

Oleks Shturmov
oleks@oleks.info

Morten Brøns-Pedersen
f@ntast.dk

Troels Henriksen
athas@sigkill.dk

1. Introduction 2
- 1.1. Unix-Like Operating Systems . . . 3
- 1.2. Logging In on a Unix-like Machine . . . 4

2. Shell 101 4
- 2.1. The Prompt . . . 5
- 2.2. The Working Directory . . . 5
- 2.3. `touch`, `ls`, and `rm` . . . 6
- 2.4. Globbing . . . 9
- 2.5. `pwd`, `mkdir`, `mv`, and `tree` . . . 10
- 2.6. `cd` . . . 14
- 2.7. `echo`, `cat`, and `>` . . . 16
- 2.8. `wc` and `|` . . . 18
- 2.9. `nano` . . . 19
- 2.10. `history` . . . 20
- 2.11. `--help`, `man`, and `less` . . . 20
- 2.12. Hello, Shell Script . . . 21
- 2.13. `file` and `which` . . . 21

3. Working with More Data 22
- 3.1. `curl` and `less` . . . 23
- 3.2. `head` and `tail` . . . 23
- 3.3. `cut` and `paste` . . . 24
- 3.4. `sort` and `uniq` . . . 27

4. Search, Replace, and Compare 28
- 4.1. `grep` . . . 28
- 4.2. `sed` or `perl` . . . 32
- 4.3. Other Languages . . . 34
- 4.4. `cmp`, `comm`, and `diff` . . . 34

5. Shell Scripting 36
5.1. Basic Control Flow 37
5.2. `PATH` 39
5.3. Exercise Series: `tmpdir` 40
5.4. `mktemp` 41
6. Applications 42
6.1. ImageMagick 42
6.2. `gnuplot` 42
7. Makefiles 43
7.1. Hello, `make` 44
7.2. Hello, `Makefile` 45
7.3. Phony Targets 46
7.4. A `test` target 49
7.5. The Default Target 51
7.6. A More Complicated Program 52
7.7. Variables 53
7.8. Conclusion 55
7.9. Further Study 55
8. Conclusion 55

NB! This is a draft version of the notes.

Preface

The following notes were originally written by Oleks and Morten in preparation for a 1-day course for some Danish high-school students[1] in 2015. In 2016, they were expanded upon by Oleks and Troels in preparation for a week-long workshop for (among others) 2nd-year students at DIKU. In 2017, Oleks rewrote the notes in Madoko, and added a section on Makefiles, repeating the workshop.

1. Introduction

An *operating system* stands between the programs you are running (e.g. Chrome, Firefox, or Adobe Acrobat Reader), and the physical hardware packed in your computer. An operating system makes sure that the few hardware resources you have in store, are fairly shared among your programs. It also makes sure that programs don't unintentionally get in the way of one another, and in the end permits you to run multiple programs, simultaneously on one computer.

Operating systems first appeared in the 1960s in response to high demand for computational power. At the time, different users (students, researchers, and staff) wanted to use the big clunky machines filling their basements for many different tasks. Some tasks took longer than others. Some tasks demanded more memory, or disk space than others. Early operating systems made sure to share those resources fairly among the users of what was often, one computer.

[1] See also Akademiet for Talentfulde Unge. The original notes are available here.

As computers evolved and personal computers emerged, the focus shifted from supporting multiple users, to supporting multiple programs running simultaneously on one computer.

Introvert personal computers quickly proved unproductive and boring: The Internet emerged to connect these marvelous machines into a World-Wide Web of raw computational power, where your operating system now also mediates your communication with the dangerous world that's out there.

This enabled the rich desktop, laptop, and handheld devices that we have today.

1.1. Unix-Like Operating Systems

UNIX® is a trademark of The Open Group. They certify which operating systems conform to their Single UNIX Specification.

Colloquially however, "Unix" refers to a family of operating systems developed at Bell Labs in the 1970s, and their countless descendants. Modern descendants are better called "Unix-like" operating systems. You might be familiar with such Unix-like operating systems as Linux, FreeBSD, OpenBSD, OS X, and iOS. Most notoriously, Microsoft Windows is *not* a Unix-like operating system.

A 1982 Bell Labs video, recently made available under the AT&T Archives, starring such pioneers as Ken Thompson, Dennis Ritchie, and Brian Kernighan, gives further insight into what the original UNIX systems were like, and the philosophy and history behind them.

An important aspect of the history of UNIX is that it has always been guided by the needs of its users. This goal, quite incidentally, results in a philosophy:

> Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is an approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial tasks in isolation, but, combined with other programs, become general and useful tools.
>
> — *Brian Kernigan and Rob Pike. The UNIX Programming Environment (1984).*

The ultimate purpose of this document is to introduce you to this philosophy of using the computer. This philosophy is likely to be different from what you are accustomed to; and yet, it stands on the shoulders of many humble pioneers of modern operating systems.

The reason that this way of using the computer is not in wide-spread adoption is perhaps due to:

1. a general public disinterest in computer programming,

2. the general public fondness of graphical user interfaces , and
3. the resulting sectarianism of those who practice what we preach.

Last, but not leat, many aspects of a Unix-like operating system are ultimately about *freedom*: the freedom to chose how to set up your computer. That is "free" as in free will, and not as in free beer. The price you pay for this freedom is sometimes your patience and your time.

1.2. Logging In on a Unix-like Machine

For Windows users we recommend installing PuTTY. For OS X and Linux users, we recommend installing Mosh. Mosh is also available as a Chrome Extension.

2. Shell 101

Dear to the heart of a Unix-like operating system is a command-line interface with the operating system, often referred to as a "shell", or "terminal".

A Command-Line Interface (CLI) interprets textual commands (rather than mouse clicks, gestures, and alike). To this end, a CLI presents you with a "line" where you can enter text, and hereby "prompts" you to enter a command.

You can then type away at the keyboard, and hit the Enter key to "enter" a command. The CLI responds by trying to interpret your command, and if the command makes sense, by executing it. This execution may, or may not have a directly observable effect. If the execution terminates, you will be presented with another prompt, prompting for the next command.

The shell typically also has a Text-based User Interface (TUI), meaning that if there is something to tell, it will do so by means of text. It is important for a user-friendly experience that the utilities you use in your shell have good text-formatting defaults and options.

When first logging in on a headless Unix-like machine, you are greeted with a welcome message and a prompt. For example, when logging in our machine, you are greeted as follows:

```
Welcome to syrakuse.
Happy hacking!
alis@syrakuse:~$
```

In place of `alis` however, you will see the username you logged in with.

Try pressing enter a couple times,

```
Welcome to syrakyse.
Happy hacking!
alis@syrakuse:~$
alis@syrakuse:~$
alis@syrakuse:~$
alis@syrakuse:~$
```

This is how you "enter" the commands for this computer to execute.

The empty command is a little silly, so let's try something (slightly) less silly:

```
alis@syrakuse:~$ 42
-bash: 42: command not found
alis@syrakuse:~$ hello
-bash: hello: command not found
alis@syrakuse:~$ cowsay hello
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
alis@syrakuse:~$
```

`bash` here is the program that interprets your commands: `bash` here is your "shell". `42` and `hello` are not commands that this computer understands, but it understands `cowsay`. `cowsay` is a classic, silly little game we've put on this machine.

2.1. The Prompt

The line

```
alis@syrakuse:~$
```

is called a "prompt". This prompt shows the username you logged in with (in this case, `alis`), the hostname of the machine you logged in on (in this case, `syrakuse`), and the working directory of your shell (in this case, `~` (tilde)).

In the remainder of this document, we won't point out the username and hostname, as we do not intend on changing users or machines. Going forth, our prompts will simply look like this:

```
~$
```

2.2. The Working Directory

All major operating systems (both Unix-like and Windows) are centered around the idea of the user working with directories and files: a *file system*. In practice, this has two aspects:

1. every user has a some sort of a *home directory*; and
2. every program is working from the point of view of a particular directory: its *working directory*.

When you login on our machine, you land in a `bash` program, whose working directory is the home directory of the user you logged in with. The user's home directory has the alias ~ on Unix-like systems.

The following sections show how you can figure out what your working directory is, how to navigate the file system, create new directories, and change the working directory of `bash`.

2.3. `touch`, `ls`, and `rm`

There is an intricate relationship between your shell and your file-system. One of the most basic things you can do in a shell is to "touch" a file. If you touch a file that does not exist, it will be created. If you touch a file that already exists, its modification time is set to the current time.

Let's `touch` a couple files:

```
~$ touch hello
~$ touch world
~$ touch shell
```

We can now use `ls` to see what we've done to the working directory:

```
~$ ls
hello shell world
```

`ls`, without further arguments, lists all the entries in the working directory. (For now, all "entries" are files, but later we will also deal in directories).

By default, `ls` will sort the entries alphabetically. You can parametrise this behaviour by passing in command-line options. Command-line options are given as command-line arguments, separated from the command by at least one space character, and (conventionally) begin with either a `-` (dash) or `--` (double dash).

For instance, `ls -t` will list the entries ordered by their modification time:

```
~$ ls -t
shell world hello
```

Indeed, this corresponds to the order in which we touched the files above.

`touch` can also touch multiple files at once, with the filenames separated by at least one space character. For instance, we can touch `hello` and `world` with just one command:

```
~$ touch hello world
~$ ls -t
hello world shell
```

Space is special. If you want a space in your filename, you will have to either surround it with `'` (single quotes), `"` (double quotes), or *escape* the space character. For instance, the following commands are equivalent:

```
~$ touch 'hello, world'
~$ touch "hello, world"
~$ touch hello,\ world
```

Some other characters that we often escape are `'`, `"`, and `\` (backslash) itself. In general, there are many special characters around in a shell. To be on the safe side, stick to characters you know the meaning of when naming files in the shell.

If we try to `ls` again, things start getting cryptic:

```
~$ ls
hello        hello, world shell        world
```

How many files do we actually have here? To get a line-by-line directory listing, with one entry per line, we can pass in the option `-l` (small L) to `ls`:

```
~$ ls -l
total 0
-rw-r--r--  1 alis  alis  0 Aug 14 13:38 hello
-rw-r--r--  1 alis  alis  0 Aug 14 13:39 hello, world
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 shell
-rw-r--r--  1 alis  alis  0 Aug 14 13:38 world
```

This gives us a "long" directory listing, with quite a number of additional details. We explore these below. To start with, you can now count the number of files in the working directory by (manually) counting the number of lines that start with a `-` (dash).

The `-l` option can be mixed with `-t`. As a matter of convenience, all of the following are equivalent, and you can pick the mnemonic that makes most sense to you: `ls -l -t`, `ls -t -l`, `ls -lt`, and `ls -tl`. (If none of this makes sense to you, you can introduce an "alias". More about aliases later.)

For instance, `ls -lt` gives a long listing of the working directory, with entries ordered by their modification time:

```
~$ ls -lt
total 0
-rw-r--r--  1 alis  alis  0 Aug 14 13:39 hello, world
-rw-r--r--  1 alis  alis  0 Aug 14 13:38 hello
-rw-r--r--  1 alis  alis  0 Aug 14 13:38 world
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 shell
```

`ls` can also accept file system paths as command-line arguments. The effect is that it lists the paths given as arguments, instead of looking at the current working directory. As is convention, any command-line options to `ls` must precede the other command-line arguments. For instance, this shows that we touched the `shell` before we touched the `world`:

```
~$ ls -lt shell world
-rw-r--r--  1 alis  alis  0 Aug 14 13:38 world
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 shell
```

We used 3 different approaches to escaping the space character above. There are some subtle differences between these:

1. Arguments enclosed in `'` are interpreted literally. There is no escaping, and so you also cannot express `'` itself in an argument enclosed in `'`.

2. Arguments enclosed in `"` is interpreted literally, except for certain character sequences starting with `$`, ‘, and `\`. We will come back to `$` and ‘ later. `\` is used for escaping. For instance, if you want to use a `"` in an argument.
3. A `\` followed by a character will preserve the literal meaning of this subsequent character. There are some subtle differences between escaping inside `"` and outside, but we invite you to explore these at your leisure.

Things get even more intricate when you realise that you can mix and match all of these approaches. The following command are equivalent:

```
~$ touch "Here's a file with \"\" in the filename"
~$ touch Here\'s\ a\ file\ with\ \"\"\ in\ the\ filename
~$ touch Here\'s\ 'a file with "" in the filename'
~$ touch Here\'s" a file with \"\" in "'the filename'
```

From now on, we will refer to the use of `'`, `"`, and `\` collectively as “escaping”.

By now we’ve made a great big mess of our working directory. It is time to clean up a little bit. `rm` can help us clean up.

> **NB!** Files and directories removed using `rm` are nearly impossible to recover. `rm` is quick-and-dirty: It takes no backups, and cannot be undone. It also “removes” file-system entries by simply letting the file-system forget the location of the data on disk. Hence, it is sometimes *possible* to recover the data by scanning the raw disk device, but the general advice is: **measure twice, `rm` once**.

For instance, let us remove `world`:

```
~$ rm world
```

We can use `ls` or `rm` to verify that `world` indeed is gone:

```
~$ ls world
ls: world: No such file or directory
~$ rm world
rm: world: No such file or directory
```

As with `touch`, `rm` accepts multiple command line arguments, and deletes all the given file names. For instance, to remove the other silly little files:

```
~$ rm hello shell
```

How about the file with a long and complicated name? Would we really have to type out that mess again? As we said, there is an intricate relationship between your shell and your file system. The shell can help you type out file system paths quickly and correctly through a mechanism called “tab completion”.

Type `rm Here` and press TAB. This should complete the file name using the escape-character strategy described above. If the shell cannot make out what file system path you are trying to type, it will list the options for you, prompting you to enter another character, to break the ambiguity.

2.3.1. Exercises

1. What is the difference between `touch hello world` and `touch world hello`? If doubt, try it out.

2. Assuming you have the files `hello` and `world`, is there a difference between `ls hello world` and `ls world hello`? Why, or why not?

3. Create a couple files with names composed of just spaces (e.g., 1 space, 2 spaces, and 3 spaces). How does this change the readability of `ls` and `ls -l`? (Try also passing the option `--quoting-style=shell` to `ls`.)

4. Create a file `"hello,\ shell"` (containing both `"` and `\`).

   Hence, `ls -l` should show something like this:

   ```
   ~$ ls -l
   ...
   -rw-r--r-- 1 alis alis 0 Aug 14 13:37 "hello,\ shell"
   ...
   ```

2.4. Globbing

Consider touching the following files:

```
~$ touch hello world shell hello,\ world hello,\ shell
```

What if you want to do something with all files that begin with "hello"?

The `*` (star) character can be used to specify a set of files, without typing out their names in full.

For instance, to list just the files that begin with "hello":

```
~$ ls -l hello*
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 hello
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 hello, shell
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 hello, world
```

Similarly, we can list all the files that end with "world":

```
~$ ls -l *world
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 hello, world
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 world
```

`*` is part of a built-in shell mechanism called "globbing" — the use of patterns to specify sets of filenames.

To show that `*` is a built-in shell mechanism, try typing `ls s*` and press TAB. Assuming you only have one file starting with "s" in your working directory, namely "shell", this will expand the command to `ls shell`.

A `*` can be used both at the end, start, or in the middle of an argument, provided it is not otherwise escaped.

One peculiar use of `*` is as an argument to `ls`:

	Matches	Number of times
*	Any character.	Any number of times.
?	Any character.	Once.
[abc]	Either a, b, or c.	Once.
[0-9]	Either 0, 1, …, 9.	Once.
[!abc]	Neither a, b, nor c.	Once.
[!0-9]	Neither 0, 1, …, 9.	Once.

Table 1. Globbing patterns an their meaning.

```
~$ ls -l *
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 hello
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 hello, shell
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 hello, world
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 shell
-rw-r--r--  1 alis  alis  0 Aug 14 13:37 world
```

Note how this removes the peculiar "total" line from before, and you can now count the number of entries in the working directory simply by counting the number of lines output by `ls -l *`.

Some common globbing patterns are summarised in Table 1.

> **NB!** Control characters used in globbing patterns must be escaped if they are to be interpreted literally.

2.4.1. Exercises

For the purposes of the subsequent exercises, pl

```
~$ touch hello hallo haaallo
~$ touch 1 2 3 4 5 6 7 8 9
```

For each of the below exercises use a globbing pattern:

1. List all the entries that start with an `h`.
2. List all the entries that start with an alphabetic character (`a-z`).
3. List all the entries that do not start with an alphabetic character.
4. List all the entries that end with a `d`.
5. List all the entries that end with a `l` or `d`.
6. List all the entries that do not end with an `l` or `d`.
7. List just the files `hello` and `hallo`, but not `haaallo`.
8. List all the entries containing a comma.
9. Remove all the entries from your working directory.

2.5. `pwd`, `mkdir`, `mv`, and `tree`

`pwd` prints the *current* working directory:

```
~$ pwd
/home/alis
~$
```

In Unix-like operating systems, file system paths are separated by `/` (forward slash). (In Windows, they are separated by `\` (backward slash).) Furthermore, in Unix-likes systems, unlike in Windows, all directories and files reside in one file system, starting at `/`. This is called the *root directory*.

In this case, we see that the root directory has a subdirectory `home`, which has a sub-subdirectory `alis`. This is `alis`'s home directory. So `~` (in `alis`'s case) is really an alias for `/home/alis`.

`mkdir` can create new directories in our working directory:

```
~$ mkdir Europe
~$ mkdir Africa
~$ mkdir Antarctica Asia  Australia
~$ mkdir "North America" "South America"
```

As before, we can use `ls` to list the entries in the current working directory:

```
~$ ls
Africa   Antarctica   Asia   Europe  North America  South America
```

As before, we can use `ls -l` for a less cryptic listing:

```
~$ ls -l
total 28
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Africa
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Antarctica
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Asia
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Australia
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Europe
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 North America
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 South America
```

Let us go ahead and create some subdirectories: We would like to create directories like `Europe/Denmark`, `Europe/Germany`, etc. We can use tab completion to help us along, but we can also cycle through our command history using the arrow keys. This is useful when all you want to do is extend or repeat a recent command.

Try using the arrow keys when doing the following:

```
~$ mkdir Europe/Denmark
~$ mkdir Europe/Denmark/Copenhagen
```

Crucially, we cannot create the directory `Europe/Denmark/Copenhagen` before the `Europe/Denmark` directory.

Let's create a couple more directories to work with:

```
~$ mkdir Europe/Germany
~$ mkdir Europe/Germany/Berlin
```

```
~$ mkdir Europe/France
~$ mkdir Europe/France/Paris
~$ ls -l Europe
total 12
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Denmark
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Germany
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 France
~$ mkdir North\ America/United\ States
~$ mkdir North\ America/United\ States/Washington
```

Note, how if we type `ls -l E`, and press TAB, this will complete as `ls -l Europe/`, with the trailing slash indicating that `Europe` is a directory.

If our ultimate intent is to create a directory of continents, countries, and cities, `mkdir` comes with a convenient option out of the box: `mkdir -p` (`p` for parents) can be used to create a directory, and all the directories leading up to it, if they are missing:

```
$ mkdir -p Asia/China/Beijing
$ mkdir -p Asia/Japan/Tokyo
```

Yet another way to create a file system hierarchy is to create files and directories and move them around. To this end, we have an `mv` command:

```
~$ mkdir Aarhus Aalborg Odense
~$ mv Aarhus Aalborg Odense Europe/Denmark/
```

`mv` moves the files or directories at the given paths to whatever is given as the last argument.

Crucially, `mv` can also be used to rename (i.e., "move") entries (e.g., if you make a mistake):

```
~$ mkdir Phenix
~$ mv Phenix Phoenix
~$ mv Phoenix North\ America/United\ States/
```

Another, more recreational way to correct a mistake is to remove the directory, and try again. Removing directories is (intentionally) slightly harder than files. This is an attempt to remind you that you now might be deleting more than you think. Pass the option `-r` to `rm` to remove a directory:

```
~$ rm -r North\ America/United\ States/
```

You must use the `-r` option even if the directory is empty:

```
~$ rm Asia/Japan/Tokyo
rm: cannot remove 'Asia/Japan/Tokyo': Is a directory
~$ rm -r Asia/Japan/Tokyo
```

Finally, you can admire the file system hierarchy you created in all its glory using `tree`:

```
~$ tree
.
```

```
    Africa
    Antarctica
    Asia
        China
            Beijing
        Japan
    Australia
    Europe
        Denmark
            Aalborg
            Aarhus
            Copenhagen
            Odense
        France
            Paris
        Germany
            Berlin
    North America
    South America

19 directories, 0 files
```

Note how we only removed the `Tokyo` directory, but not its parent directory, `Japan`.

Executing `tree` without further constraints can be quite incomprehensible. Pass in the `-L` option followed by an integer argument to restrict the depth of the printed tree:

```
~$ tree -L 2
.
    Africa
    Antarctica
    Asia
        China
        Japan
    Australia
    Europe
        Denmark
        France
        Germany
    North America
    South America

12 directories, 0 files
```

2.5.1. Exercises

1. List the contents of the `/` directory.
2. List the contents of the `/home/` directory.
3. Create the directory `~/1/1/2/3/5/8/13/21`.
4. Create a directory `America` and move `North America` and `South America` to this directory.
5. Move the folders back out and remove the `America` directory.

2.6. `cd`

`cd` changes the current working directory:

```
~$ cd Europe
~/Europe$ ls -l
total 12
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Denmark
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 Germany
drwxr-xr-x 2 alis alis 4096 Sep 21 13:37 France
~/Europe$ pwd
/home/alis/Europe
~/Europe$
```

It is convenient that the shell's prompt lets you know where you stand. It almost makes it irrelevant to know the command `pwd`, as long as you know what `~` means.

`cd` with no arguments will lead you back home:

```
~/Europe$ cd
~$
```

Of course, you can also use the path alias `~`:

```
~$ cd Europe
~/Europe$ cd ~
~$
```

Let's go deeper:

```
~$ cd Europe/Denmark/Copenhagen
~/Europe/Denmark/Copenhagen$
```

How do we go back?

Every directory has a special directory `..`

If you `cd` to `..` you go "up" a directory in the file system hierarchy. We say that `..` refers to the *parent directory*.

```
~/Europe/Denmark/Copenhagen$ cd ..
~/Europe/Denmark$
```

`..` is actually a special, virtual directory that every directory has. For instance, here is the long way home:

```
~/Europe/Denmark$ cd ../../
~$
```

If every directory has a special directory `..` then how come we didn't see it in our directory listings above? This is because Unix-like convention has it that hidden files and directories start with a `.` (dot). Of course, there is nothing overly special about "hidden" files and directories (on Unix-like systems or Windows). We can get `ls` to show *all* the directories in a directory using the `-a` (meaning "all") option:

```
~$ ls -la
total 40
drwxr-xr-x 11 alis alis 4096 Sep 21 13:37 .
drwxr-xr-x  3 alis alis 4096 Sep 21 13:37 ..
drwxr-xr-x  3 alis alis 4096 Sep 21 13:37 1
drwxr-xr-x  4 alis alis 4096 Sep 21 13:37 Africa
drwxr-xr-x  2 alis alis 4096 Sep 21 13:37 Antarctica
drwxr-xr-x  2 alis alis 4096 Sep 21 13:37 Asia
drwxr-xr-x  2 alis alis 4096 Sep 21 13:37 Australia
drwxr-xr-x  2 alis alis 4096 Sep 21 13:37 Europe
drwxr-xr-x  2 alis alis 4096 Sep 21 13:37 North America
drwxr-xr-x  2 alis alis 4096 Sep 21 13:37 South America
~$
```

`.` is also special directory, it refers to the *current directory*.

It is notable, that the gobbling pattern `*` will not produce the virtual directories `.` and `..`.

2.6.1. Exercises

1. `cd ~/../../../../../../../` Where do you end up?
2. Go back home.
3. `cd ~/./../alis/Europe/./Denmark/../../` Where do you end up?
4. Create a hidden directory in your home directory.
5. Remove all entries from your home directory, except the continents.

   Now, move all the continents to a dedicated `Earth` directory, without explicitly listing them (i.e., use `*`).

   In the end, your home directory should looks like this:

   ```
   $ tree -L 2
   .
     Continents
         Africa
         Antarctica
         Asia
   ```

```
        Australia
        Europe
        North America
        South America

8 directories, 0 files
```

(Hint: Use a temporarily hidden directory.)

2.7. `echo`, `cat`, and `>`

`echo` is a program that can display a line of text. For instance:

```
~$ echo "Roses are red,"
Roses are red,
~$
```

Note, again how arguments containing spaces are surrounded by double quotes.

At first sight, `echo` is a rather useless program. This is where the power of a Unix-like operating system comes into play: your shell can *redirect* the output from a program to a file. To do this, follow the command by the a `>` (output redirection) character, followed by a path to the file where you want the output.

```
~$ echo "Roses are red," > roses.txt
~$
```

We can use `ls` to check to see what happened:

```
~$ ls -lah
...
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 "hello,\ shell"
-rw-r--r--  1 alis alis   15 Sep 21 13:37 roses.txt
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 Lynx
...
~$
```

So now there is something called `roses.txt` in our home directory. Unlike the directories we created before, the left-most character printed by `ls` is a `-` (dash), rather than a `d`. This indicates that `roses.txt` is *not* a directory.

`cd` can help us verify this:

```
~$ cd roses.txt
bash: cd: roses.txt: Not a directory
```

> **TIP** Sometimes, all you want to do is inspect the filesystem attributes of a given file. To do this, just pass in the path as an argument to `ls`:
>
> ```
> ~$ ls -lah roses.txt
> -rw-r--r--  1 alis alis   15 Sep 21 13:37 roses.txt
> ```

`cat` is a program that can print the contents of a file directly inside the shell:

```
~$ cat roses.txt
Roses are red,
~$
```

Let's create another file:

```
~$ echo "Violets are blue," > violets.txt
~$
```

> **NB** `>` will overwrite the file if it already exists.

`cat` can also *concatenate* files, and print the contents inside the shell:

```
~$ cat roses.txt violets.txt
Roses are red,
Violets are blue,
~$
```

Of course, we can redirect the output of any command, so we can store this, more complete poem in `poem.txt`.

```
~$ cat roses.txt violets.txt > poem.txt
~$ cat poem.txt
Roses are red,
Violets are blue,
~$
```

2.7.1. Exercises

1. Create a file `sugar.txt` with the line `Sugar is sweet,`
2. Create a file `you.txt` with the line `And so are you.`
3. Complete the poem in `poem.txt` by combining `roses.txt`, `violets.txt`, `sugar.txt`, and `you.txt` (in that order).

   It should be possible to do this in the end:

   ```
   ~$ cat poem.txt
   Roses are red,
   Violets are blue,
   Sugar is sweet,
   And so are you.
   ~$ cat violets.txt you.txt
   Violets are blue,
   And so are you.
   ~$
   ```

2.8. wc and |

`wc` prints the line, word, and byte count of a file.

```
~$ wc poem.txt
 4 13 65 poem.txt
~$
```

So our poem is 4 lines, 13 words, or 65 bytes in length.

> **TIP** This explains the `65` in the output of `ls -lah` for `poem.txt`:
>
> ```
> ~$ ls -lah poem.txt
> -rw-r--r-- 1 alis alis 65 Sep 21 13:37 poem.txt
> ```

The name `wc` is easy to remember if you think that it stands for "word count", although the program can do fair a bit more than that. In fact, it isn't even that good at counting words. Any sequence of non-whitespace characters is counted as a word. For instance, numbers are also counted as words. Your high-school teacher would not be happy with such a word count.

> **TIP** If you *just* want the line count for a file, use the `-l` option.
>
> ```
> ~$ wc -l poem.txt
> 4 poem.txt
> ~$
> ```

What if we wanted a word count of a silly little poem like this?

```
~$ cat violets.txt you.txt
Violets are blue,
And so are you.
~$
```

We could use output redirection to put the silly poem in a silly file, pass the filename to `wc`, and finally remove the silly file (more on this below); but this would indeed be rather *silly*: Why create a file in the first place?

We can *pipe* the output of one program as input to another. To do this, type the first command, a `|` (pipe) character, then the second command:

```
~$ cat violets.txt you.txt | wc
      2       7      34
```

`wc` with no arguments, counts the lines, words, and bytes passed to it via the pipeline. Now `wc` does not print a file name: There is no filename to print!

The silly poem is just 2 lines, 7 words, or 34 bytes in length.

Let's verify that this "pipe"-thing works by doing this with `poem.txt`:

```
~$ cat poem.txt | wc
      4      13      65
```

Except for the silly whitespace (more on how to handle this later), and the missing filename, the output is the same as with `wc poem.txt` above.

2.8.1. Exercises

1. Count the number of files and directories directly under the `/` directory.
2. Count the number of users on the system. (Hint: Recall what `~` is an alias for.)

2.9. `nano`

Although we can read and write files using our various command-line utilities, in conjunction with clever shell tricks, this mode of operation can get a little cumbersome. Text-editors are dedicated utilities to this end.

There are two classical text-editors in a Unix-like environment: `vim` and `emacs`. The users of one are often viciously dispassionate about the users of the other. More humble users use whatever suits the task at hand. For instance, `vi` (a slimmer, older version of `vim`) is available on most systems out of the box, and so `vim` proliferates in server environments, while `emacs` has flexible user-interface customization options, making it more suitable in a desktop environment.

Another text editor available on many systems out of the box is `nano`. To avoid duels over the choice of text-editor, and still teach you a somewhat ubiquitous tool, we decided to focus on `nano`.

To get started on the recreational activity of recovering `poem.txt`:

```
~$ nano poem.txt
```

Much like the shell, `nano` uses a text-based user interface (TUI). Unlike the shell, `nano` relies to a greater extent on *keyboard shortcuts*, rather than commands. Furthermore, the `nano` TUI is quite reminiscent of a classical graphical user-interface (GUI). We believe that this makes `nano` more suitable for beginners.

At the bottom of the TUI, `nano` shows a short reference of useful shortcuts:

```
^G Get Help     ^O Write Out     ^W Where Is     ...
^X Exit         ^R Read File     ^\ Replace      ...
```

Here, `^` indicates the `Ctrl` character. For instance, type `Ctrl+o` to save (i.e., "write out") the file you are editing. `nano` will now prompt you:

```
File Name to Write: poem.txt
```

Hit `Enter` to confirm and overwrite `poem.txt`. To exit `nano`, type `Ctrl+x`.

`vim` also uses a TUI by default, while `emacs` can be started in this mode with the command-line argument `-nw` (i.e., no window system). So `vim` and `emacs` can be used to similarly edit `poem.txt`, but they are far less friendly to beginners.

A typical problem that beginning users have is how exit either `vim` or `emacs` once they open them. In `vim`, you can press Esc to enter a so-called "command mode", enter the command `:q` and press Enter. In `emacs`, you use the keyboard sequence `Ctrl+x`, `Ctrl+c`. Figuring out how to edit and save files in either `vim` or `emacs` is left as an exercise for the bored reader. Else, continue with `nano`.

2.9.1. Exercises

1. Open `poem.txt` in `nano`.
2. Cut out the first line and paste it (uncut) at the bottom of the poem. Save the new file.
3. Determine the number of lines and characters in the poem using `nano`. How many characters are there in the longest line?
4. Copy the entire poem beneath itself without doing this line-by-line. Hint: use "Read File".

2.10. `history`

We have covered quite a number of commands. One way to revisit them is to read the text again. Another, is to take a look at the history of commands that you have executed in your shell. Besides, wouldn't it be nice to store these in a file so that you can later revisit what you did?

You can do this using the built-in `history` command:

```
~$ history
...
  566  cat violets.txt you.txt | wc
  567  cat poem.txt | wc
  568  nano poem.txt
  569  history
~$
```

This will likely show you quite a long list of commands. If this list goes off the screen, you can scroll using `Shift+PgUp` and `Shift+PgDn`.

Although we could execute `history` and redirect the output to a file, the numbers printed by `history` are a bit annoying.

To save your history to a file, you better use the `-w` option:

```
~$ history -w commands.txt
~$ cat commands.txt
cat violets.txt you.txt | wc
cat poem.txt | wc
nano poem.txt
history
history -w commands.txt
```

2.11. `--help`, `man`, and `less`

We have covered quite a number of options. How did we learn these options? Probably word of mouth, and the Internet. However, there is another useful source of documentation.

Most commands have a `--help` or `-h` option which you can use for a quick overview on how to use the command:

```
~$ history --help
```

Such help pages can be quite extensive, but the convention is to keep them short and to the point. It is then customary for the programmer to write a more extensive "manual" for the program. Such manuals can be browsed using the `man` program.

TODO: `man` and `less`.

2.12. Hello, Shell Script

Composing small utilities to form complicated commands is fun, but it is also hard work. We can save our work by encoding a command into a so-called shell script — a file containing shell commands. In effect, we are creating a utility of our own.

Let's walk through creating a shell-script for doubling the contents of a file.

First, open a file `double.sh` in `nano`. The `.sh` extension follows the convention that shell scripts should have the filename extensions `.sh`, although this does not really make it a "shell script".

`double.sh`, as a command-line utility, will take a command line argument (`$1`), regard it as a path to an existing file, `cat` this file twice into a temporary file (`$1.double`), and if this succeeds, move `$1.double` to `$1`, replacing the original file. Here, `$1` is a shell variable referring to the first command-line argument. If no such argument is given, `$1` is an empty string.

Write the following to `double.sh` using `nano`:

```
cat $1 $1 > $1.double
mv $1.double $1
```

Now, to run this shell script, pass it as an argument to the program `bash`:

```
~$ wc -l poem.txt
       4 poem.txt
~$ bash double.sh poem.txt
~$ wc -l poem.txt
       8 poem.txt
```

2.13. `file` and `which`

In a Unix-like operating system, everything is a file. Furthermore, it is the contents, or the metadata of a file (not e.g., a filename extension), that determines the *type* of a file.

The `file` utility exists to help users "guess" the type of a file. Its usage is simple:

```
~$ file poem.txt
poem.txt: ASCII text
```

```
~$ file double.sh
double.sh: ASCII text
~$ file 1/
1/: directory
~$ file .
.: directory
```

When you type a program name in your shell, this program must exist as an executable somewhere on your filesystem. You can use `which` to figure out what a given program name resolves to.

In Unix-like operating systems it is conventional to have short names (aliases) for more concrete programs. One such popular program is Python. Our server has Python version 2.7 installed, but it suffices to type `python` to start it up. The following sequence of `which` and `file` commands shows how we can figure out the concrete executable behind "`python`":

```
$ which python
/usr/local/bin/python
~$ file /usr/local/bin/python
/usr/local/bin/python: symbolic link to python2
~$ which python2
/usr/local/bin/python2
~$ file /usr/local/bin/python2
/usr/local/bin/python2: symbolic link to python2.7
~$ which python2.7
/usr/local/bin/python2.7
~$ file /usr/local/bin/python2
/usr/local/bin/python2: symbolic link to python2.7
~$ file /usr/local/bin/python2.7
/usr/local/bin/python2.7: ELF 64-bit LSB executable ...
```

The last line indicates that we've reached the actual executable that gets loaded into memory when we type `python` in our shell. Until then, we merely follow so-called "symbolic links".

2.13.1. Exercises

1. Which concrete executable does `sh` resolve to?
2. Which concrete executable does `bash` resolve to?
3. Which concrete executable does `which` resolve to?

3. Working with More Data

The world would be pretty boring if all you could do was write poems and mess about with your file system. It is time to get on the Internet!

3.1. `curl` and `less`

`curl` can fetch a URL and write the fetched data to your shell.

In 2014, OpenDNS, a domain-name service, published the top 10.000 URLs that their users had visited, ordered by popularity. We've mirrored this under https://dikunix.dk/~oleks/uldp17/domains.txt.

You can fetch this list using `curl`:

```
~$ curl https://dikunix.dk/~oleks/uldp17/domains.txt
```

> **TIP** Use `Shift` + `PgUp` to scroll up, and `Shift` + `PgDn` to scroll down in your shell.
>
> **NB!** On Apple computers, `PgUp` is `Fn+Up`, similarly for `PgDn`.

The 10.000 lines will blink before your eyes. It is unlikely that your shell will let you see all of the 10.000 lines vis-a-vis the **TIP** above. Your shell is trying to spare you the trouble.

If you would like to scroll around in the file to get a feel for its contents, you can pipe it over to `less`:

```
~$ curl https://dikunix.dk/~oleks/uldp17/domains.txt | less
```

`less` is a great tool for *reading* files, *especially* large files: unlike most text-editors, it does not lead the entire file into memory, but just enough to show the part visible on the screen. This is an inherent consequence of the goal of `less` to enable reading rather than *editing* text.

Type `h` to get help, and learn more about `less`.

Type `q` to quit.

3.1.1. Exercises

1. Make a local, snapshot copy of `domains.txt`, so as to not have to download the data every time you want to do something with it. Save it as `domains.txt` in your home directory.
2. Use `less` to read `domains.txt`.

3.2. `head` and `tail`

When you have a pretty long file, such as a web server access log, you sometimes want to just get a glance of the start of the file, or the end of the file.

`head` can be used to output the first part of a file:

```
~$ head domains.txt
google.com
facebook.com
doubleclick.net
google-analytics.com
akamaihd.net
googlesyndication.com
```

```
googleapis.com
googleadservices.com
facebook.net
youtube.com
~$
```

`tail` can be used to output the last part of a file:

```
~$ tail domains.txt
synxis.com
adyoulike.com
costco.ca
pressly.com
doorsteps.com
clkbid.com
cyveillance.com
musicnet.com
mrnumber.com
arenabg.com
~$
```

For both `head` and `tail` you can specify the number of first or last lines to print, using the `-n` option:

```
~$ head -n 20 domains.txt
... (first 20 lines of domains.txt)
~$ tail -n 20 domains.txt
... (last 20 lines of domains.txt)
~$
```

3.2.1. Exercises

1. What are the top 100 most-visited URLs?
2. What are the bottom 10 of the top 100?
3. What is the 100th most-visited URL?

3.3. `cut` and `paste`

You will find another dataset under `/var/www/logs/normalized.log`. This a normalized version of `/var/www/logs/access.log`, the access log of our web server. Later, we will ask you to perform this normalization using a shell a script.

Copy both logs to your home directory.

> **NB!** If you do not have access to our server, ask us, or a fellow student for a copy. Please do not make this log public, as it may contain personally identifiable information.

Looking at the file, we notice some silly `"` (double quotes) in the log file. They separate the 4 fields of the log file:

1. IP (version 4) address of the requesting party.
2. Timestamp of request.
3. A description of the request.
4. A so-called *browser string*: What the requesting party otherwise tells about itself: Typically some sort of a browser, program, or crawler name.

This type of file is typically called a CSV-file. CSV stands for comma-separated values. You are probably familiar with Microsoft Excel, Google Sheets, Apple Numbers, or LibreOffice Calc. A CSV file maps naturally to a "spreadsheet": each row is a line in the file, with the columns separated by a "comma".

The name CSV is unfortunate. Oftentimes, the separator is a semi-colon (`;`), and in our case a double quote (`"`). People use whatever separates the columns best (you will soon see how to do this). We use a double quote because double quotes can't occur in either a URL or a browser string.

`cut` is a utility that can cut up the lines of a file.

To get just the IP addresses of parties that have tried to access the server, we can cut up each line at `"` and select the first field:

```
~$ cat normalized.log | cut -d\" -f1
...
213.211.253.38
213.211.253.38
130.225.98.193
...
~$
```

You can select multiple fields, but they are always selected by `cut` in increasing order:

```
~$ cat normalized.log | cut -d\" -f1,4
...
213.211.253.38"x00_-gawa.sa.pilipinas.2015
213.211.253.38"x00_-gawa.sa.pilipinas.2015
130.225.98.193"curl/7.44.0
...
~$ cat normalized.log | cut -d\" -f4,1
...
213.211.253.38"x00_-gawa.sa.pilipinas.2015
213.211.253.38"x00_-gawa.sa.pilipinas.2015
130.225.98.193"curl/7.44.0
...
~$
```

If we would like to reorder the columns, we can use `paste`. `paste` places each line of a given file along the corresponding line of each subsequent file.

```
~$ cat normalized.log | cut -d\" -f1 > ips.txt
~$ cat normalized.log | cut -d\" -f4 > user_agents.txt
~$ paste ips.txt user_agents.txt
```

```
...
x00_-gawa.sa.pilipinas.2015    213.211.253.38
x00_-gawa.sa.pilipinas.2015    213.211.253.38
curl/7.44.0    130.225.98.193
...
```

By default, `paste` uses TAB as a separator. If you would like a custom separator, you can use the `-d` option:

```
~$ paste -d\" ips.txt user_agents.txt
...
x00_-gawa.sa.pilipinas.2015"213.211.253.38
x00_-gawa.sa.pilipinas.2015"213.211.253.38
curl/7.44.0"130.225.98.193
...
```

3.3.1. Exercises

1. An IP (version 4) address is composed of 4 fields, each a number between 0 and 255, typically separated by a `.` (dot). List the first field of every IP address in the access log.

2. The server logs the time of access in the following format:

   ```
   day/month/year:hour:minute:second zone
   where
     day = 2*digit
     month = 3*letter
     year = 4*digit
     hour = 2*digit
     minute = 2*digit
     second = 2*digit
     zone = (`+' | `-') 4*digit
   ```

   List all the dates of the month on which the log was accessed.

3. The HTTP request itself is described in the third column of the log file. It consists of 3 things:

   a. The HTTP Method used. Typically `GET`.
   b. The resource that was requested. This is typically the tail of the URL that the crawler used, i.e. the URL after the leading `https://uldp16.onlineta.org` has been cut off.
   c. The HTTP version used. Should be `HTTP/1.1` or `HTTP/1.0`.

   List all the resources that were requested.

4. Write a shell script `normalize.sh` that turns `access.log` into `normalized.log`. (Hint: First cut out the parts you need into some temporary files. Then, paste those files together.)

3.4. sort and uniq

`sort` is a utility that can sort the lines of a file and output the sorted result.

```
~$ cat normalized.log | cut -d\" -f4 | sort
...
curl/7.44.0
...
x00_-gawa.sa.pilipinas.2015
x00_-gawa.sa.pilipinas.2015
...
~$ cat normalized.log | cut -d\" -f1 | sort -n
...
130.225.98.193
213.211.253.38
213.211.253.38
...
~$
```

The `-n` option tells `sort` to sort in *alphanumeric* order rather than *lexicographic* order. For instance, in lexicographic order, `213.211.253.38` comes *before* `36.225.235.94`. In alphanumeric order, it comes *after*.

See also the man page for `sort` for other useful sorting options.

`uniq` is a utility that can remove the immediate duplicates of lines. If you have a sorted file, you can use `uniq` to output the unique lines of the original file.

```
~$ cat normalized.log | cut -d\" -f1 | sort -n | uniq
...
130.225.98.193
213.211.253.38
...
~$
```

`uniq` also has the rather useful option that it can count the number of duplicate occurrences before it deletes them:

```
~$ cat normalized.log | cut -d\" -f1 | sort -n | uniq -c
...
     11 130.225.98.193
      4 213.211.253.38
...
~$
```

3.4.1. Exercises

1. List the unique browser strings.
2. Count how many times each browser string occurs.
3. Count the total number of unique IP addresses that have tried to access the server.
4. What are the different HTTP methods that have been used?
5. Count the number of requests made in every day for which the access log has data.
6. Count the number of *unique* requests made in every day for which the access log has data.
7. List the number of requests in each hour of the day (use the same time zone as the server, so just cut out the hour of every entry in the log).

4. Search, Replace, and Compare

4.1. `grep`

Let's say we want to find all the requests that were made using (supposedly) the Google Chrome web browser.

`grep` is a utility that can print the lines matching a pattern.

> **NOTE** `grep` is a slightly old utility, and so we will always use the more modern, *extended* variant of `grep`, be specifying the `-E` option. This is equivalent to using the `egrep` utility instead.

```
~$ cat normalized.log | grep -E "Chrome"
...
130.225.98.193...Chrome/45.0.2454.85 Safari/537.36
...
~$
```

The "patterns" that you can specify to `grep` are called *regular expressions*. Regular expressions offer a powerful syntax for matching and replacing strings. They deserve a special role in theoretical Computer Science. We will only take a look at some simple, practical elements of regular expressions.

For instance, the Internet Explorer web browser is typically identified by the string `Trident` occurring in the browser string. Here's how you would find all those requests that were made using *either* Google Chrome or Internet Explorer:

```
~$ cat normalized.log | grep -E "Chrome|Trident"
...
130.225.98.193...Chrome/45.0.2454.85 Safari/537.36
...
```

```
104.148.44.191...Windows NT 6.1; Trident/5.0
...
~$
```

(If this doesn't work, i.e. we have no Internet Explorer users in the audience, try `Firefox` instead.)

The character `|` is a regular expression *metacharacter*, and serves to say that the line should contain the string either `Chrome` *or* `Trident`.

The metacharacter `|` can be used multiple times. For instance, the Firefox web browser typically identifies itself with the string `Gecko`. Here is how you would find all the requests made with either Google Chrome, Internet Explorer or Firefox.

```
~$ cat normalized.log | grep -E "Chrome|Trident|Gecko"
...
130.225.98.193...Chrome/45.0.2454.85 Safari/537.36
...
104.148.44.191...Windows NT 6.1; Trident/5.0
...
109.200.246.88...Gecko/20100101 Firefox/9.0.1
...
~$
```

You can also ask `grep` for the inverse match, i.e. those lines not matching the given pattern using the `-v` option:

```
~$ cat normalized.log | grep -Ev "Chrome|Trident|Gecko"
...
130.225.98.193...curl/7.44.0
...
213.211.253.38...x00_-gawa.sa.pilipinas.2015
...
122.154.24.254...-
...
~$
```

> **TIP** To get `grep` to match in a *case-insensitive* manner. e.g. to match strings like `chrome` even though the pattern says `Chrome`, use the `-i` option. (An example follows further below.)

A regular expression is a string of characters. Every character is either a metacharacter, or a literal character. For instance, in the regular expression `Chrome`, all characters are literal characters. Here is an overview of a couple useful metacharacters:

Metacharacter	Meaning
‘	‘ (pipe)
`^` (caret)	Start of string
`$` (dollar)	End of string
`.` (dot)	Any character

If you actually want to match what is otherwise a metacharacter, you will need to *escape* it. Similarly to strings in `bash`, this is done by prefixing the metacharacter with the metacharacter `\`. For instance, `\|`, `\^`, `\$`, and `\.`.

Consider our little poem from before:

```
~$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
~$
```

Here is how we would find all the lines ending with a `.` (dot):

```
~$ cat poem.txt | grep -E "\.$"
And so are you.
```

What if we wanted to find all the strings ending with a `.` (dot) *or* a `,` (comma)? To this end, we could use *character groups*. A character group is a list of characters acceptable at a given position in the string.

Character groups are perhaps best shown by example. All the following regular expressions match a single digit, i.e. a string that is either `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`:

1. `^[01234569]$`
2. `^[0-9]$`
3. `^[0-56-9]$`

A computer represents characters as numbers. There exists a range of standard ways of encoding characters as numbers, with perhaps the most ubiquitous being the UTF-8 character encoding.

Character ranges are based on these numeric representations of the characters involved. For instance:

1. The range `[a-z]` will contain the lower-case English alphabetic characters.

2. The range `[a-zA-Z]` contains both lower-case and upper-case English alphabetic characters.

3. The range `[a-zæøå]` contains the lower-case Danish alphabetic characters.

4. The range `[a-zæøåA-ZÆØÅ]` contains the lower and upper-case Danish alphabetic characters.

5. The range `[z-a]` contains no characters because the character code for `z` is larger than the character code for `a`.

   TIP If you want to quickly find the character code of a particular character, right-click in your Internet browser, e.g. Google Chrome, and bring up the HTML inspector. (Typically called something like "Inspect this element".) In the inspector window, you should be able to find a JavaScript *console*. Enter the following command in the console:

```
> "A".charCodeAt(0)
65
```

Use any character you'd like in place of `A`.

These regular expressions use a couple new metacharacters:

Metacharacter	Meaning
`[`	Start a character group
`-`	Character range[2]
`]`	End a character group

Getting back to our original problem, here is how we would find all the lines ending with a `.` (dot) *or* a `,` (comma) in our little poem:

```
~$ cat poem.txt | grep -E "[\.,]$"
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
```

How about those lines beginning with a flower?

This requires another little regular expression construct called *capturing groups*. Why these groups are called "capturing" will become clear shortly. For now, let us see how they help us solve our problem: A capturing group allows us to put a regular expression inside a regular expression, and treat it like a character.

For instance, here is how we could find all those lines that begin with a flower (note the use of the case-insensitive option):

```
~$ cat poem.txt | grep -Ei "^(roses|violets)"
Roses are red,
Violets are blue,
~$
```

This leaves us with the following additional metacharacters:

Metacharacter	Meaning
`(`	Start a capturing group
`)`	End a capturing group

Last, but not least, the elements of a regular expression can be quantified, to indicate that some parts of the string repeat a number of times.

Suffix	Meaning
`*`	This repeats 0 or more times
`+`	This repeats 1 or more times
`{` n `}`	This repeats exactly n times
`{` n `,}`	This repeats n or more times
`{` n `,` m `}`	This repeats between n and m times

Here `*`, `+`, `{`, `,`[3], and `}` are all metacharacters.

For instance, to find all the entries in the access log, with the browser string indicating that the requesting party is using Google Chrome on Linux, we could do this:

```
~$ cat normalized.log | grep -E "Linux.*Chrome"
...
...(X11; Linux x86_64)...Chrome/45.0.2454.85 Safari/537.36
...
~$
```

Here we accept any character between the *substrings* `Linux` and `Chrome`, 0 or more times. The result is those strings that contain both `Linux` and `Chrome` (in that order).

> **TIP** The above quantifiers are *greedy*: They will consume all possible occurrences. Sometimes, what you want is to consume up until the first occurrence of the next string. Follow the quantifier with `?` to make it *non-greedy*.

4.1.1. Exercises

1. How many requests were made from Google Chrome?

2. How many requests were made from neither Google Chrome, Internet Explorer, or Firefox?

3. If the visitor does not supply a browser string, our web server writes the browser string `-` (dash) to the log. Find the IP addresses of those visitors that do not supply a browser string.

4. Complete the first 14 lessons on http://regexone.com/. The tool is very permissive, and there are multiple answers which will be deemed “correct”. For every lesson, try to come up with as many ways as possible to get the tool mark your answer as “correct”.

4.2. `sed` or `perl`

`sed` is a tool for transforming streams of data. Transformation can take place using regular expressions.

[3]The `,` character only acts as a metacharacter between `{` and `}`.

> **NOTE** `sed`, like `grep`, is a pretty old tool, so we will also always use the more modern, *extended* variant of `sed`, using `sed -E`.
>
> Some operating systems (most notably, OS X) do not have an extended variant of `sed`. In most cases, it is safe to replace all occurrences of `sed -E` below with `perl -pe`.

We can use `sed` for a lot of things (e.g., to play Tetris). We will only use `sed` for replacing the substrings of every line in a file, matching a regular expression with something else. The general way to call `sed` when doing this is as follows:

`~$ sed -E "s/` regular expression `/` replace expression `/g"`

For instance, we can replace all occurrences of `"` in our log-file with `___`.

```
~$ cat normalized.log | sed -E "s/\"/___/g"
...
109.200.246.88___2015-09-20T18:37:08-04:00___...
...
~$
```

The second argument `sed` (e.g. `"s/\"/___/g"`) is a `sed`-command. We are only concerned with search-and-replace commands. These commands start with an `s`, and are followed by a regular expression and a *replace expression*, started, separated, and ended by a `/`. (So `/` is now also a metacharacter, and must otherwise be escaped.) The final `g` signifies that we want to replace *all* matching substrings on every line (mnemonic: `g` for *global*).

A more interesting use of `sed` is using capturing groups. Capturing groups are so-called because the substrings they capture become available in a replace expression. To insert a substring matched in a capturing group, type `\` followed by the number of capturing group in the regular expression, numbered from left to right, starting at 1.

For instance, instead of using `cut`, we could've used `sed` to cut up our file:

```
~$ cat normalized.log | sed -E "s/^(.*)\"(.*)\"(.*)\"(.*)$/\1/g"
...
93.174.93.218
140.117.68.161
...
~$ cat normalized.log | sed -E "s/^(.*)\"(.*)\"(.*)\"(.*)$/\2/g"
...
2015-09-20T13:08:52-04:00
2015-09-20T13:52:37-04:00
...
~$
```

Unlike with `cut`, with `sed` we can even reorder the fields!

```
~$ cat normalized.log | sed -E "s/^(.*)\"(.*)\"(.*)\"(.*)$/\2\"\1/g"
...
2015-09-20T13:52:37-04:00"104.148.44.191
2015-09-20T13:52:38-04:00"104.148.44.191
```

```
...
~$
```

4.2.1. Exercises

1. The timestamps in our access log are in ISO 8601 format. List all the unique access dates in the format date-month-year.
2. List all the unique access times in the format hour.minte.
3. Replace the ISO 8601 date and time by the above. Don't change any other fields in the log.
4. Strip all digits and punctuation from the browser string.
5. List the number of requests made from each unique IP. Use following format: IP (two spaces) count. So, strip the leading spaces and swap the columns that you get after `uniq -c`.
6. List the number of requests in each hour of the day (use the same time zone as the server, so just cut out the hour of every entry in the log). Use following format: hour (two spaces) count. So, strip the leading spaces and swap the columns that you get after `uniq -c`.

4.3. Other Languages

Most general-purpose programming languages come with some sort of a regular expression engine. Some are more advanced than others, sometimes breaking the otherwise elegant theory of regular languages (this theory is beyond the scope of these notes). The syntax of regular expressions sometimes also varies.

You can find a good overview of regular expressions, and what they look like in various languages at http://www.regular-expressions.info/.

4.4. `cmp`, `comm`, and `diff`

Consider having to compare the IP-addresses of Chrome and Firefox users:

```
~$ cat normalized.log | grep "Chrome" | cut -d\" -f1 | sort | uniq > chrome.txt
~$ cat normalized.log | grep "Firefox" | cut -d\" -f1 | sort | uniq > firefox.txt
```

`cmp` can be used to compare files, byte-by-byte:

```
~$ cmp chrome.txt firefox.txt
chrome.txt firefox.txt differ: char 2, line 1
```

If the files are exactly alike, `cmp` outputs nothing.

`comm` (meaning, "common") can be used to compare files, line-by-line.

Given paths to two files as arguments, `comm` will report which lines only occur in the first file, which lines only occur in the second, and which lines occur in both. The output is arranged in 3 columns, having this meaning:

```
~$ comm chrome.txt firefox.txt
107.77.253.7
109.17.203.138
109.56.107.144
        118.193.31.222
        130.225.165.46
                130.225.188.33
                130.225.188.65
        130.225.238.1
        130.225.98.201
138.197.65.174
152.115.128.10
...
```

Individual columns can be suppressed using the arguments `-1`, `-2`, and `-3`, respectively:

```
~$ comm -2 -3
107.77.253.7
109.17.203.138
109.56.107.144
138.197.65.174
152.115.128.10
~$ comm -1 -3
118.193.31.222
130.225.165.46
130.225.238.1
130.225.98.201
~$ comm -1 -2
130.225.188.33
130.225.188.65
62.61.128.183
```

The last command above gives us the IP addresses that have used both Chrome and Firefox to request resources from our web-server.

`diff` (meaning, "difference") can also be used to compare files, line-by-line. The output of `diff` however, contains more technical metadata, making it easier for automated tools to merge files together. For instance, `diff` is a basic building block of the git version control system.

You might be familiar with "diffs" from using git or browsing GitHub. The output of a `diff` command can be a bit cryptic to a human, and the level of detail may vary, depending on the options. An easy to remember `-ruN` option gives you an output similar to what you would find on GitHub:

```
~$ diff -ruN chrome.txt firefox.txt
--- chrome.txt  Tue Aug 15 20:28:46 2017
+++ firefox.txt Tue Aug 15 20:24:18 2017
@@ -1,35 +1,15 @@
```

```
-107.77.253.7
-109.17.203.138
-109.56.107.144
+118.193.31.222
+130.225.165.46
```

5. Shell Scripting

One useful aspect of a Unix-like environment is the ease of turning manual workflows into automated scripts. If you find yourself typing some specific sequence of commands frequently, you can put them in a file (e.g., `myscript.sh`), and run it:

```
~$ sh myscript.sh
```

This will cause the contents of the script to be executed just as if you had typed it yourself. (Actually, there are some exotic differences — primary among these is that using `cd` in the script will not have an effect once the script ends.) The `.sh` extension is irrelevant to the operating system — you can call it whatever you want and it will still work.

If you want to make your script feel even more like an ordinary program, there are two things you must do:

1. You should add a *shebang* as the first line of the file:

   ```
   #!/usr/bin/env sh
   ```

   A shebang tells the operating system what kind of code is contained in the file[4]. In this case, we tell it to execute it using whatever `sh` maps to in our current environment (equivalent to typing `sh` in the terminal).

2. You should mark the file as *executable*, or else the operating system will refuse to start it. This can be done with the `chmod` command:

   ```
   ~$ chmod +x myscript.sh
   ```

Now we can launch the script without explicitly invoking `sh`, just by passing the path to the script:

```
~$ ./myscript.sh
```

In this case, the path indicates the current directory, but if the script was somewhere else, we could run it as:

```
~$ ~/scripts/myscript.sh
```

If you want to be able to run the script just by typing `myscript.sh`, you must add its containing directory to the `$PATH` environment variable. We will briefly cover this later.

[4] Windows is not fond of shebangs, and instead relies on the filename extension.

5.0.1. Exercises

Write a shell script for each of the following. Latter scripts can (and probably should) refer to earlier scripts.

1. List the IP addresses that accessed the server.
2. Count how many times each unique IP address occured.
3. Get the requests made (supposedly) from Firefox.
4. Get the IP addresses of Firefox all (supposed) users.
5. Replace the ISO 8601 dates as we did above.

5.1. Basic Control Flow

Copying commands into a file is a good starting point, but shell scripting first becomes a truly powerful tool when you add *control flow* (conditionals and branches) to your scripts. While much less powerful than a conventional programming language, a shell script is a convenient way to automate workflows that involve Unix-like programs.

In fact, control flow is so esoteric in shell scripting that many users resort to Unix-like utilities to achieve something comparable to higher-order functional programming (e.g., `find` with the `-exec` argument, which we will cover later).

First, it is important to understand the notion of an *exit code*. Every program and shell command finishes by returning a number in the interval 0 to 255. By default, this number is not printed, but it is stored in the *shell variable* `$?`. For example, we can attempt to `ls` a file that does not exist:

```
~$ ls /foo/bar
ls: cannot access /foo/bar: No such file or directory
```

To print the exit code of the above command, we echo the `$?` variable:

```
~$ echo $?
2
```

By convention, the exit code 0 means "success", and other exit codes are used to indicate errors. In this case, it seems that `ls` uses the exit code `2` to indicate that the specified file does not exist. It may use a different exit code if the file exists, but you do not have permission to view it.

Note what happens if we check the `$?` variable again:

```
~$ echo $?
0
```

The exit code is now 0! This is because the variable `$?` is overwritten every time we run a command. In this case, it now contained the exit code of our first `echo` command, which completed succesfully. It is often a good idea to save away the value of `$?` in some other shell variable, as most commands will overwrite `$?`.

Typically, we are not much concerned with the specific code being returned — we only care whether it is 0 (success) or anything else (failure). This is also how the shell `if-then-else` construct operates. For example:

```
~$ if ls /foo/bar ; then echo ':-)'; else echo ':-('; fi
ls: cannot access /foo/bar: No such file or directory
:-(
~$ if ls /dev/null ; then echo ':-)'; else echo ':-('; fi
/dev/null
:-)
```

The semicolons are necessary to indicate where the arguments to `echo` stop. We could also use a line break instead.

5.1.1. A Simple Testing Program

Let us try to write a simple practical shell script. We want to create a program, `utest.sh`, that is invoked as follows:

```
~$ sh utest.sh prog input_file output_file
```

`utest.sh` will execute the program `prog` and feed it input from `input_file`. The output from `prog` will be captured and compared to `output_file`, and if it differs, `utest` will complain. Essentially, we are writing a small testing tool that is used to test whether some program, given some input, produces some specific output. You could use it to create tests for the log file analysis scripts you wrote previously.

I will go through `utest.sh` line by line. Some new concepts will be introduced as necessary.

```
#!/usr/bin/env sh
```

First we have the shebang. While not strictly necessary, it is good style.

```
if test $# -lt 3; then
  echo "Usage: <program> <input> <output>"
  exit 1
fi
```

The *number* of arguments given to the script is stored in the variable `$#`. We use the `test` program to check whether it is less than (`-lt`) 3. If so, we complain to the user and exit with a code indicating error.

```
program=$1
input=$2
output=$3
```

The arguments to the script are stored in variables named `$n`, where `n` is a number. Using such ill-descriptive names is a sure way to encrypt your shell-script. We create new variables with more descriptive names for holding the arguments.

```
if ! ($program < $input | cmp $output /dev/stdin); then
    echo 'Failed.'
fi
```

A lot of things are happening here. Let's take them one by one.

```
$program < $input
```

This runs the program stored in the variable `$program` with input from the file named by the variable `$input`.

```
($program < $input | cmp $output /dev/stdin)
```

We pipe the output of `$program` into the `cmp` program. At its most basic operation, the `cmp` program takes two files as arguments, and prints in-how-far they differ. In this case, the first file is `$output` (remember, the file containing *expected* output), as well as the special pseudo-file `/dev/stdin`, which corresponds to the input read from the pipe.

```
if ! ($program < $input | cmp $output /dev/stdin); then
    echo 'Failed.'
fi
```

The entire pipeline is wrapped in parentheses and prefixed with `!`. The `!` simply inverts the exit code of a command - this is because `cmp` returns 0 ("true") if the files are *identical*, while we want to enter the branch if they are *different*.

5.1.2. Exercises

1. Turn your command lines for the previous set of exercises into shell scripts.
2. Use `utest.sh` to test your shell scripts.
3. Write a master test script that automatically runs `utest.sh` on all your scripts and their corresponding input-output files.
4. Learn about shell script `while`-loops and the `shift` command, and write a version of `utest.sh` called `mtest.sh` that can operate on any number of test sets. The script should take as argument the program to test, followed by input-output file pairs. For instance,

   ```
   ~$ ./mtest.sh prog in.1 out.1 in.2 out.2
   ```

5.2. PATH

Previously, you learned how we can use `which` and `file` to find the concrete executable behind a shell command. To make your shell scripts feel as native as other shell commands, you should add the directory containing those (executable) shell scripts to your `PATH` environment variable.

One way to do this is to do this in your `~/.profile`. This file is loaded by most shells *upon login*. That is, after modifying this file, it is best you log out and log in again.

For instance, to add your home directory to your `PATH`, add the following line at the end of your `~/.profile`:

```
export PATH="$HOME:$PATH"
```

The directories are separated by `:` in `PATH`, whole `$HOME` and `$PATH` are shell variables for your home directory and current `PATH` environment variable. The `export` qualifier makes sure to export this change to your environment variables which persist after `~/.profile` is read.

Remember to log out and log in again to see the changes applied. Now you should be run shell scripts in your home directory from anywhere, and without prefixing them with `./`, or more treacherous paths.

5.3. Exercise Series: `tmpdir`

Many programs need some sort of "scratch-space". In such contexts, it would be useful to have a `tmpdir` utility which creates a temporary directory, runs a given program to completion, and deletes the temporary directory in the end.

The section below describes the `mktemp` utility, a standard utility for creating temporary directories and files. See this first.

Write a shell script, `tmpdir`, which when executed as follows:

```
~$ tmpdir a b c
```

Does the following:

1. Create a temporary directory with the template `/tmp/mytempdir-XXXXXX`.
2. `cd` into that directory.
3. Execute `a b c`.
4. `cd` back to the original working directory.
5. Remove the temporary directory, without regard for its contents.

Hint: Use a subshell to get the current working directory.

For instance, the following showcases how the temporary directory only persists for the duration of the `tmpdir` command:

```
~$ tmpdir pwd
/tmp/mytempdir-oFyIZZ
$ file /tmp/mytempdir-oFyIZZ
...: cannot open `/tmp/mytempdir-oFyIZZ' (No such file or directory)
```

Next, try a variation, so that when executed as follows:

```
~$ tmpdir a b c
```

`tmpdir` does the following:

1. Create a temporary directory with the template `/tmp/mytempdir-XXXXXX`.
2. Execute `a b c TMPDIR`, where `TMPDIR` is the path to the above directory.
3. Remove the temporary directory, without regard for its contents.

For instance,

```
~$ tmpdir echo The tempdir is
The tempdir is /tmp/mytempdir-zGPMAL
```

Next, try another variation, so that when executed as follows:

```
~$ tmpdir a b c
```

`tmpdir` does the following:

1. Create a temporary directory with the template `/tmp/mytempdir-XXXXXX`.
2. Execute `a TMPDIR b c`, where `TMPDIR` is the path to the above directory.
3. Remove the temporary directory, without regard for its contents.

For instance,

```
~$ tmpdir echo is the tempdir
/tmp/mytempdir-vDwq3e is the tempdir
```

Hint: Use `shift`.

Last, but not least, try a variation combining all of the above. Let `tmpdir` take one optional parameter, indicating which of the 3 above variations it should perform. The options are as follows:

- `-c` - `cd` into the temporary directory before executing the command.
- `-#` - the path to the temporary directory is given as the last argument to the executed command.
- `-1` - the path to the temporary directory is given as the first argument to the executed command.

By default, if no argument is given, assume `-c`.

For instance,

```
~$ tmpdir -c pwd
/tmp/mytempdir-udBkJX
~$ tmpdir -# echo The tempdir is
The tempdir is /tmp/mytempdir-IjvJPf
~$ tmpdir -1 echo is the tempdir
/tmp/mytempdir-8VmA46 is the tempdir
~$ tmpdir pwd
/tmp/mytempdir-gcpAc8
```

5.4. mktemp

The `mktemp` utility can be used to create a temporary file or directory. Since many such files or directories may exist under any given directory, `mktemp` takes a template intended for an application specific path, and filename prefix and suffix.

For instance, to create a temporary file in the current working directory:

```
~$ mktemp testing-mktemp-XXXX
testing-mktemp-Omrx
```

```
$ file testing-mktemp-Omrx
testing-mktemp-Omrx: empty
```

To create a directory, pass in the `-d` option:

```
~$ mktemp -d testing-mktemp-XXXX
testing-mktemp-7vPK
~$ file testing-mktemp-7vPK
testing-mktemp-7vPK: directory
```

A typical place to put temporary files and directories under the `/tmp/` directory on a Unix-like system. This is the typical "mount-point" of a `tmpfs` filesystem, i.e. a filesystem dedicated to temporary storage. Such a filesystem is often in-memory only, and so temporary files and directories are never stored on disk, and are purged upon reboot.

5.4.1. Exercises

1. Create a temporary directory under `/tmp/` using `mktemp`. Delete it.

6. Applications

6.1. ImageMagick

ImageMagick®, is a suite of tools to create, edit, and compose images, using either raster or vector graphics.

(Coming soon.)

6.2. `gnuplot`

`gnuplot` is a command-line driven graphing utility. `gnuplot` is installed on our server, but you can also go ahead and install it locally. Some (Linux) p:latforms have multiple versions of `gnuplot` based on different graphical user interface (GUI) frameworks. The choice, for our purposes, does not matter.

If you run `gnuplot` in the wild it will attempt to start up a GUI once there is a plot to show. Since we have no GUI capabilities when connected to our server, we want to suppress this default behaviour. Initial user-level settings for `gnuplot` can be specified in your `~/.gnuplot` file.

To ask `gnuplot` to plot graphs in ASCII, add this line to your `~/.gnuplot`:

```
set terminal dumb
```

We can now try to plot something in the terminal:

```
$ echo "plot sin(x)" | gnuplot
```

or equivalently,

```
$ gnuplot -e "plot sin(x)"
```

To plot, multiple things at once, we can separate them by comma's:

```
$ gnuplot -e "plot sin(x), cos(x)"
```

To limit the x-axis range of the plot, we can specify this range after the `plot` command:

```
$ gnuplot -e "plot [-5:5] sin(x), cos(x)"
```

Alternatively, we can issue a command to set the x-axis range before calling `plot`:

```
$ gnuplot -e "set xrange [-5:5]; plot sin(x), cos(x)"
```

Similarly, we can set the y-axis range:

```
$ gnuplot -e "plot [-5:5] [-5:5] sin(x), cos(x)"
```

`plot` is typically the last command you want to issue to `gnuplot`, preceding this with a number of plot configurations. It can get a little long-winded to type those in as part of a shell command, so `gnuplot` commands are typically stored in separate files.

```
$ gnuplot -e "set xlabel plot [-5:5] [-5:5] sin(x), cos(x)"
```

6.2.1. Further Reading

For more on the different settings:

1. http://people.duke.edu/~hpgavin/gnuplot.html
2. http://www.ibm.com/developerworks/aix/library/au-gnuplot/
3. http://site.ebrary.com.ep.fjernadgang.kb.dk/lib/royallibrary/reader.action?docID=10537861 (via REX)

7. Makefiles

`make` is a canonical build system for Unix-like development environments. Although primarily geared towards software development, `make` can also come in handy for data processing.

This tutorial is not a comprehensive introduction to `make`. We only cover what you might need for the course Machine Architecture (ARK) at DIKU, and beyond. Most crucially, `make` goes far beyond programming in C, we don't.

This tutorial assumes that you've already set up a Unix-like programming environment, know how to work with directories and files, and know your way around a text editor. We assume that you've already tried to get started with C, but you don't have to have completed it yet. Another great kick-starter is the G1 assignment.

This tutorial assumes that you already have GNU Binutils and GCC, the GNU Compiler Collection installed.

Go back to the aforementioned material if you are feeling uneasy.

7.1. Hello, make

It is about as easy to get started with `make` as it is to get started with C. Hopefully, it takes a bit less forbearance to become proficient in `make`, than it takes to become proficient in C.

To get started, create an empty directory and navigate to it:

Terminal.

```
$ mkdir canon
$ cd canon
```

Consider a simple program which fits on a file, which we can call `main.c`:

main.c.

```
int main() { return 42; }
```

With no further work, we can already use `make` to compile this program:

Terminal.

```
$ make main
cc     main.c   -o main
```

> **Note**
>
> `cc` is a relic of the 70's. Typically, this is just a symbolic link to your standard C compiler. Presumably, this is GCC. You can check what `cc` really is using the programs `which` and `file`:
>
> **Terminal.**
>
> ```
> $ which cc
> /usr/bin/cc
> $ file /usr/bin/cc
> /usr/bin/cc: symbolic link to gcc
> ```

If we instead called our file `root.c` we would have to compile it like this:

Terminal.

```
$ make root
cc     root.c   -o root
```

If we forget the program name, `make` will complain, not knowing what to do:

Terminal.

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

Similarly, if we misspell the program name, `make` will complain:

Terminal.

```
$ make homework
make: *** No rule to make target 'homework'.  Stop.
```

As long as we pass the command-line argument `main` to `make`, `make` can save us a couple keystrokes, and compile our `main.c` into an executable file called `main` on our behalf.

> **Note**
>
> Your directory structure should now look like this:
>
> **Terminal.**
>
> ```
> $ ls -lh
> -rwxr-xr-x [..] main
> -rw-r--r-- [..] main.c
> ```

We can try to run our program and check its exit code:

Terminal.

```
$ ./main || echo $?
42
```

A classical feature of `make`, is that it helps us avoid unnecessary recompilations.

Try to call `make main` again (without changing `main.c`):

Terminal.

```
$ make main
make: 'main' is up to date.
```

In this basic use, `make` takes `main.c` as a **prerequisite** for the **target** `main`. That is, `make` sets up a **dependency graph** which can be illustrated like this:

```
          +------+
          | main |
          +------+
              |
(depends on)  |
              v
         +--------+
         | main.c |
         +--------+
```

To implement this dependency graph, `make` will compare the **modification time** of `main.c` with that of `main`. If the prerequisite was modified after the target, `make` will run a **recipe** in attempt to bring the target "up to date" with the prerequisite. The default recipe in this case, is to compile the C file.

7.2. Hello, `Makefile`

The behaviour of calling `make` in a particular directory can be customized by creating a special file called `Makefile` in that directory. As a (de)motivating example, here is a `Makefile` that (in our case) will achieve the exact same effect as having no `Makefile` at all (except use the expected C compiler!):

Makefile.

```
main: main.c
	gcc main.c -o main
```

> **Note**
>
> Your directory structure should now look like this:
>
> **Terminal.**
>
> ```
> $ ls -lh
> -rwxr-xr-x [..] main
> -rw-r--r-- [..] main.c
> -rw-r--r-- [..] Makefile
> ```

A `Makefile` specifies a number of **rules**. A rule has a number of **targets** and **prerequisites**, as well as a **recipe** for brining the targets "up to date" with the prerequisites. A recipe is a sequence of **commands** which will be called in order, from top to bottom, each in their own shell.

The format of a `Makefile` rule goes as follows:

```
TARGETS `:` PREREQUISITES LINE-BREAK
TAB COMMAND LINE-BREAK
TAB COMMAND LINE-BREAK
TAB COMMAND LINE-BREAK
...
```

> **Important**
>
> Every line of a recipe must begin with a **tab character**.
>
> To quote the GNU `make` manual: "This is an obscurity that catches the unwary."

There is one benefit to our `Makefile` however: we no longer need to specify `main` as the command-line argument to `make`. It is now assumed by default:

Terminal.

```
$ make
make: 'main' is up to date.
$ rm main
$ make
gcc main.c -o main
```

7.3. Phony Targets

To make our `Makefile` a bit more useful, let's create a classical phony target — `clean`. `clean` will be "phony" in the sense that its recipe will not produce a file called `clean`. Instead, `clean` will clean up the mess our invocations of `make` have made above — in our case, just remove the `main` file.

A simple approach would've been to just add the `clean` target to our `Makefile`:

Makefile.

```
#BadMakefile

main: main.c
	gcc main.c -o main

clean:
	rm main
```

Unfortunately, if we were ever to place a file called `clean` into our directory, the `clean` target would always be considered up to date (why?). For instance, consider the following session at the terminal:

Terminal.

```
$ echo 42 > clean
$ make clean
make: 'clean' is up to date.
$ make
gcc main.c -o main
$ make clean
make: 'clean' is up to date.
$ ls -lh
-rw-r--r-- [..] clean
-rwxr-xr-x [..] main
-rw-r--r-- [..] main.c
-rw-r--r-- [..] Makefile
```

To avoid this problem (and make sure the recipe for `clean` is always run when we ask it to), we have to mark the `clean` target as `.PHONY`:

Makefile.

```
.PHONY: clean

main: main.c
	gcc main.c -o main

clean:
	rm main
```

Continuing the terminal session from before..

Terminal.

```
$ make clean
rm main
```

Note

If you followed our ill advice and created a file called `clean`, remove it so that we again have a directory structure like this:

Terminal.

```
$ ls -lh
-rwxr-xr-x [..] main
-rw-r--r-- [..] main.c
-rw-r--r-- [..] Makefile
```

If you spuriously try to play around, and try to `make clean` again, you'll get to see `make` fail:

Terminal.

```
$ make clean
rm main
rm: cannot remove ‘main’: No such file or directory
Makefile:7: recipe for target 'clean' failed
make: *** [clean] Error 1
```

The recipe is failing because we've already removed the file called `main`.`make` then tries to be helpful and tell us that it failed on line 7 of the `Makefile`, in the midst of the recipe for the `clean` target.

A recipe fails as soon as one of its commands (executed in order from top to bottom) yields a non-zero exit code.

This is what `rm` does for a nonexistent file. We can add a `-f` command-line argument to `rm` in our recipe to make `rm` ignore nonexistent files:

Makefile.

```
.PHONY: clean

main: main.c
	gcc main.c -o main

clean:
	rm -f main
```

Warning

`-f` should in general be used with caution — you might carelessly remove important files.

Now we can go on a command spree again!

Terminal.

```
$ make
gcc main.c -o main
$ make
make: 'main' is up to date.
$ make clean
rm -f main
$ make clean
rm -f main
```

```
$ ls -lh
-rw-r--r-- [..] main.c
-rw-r--r-- [..] Makefile
```

Mental exercise: Can you come up with other ways of solving the problem with the `clean` target?

7.4. A test target

Another useful phony target is a `test` target to perform the tests we have thus far been doing manually. This target has a `main` executable as a prerequisite, and the recipe should run the executable and check its exit code. `test` is a good example of a phony target with prerequisites.

One naïve approach could go as follows:

Makefile.

```
#BadMakefile

.PHONY: test clean

main: main.c
	gcc main.c -o main

test: main
	./main

clean:
	rm -f main
```

Let's try to `make test` and see what happens:

Terminal.

```
$ make test
./main
Makefile:7: recipe for target 'test' failed
make: *** [test] Error 42
```

So `./main` yields the expected exit code alright, but it is ill practice to designate a test error as a success.

A better `Makefile` could go as follows:

Makefile.

```
.PHONY: test clean

main: main.c
	gcc main.c -o main

test: main
	./main || echo $$?
```

```
clean:
	rm -f main
```

Important

We need to double the dollar sign in our `Makefile` as a dollar sign is otherwise used to start a variable reference in a `Makefile`. We will come back to variables in makefiles below.

We can try to `make test` to make sure that things work as expected:

Terminal.

```
$ make test
./main || echo $?
42
```

Note, the `test` target lists `main` as a prerequisite. So the dependency graph deduced by `make` can be illustrated as follows:

```
          +------+
          | test |
          +------+
              |
(depends on)  |
              v
          +------+
          | main |
          +------+
              |
(depends on)  |
              v
         +--------+
         | main.c |
         +--------+
```

To see how `make` implements this dependency graph, let's try to `make clean` and `make test`:

Terminal.

```
$ make clean
rm -f main
$ make test
gcc main.c -o main
./main || echo $?
42
```

Out of mere interest, let us try to introduce an error into our program and see how `make` will handle a compilation error:

Terminal.

```
$ make clean
$ echo "int main() { return x; }" > main.c
$ make test
gcc main.c -o main
main.c: In function ‘main’:
main.c:1:21: error: ‘x’ undeclared (first use in this function)
 int main() { return x; }
                     ^
main.c:1:21: note: each undeclared identifier is reported only once for each function it app
Makefile:4: recipe for target 'main' failed
make: *** [main] Error 1
```

Perhaps as you had already expected, `make` stopped processing the dependency graph as soon as it encountered an error in one of the recipes.

7.5. The Default Target

You might've noticed that `make` with no arguments still works despite the fact that there are now multiple targets in our `Makefile`:

Terminal.

```
$ make
make: 'main' is up to date.
$ make clean
rm -f main
$ make
gcc main.c -o main
```

`make` resolves target ambiguity in a very simple way — the top target is the default target, and in our `Makefile`, the top target is `main`.

This is not a good default target for two reasons:

1. Good software development practice tells us to test early and test often. `make` is quick to type and probably what we'll use as we write our program. It is perhaps more responsible to have `test` as our default target.

2. It is a common `Makefile` convention to name the default target `all`.

We can embrace both by adding a phony target `all` at the top of our `Makefile`, listing `test` as a prerequisite:

Makefile.

```
.PHONY: all test clean

all: test

main: main.c
	gcc main.c -o main
```

```
test: main
	./main || echo $$?

clean:
	rm -f main
```

Let's take the `Makefile` for a spin:

Terminal.

```
$ make clean
rm -f main
$ make
gcc main.c -o main
./main || echo $?
42
```

7.6. A More Complicated Program

Consider our stack calculator from the accompanying tutorial on Getting Started with C.

There, we had a stack data structure declared in a header file `stack.h`, and implemented in the C file `stack.c`. We compiled the implementation follows:

Terminal.

```
gcc -Werror -Wall -Wextra -pedantic -std=c11 -c stack.c
```

We then had a file `calc.c` which implemented the actual stack calculator using the stack implementation above. `calc.c` contained a `main` function. So we then compiled the program as follows:

Terminal.

```
gcc -Werror -Wall -Wextra -pedantic -std=c11 stack.o calc.c
```

Perhaps a natural `Makefile` for our stack calculator would then go as follows:

Makefile.

```
.PHONY: all test clean

all: test

test:
	./calc

calc: stack.o calc.c
	gcc -Werror -Wall -Wextra -pedantic -std=c11 stack.o calc.c

stack.o: stack.h stack.c
	gcc -Werror -Wall -Wextra -pedantic -std=c11 -c stack.c

clean:
```

```
        rm -f stack.o
        rm -f calc
```

The dependency graph deduced by `make` in this case, can be illustrated as follows:

```
          +-----+
          | all |
          +-----+
             |
(depends on) |
             v
          +------+
          | test |
          +------+
             |
(depends on) |
             v
          +------+
          | calc |
          +------+
             |
             +--------------+
(depends on) |              | (depends on)
             v              v
         +--------+    +---------+
         | calc.c |    | stack.o |
         +--------+    +---------+
                            |
                            +--------------+
               (depends on) |              | (depends on)
                            v              v
                       +---------+    +---------+
                       | stack.h |    | stack.c |
                       +---------+    +---------+
```

7.7. Variables

Our `Makefile` is starting to get a little cryptic and a little fragile. Good software development practice tells us not to repeat ourselves. We are repeating ourselves with all those compiler flags, and the compiler flags obscuring our recipes.

`Makefile` variables let us solve this in a straight-forward way. `Makefile` variables work a bit like simple C macros in that they are merely placeholders for text. Variables are typically declared at the top of the `Makefile`, named in ALL CAPS, with words occasionally separated by `_`.

For instance, here's a `Makefile` that resolves our problems above:

Makefile.

```
CC=gcc
CFLAGS=-Werror -Wall -Wextra -pedantic -std=c11

.PHONY: all test clean

all: test

test:
	./calc

calc: stack.o calc.c
	$(CC) $(CFLAGS) stack.o calc.c

stack.o: stack.h stack.c
	$(CC) $(CFLAGS) -c stack.c

clean:
	rm -f stack.o
	rm -f calc
```

> **Note**
>
> This `Makefile` also declares a variable for the compiler used. This is useful for the portability of our source code. Other machines may not have GCC installed, but use an equally adequate C compiler.

7.8. Conclusion

We can use `make` to make sure to build the elements of our software project in proper order, and put common software development tasks at our fingertips. We can use `Makefile` variables to keep our recipes consistent, to the point, and flexible.

We call `make` "canonical" because it is widely available in Unix-like programming environments. It is often used in large software projects, and is especially ubiquitous in the open-source and free software communities.

`make` is old. Originally developed in 1977, it has had many derivatives. GNU `make`, the version of `make` we've encouraged you to use here, is the standard implementation of `make` on most Linux and OS X systems. On Windows, the standard implementation is `nmake`, and comes as part of Visual Studio.

The rogue nature of `make` has also inspired the development of many alternative tools and companions. For instance, SCons, CMake, and Mk. Each come with their own benefits and setbacks.

A most notable critique of `make` is that it demands of you to manually manage your dependencies. Integrated Development Environments, such as Eclipse, Xcode, and Visual Studio, as well as many modern programming languages, such as Go and Rust, often come with their own build-automation tools, which automatically deduce dependencies from source-code. This results in unwarranted dependence on particular languages and tools.

In today's world, `make` is reserved for those who want to exert grand control over the build process, and projects which depend on a great variety of untamed languages and tools. `make` is widespread till this day.

7.9. Further Study

This tutorial is by no means a comprehensive introduction to `make`. Most notably, we've focused on programming in C, and forgotten to mention that `make` can be made to build dependencies in parallel, and that special, magic-looking makefile variables can be used to write terse recipes.

There's probably more that we've forgotten. If you want to know more, here are a couple good resources for further study:

1. Pierce Lopez. *Make*. `http://www.ploxiln.net/make.html`. 2015.

2. Free Software Foundation, Inc. GNU `make`. `http://www.gnu.org/software/make/manual/make.html`. 2014.

8. Conclusion

If you found Unix-like operating systems interesting, we recommend that you try to play around with either Antergos or Ubuntu in your spare time. These are Linux-based operating systems well-suited for beginners coming from other operating systems. If you're afraid to mess up your computer, you can try them out in a virtual machine (e.g., using VirtualBox).

If you would like to learn more, we can recommend the book A Practical Guide to Linux Commands, Editors, and Shell Programming, Second Edition (2010), by Mark G. Sobell. This is a good, but lengthy alternative to these lecture notes.