Compiling LaTeX with `make`

Računalniški praktikum

Philipp G. Haselwarter

5th November, 2019

Abstract

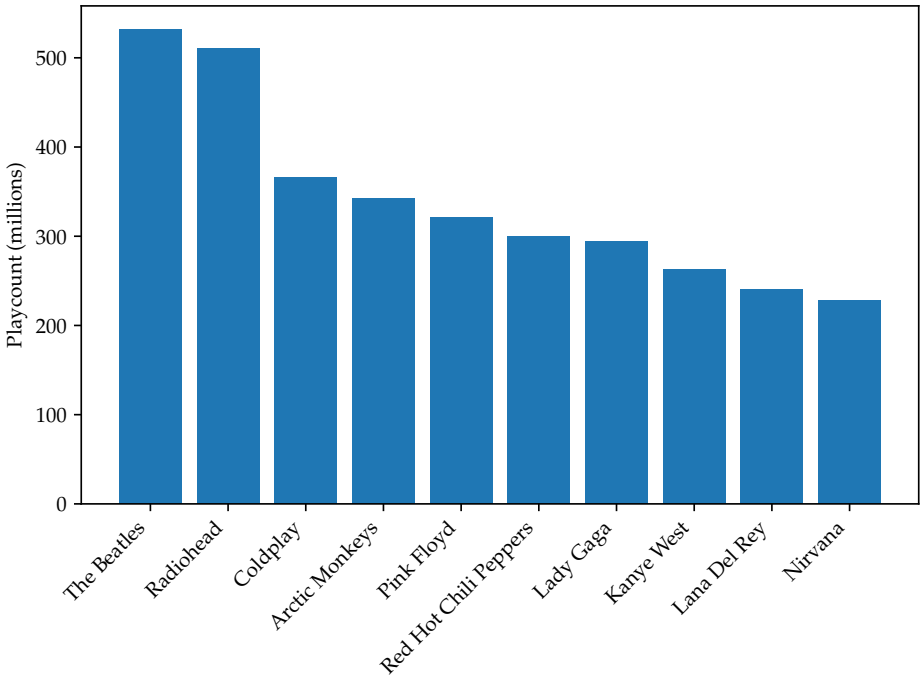
This is an example of a more complex LaTeX document. The complexity arises because other programs need to be executed before the pdf can be produced from the tex file. Furthermore, every time there is a change made to the files that are included in the tex file (as images, as data table), the pdf report should be reproduced.

Overview

We download a list of the 50 most popular musical artists from [last.fm, 2019a], extract the ten artists with the highest total playcount, produce the plot **??** from this data, and format the data as table in Appendix A. To automate this process, we will use the `make` program [GNU Make, 2019]. Furthermore, we give credit to the source of our data and the software we use in the bibliography. As we have seen, this in itself already makes the LaTeX compilation process more complicated.

Contents

Figure 1: Most popular artists from the last.fm API [last.fm, 2019b]. Retrived: 5th November, 2019



1 Files required to produce this document

- `buildsystem.tex` contains the overall structure and most of the LaTeX code
- `literature.bib` contains the literature references we cite
- `playcount.pdf` contains a barplot of the play counts of different artists
- `top_artists.csv` contains the data the plot was drawn from
- `date_downloaded.tex` defines a macro that expands to the date when the data was downloaded

2 Steps to produce this document

The steps required to compile this document are as follows:

1. run `./download_charts.sh` to produce `top_artists.json` and `date_downloaded.tex`
2. run `./parse_charts.py playcount.pdf` to produce `playcount.pdf`
3. run `./parse_charts.py top_artists.csv` to produce `top_artists.csv`
4. run `pdflatex buildsystem.tex` to produce `buildsystem.aux`
5. run `bibtex buildsystem` to produce `buildsystem.bbl`
6. run `pdflatex buildsystem.tex` twice to fix unresolved references and produce `buildsystem.pdf`

Observe that steps 2 and 3 are independent from each other and can be run in any order.

Exercise 1. Open a cygwin terminal, `cd` into your praktikum folder for today, and run these steps to produce `buildsystem.pdf`.

Exercise 2. Open `buildsystem.tex` and fix all the tasks masked with `TODO`.

Exercise 3. On a sheet of paper, draw the graph of dependencies between the files. Each node in your graph should correspond to a file, and it should have edges connecting to the files it depends upon. Start with `buildsystem.pdf`. Does any file depend upon itself? What is a sufficient condition for us to be able to compile a dependency graph? How could we relax that condition?

3 Automating document creation: a simple `makefile`

To make compilation more manageable, we would like to leave the steps outlined in section 2 to a program instead of performing them manually. Furthermore, and this is important, we want to perform only the steps that are necessary. For example, if we already downloaded the charts, we should not re-perform the download request unless we really want to. Too much stress on the server might result in getting us blocked. The operation would also unnecessarily fail if we already have the data but we're offline. If a node has two or more incoming edges, compilation should still be performed just once.

3.1 The make program

The `make` program takes a textual description of the dependency graph together with a *rule* to produce each node from its dependencies or *pre-*

requisites, and executes the rules to compile the *target*. This description of a compilation process is very general, and can be applied to languages other than latex. The `make` utility is freely available and runs on all operating systems.

To get an overview of the arguments that `make` can be invoked with, run `make --help`. The arguments we will be using today are listed in Figure 2.

argument	short form	description
`--help`	`-h`	describe options & arguments
`--makefile FILE`	`-f FILE`	read rules from *FILE*
`--no-builtin-rules`	`-r`	only use rules from current Makefile
`--debug`	`-d`	explain rule selection during compilation
`--just-print`	`-n`	only print selected rules, no execution
`--jobs N`	`-j N`	run up to N independent rules in parallel

Figure 2: `make` commandline arguments

Exercise 4. Create an empty file called `01-simple.mk` and run

```
make -f 01-simple.mk -r -d -n
```

3.2 Syntax of Makefiles

Text editors recognise files with the extension `.mk` as makefiles. The syntax for rules is as follows:

```
target : prerequisite-1 prerequisite-2 [...]
\tab  recipe
```

where `\tab` should be an actual tab character, not followed by spaces. `target` and the list of `prerequisite-N` are file names. `recipe` is a shell command that should produce `target`, assuming that all prerequisites have been taken care of. For example, to express that `top_artists.csv` depends on `parse_charts.py` and `top_artists.json`, one would write the following rule:

```
top_artists.csv : parse_charts.py top_artists.json
\tab ./parse_charts.py top_artists.csv
```

If a target requires the execution of several commands, the commands should be listed under the target, each indented with a tab, one command per line. Comments can be started with a `#` character. Long lines may be broken by putting a `\` at the end of the line and continuing on the next line.

Exercise 5. Add a rule for each node in the dependency graph to `01-simple.mk`. Start with those nodes that are leafs in the graph, i.e. that do not depend on any further files. While you add rules, run `make` with the correct arguments to see what actions would get performed. You can specify the target you want to build by adding it as a last argument to the make invocation. By default, the first target listed in the makefile will be selected. Once you're done, delete all the intermediate files with

```
rm *.pdf *.log *.blg *.bbl *.aux
```

and build the pdf again.

Exercise 6. Add two more rules to `01-simple.mk`: `clean` should delete all files ending in `.log .aux .blg .bbl`, `extraclean` should run `clean` and additionally delete the following files:

- `buildsystem.pdf`
- `top_artists.json`
- `date_downloaded.tex`
- `playcount.pdf`
- `top_artists.csv`

4 Writing a sophisticated `makefile`

Now that we have a basic `makefile` working, let's explore some more features of `make`. Create a new file `02-fancy.mk` based on `01-simple.mk`.

Exercise 7 : phony targets. The last two rules that we added, `clean` and `extraclean`, do not actually create any files. However, if through some other means a file called `clean` was actually put into the folder, the `make` would not execute the `clean` rule anymore. In `make` terminology, such rules are *phony*. Here is the Witionary entry for *phony (noun)*:

> A person who assumes an identity or quality other than their own.
> Example: "He claims to be a doctor, but he's nothing but a fast-talking phony."

We can declare a targets as phony by listing them as prerequisites for a special `make` target called `.PHONY`.

Exercise 8 : failing. Make sure that the file `buildsystem.pdf` exists. Run `make` on the `clean` target. Then run `extraclean`. The file `buildsystem.pdf` did not get deleted. Why did `make` fail? Read the

manual for the `rm` command by running `man rm` (use `page up/down` to navigate, press `q` to quit), and find an option that modifies the behaviour of `rm` so that `clean` does not fail in this situation.

Exercise 9 : multiple targets. The files `top_artists.json` and `date_downloaded.tex` are produced by identical rules. In fact, a rule can have more than one target. List the targets in a single rule. Targets are simply separated by a space, like prerequisites.

Exercise 10 : defining variables. The file names `playcount.pdf` and `top_artists.csv` occur together several times in the makefile. We can define a variable as follows:

```
SCRIPT_TARGETS = playcount.pdf top_artists.csv
```

and then reference it as `$(SCRIPT_TARGETS)`. Replace occurances of the two file names by the variable. Then repeat the process for a variable `DOWNLOAD_TARGETS` for the files that `download_charts.sh` produces.

Exercise 11 : built-in variables. The files in `$(DOWNLOAD_TARGETS)` are created by the exact same command. The situation is very similar for `$(SCRIPT_TARGETS`, except that in each case the argument to `./parse_charts.py` has to be replaced by the name of the target. We can access the name of the current target when a recipe is executed under the variable `$@`. Modify the rules for `playcount.pdf` and `top_artists.csv` so that the recipe does not explicitly mention the name of the target anymore but uses `$@` instead. Then combine the two rules into a multi-target rule.

Exercise 12 : more variables. It is good practise to define a variable for the name of the program that compiles the main target, in our case `pdflatex`, and a variable for the arguments to that program. This way, if the arguments change, they can be conveniently edited at the beginning of the file. Define a compiler variable for `pdflatex` and set the argument variable to `-synctex=1`. Then use these variables in the rest of the makefile instead.

5 Modern problems need modern solutions

The `make` utility is a very useful general purpose tool. However, LaTeX has some peculiarities that require special solutions. In particular, after `bibtex` is run, we have to invoke `pdflatex` twice more. In fact, there are some scenarios where even a third invocation is required. This makes it difficult to write robust makefiles. A solution would be to parse the log files and check whether `pdflatex` or some other package requires another pass.

The `latexmk` utility does exactly that. For simple projects that require no external programs to be run besides `bibtex`, it can completely replace a makefile.

Create a new file `03-latexmk.mk` based on `02-fancy.mk`.

Exercise 13. Replace the two consecutive invocations of `pdflatex` in the rule for `buildsystem.pdf` by a single call to `latexmk`. Be sure to add the `-pdf` option to the variable that defines the arguments for the LaTeX compiler. As `latexmk` how to produce a `.aux` file, the rule for `buildsystem.aux` will now be obsolete.

Exercise 14. What does `latexmk -c` do? What does `latexmk -C` do? Modify your new makefile to use this functionality.

6 Exploring the `last.fm` API

Exercise 15 : bonus. Read the documentation of the `last.fm` application programming interface [last.fm, 2019b]. The `download_charts.sh` script uses the `chart.getTopArtists` method of the API. Create a script `download_top_tracks.sh` that uses `chart.getTopTracks` method instead, and save the result to `top_tracks.json`.

Exercise 16 : bonus. Modify `parse_charts.py` so that it can also parse the top tracks and output a plot like the one for artists, as well as a corresponding csv file.

References

[GNU Make, 2019] GNU Make (2019). `https://www.gnu.org/software/make/`.

[last.fm, 2019a] last.fm (2019a). last.fm. `https://www.last.fm/`.

[last.fm, 2019b] last.fm (2019b). Last.fm API. `https://www.last.fm/api/`. Accessed: 27th October, 2019.

A Data table

name	playcount	listeners
The Beatles	531600358	3752838
Radiohead	510896158	4795615
Coldplay	366006642	5449800
Arctic Monkeys	342025189	3567197
Pink Floyd	321023467	3142500
Red Hot Chili Peppers	299691826	4680872
Lady Gaga	294485383	3895390
Kanye West	262418627	4485781
Lana Del Rey	240634638	1980539
Nirvana	227599240	4337030
The Killers	214034320	4486531
Daft Punk	213429168	3839953
Taylor Swift	206962845	2310662
Rihanna	205992931	4640832
Eminem	205810156	4586830
Queen	200070488	4104488
David Bowie	199327097	3417771
Green Day	192925394	3802939
Gorillaz	174696229	3637616
Beyoncé	167763731	3649746
The Cure	161545022	2991075
The Rolling Stones	158155671	3857678
Katy Perry	154416804	3819887
Drake	150834304	3461879
Michael Jackson	134362237	3612905