Metaprogramming

What do we mean by “metaprogramming”? Well, it was the best collective term we could come up with for the set of things that are more about *process* than they are about writing code or working more efficiently. In this lecture, we will look at systems for building and testing your code, and for managing dependencies. These may seem like they are of limited importance in your day-to-day as a student, but the moment you interact with a larger code base through an internship or once you enter the “real world”, you will see this everywhere. We should note that “metaprogramming” can also mean “programs that operate on programs”, whereas that is not quite the definition we are using for the purposes of this lecture.

Build systems

If you write a paper in LaTeX, what are the commands you need to run to produce your paper? What about the ones used to run your benchmarks, plot them, and then insert that plot into your paper? Or to compile the code provided in the class you’re taking and then running the tests?

For most projects, whether they contain code or not, there is a “build process”. Some sequence of operations you need to do to go from your inputs to your outputs. Often, that process might have many steps, and many branches. Run this to generate this plot, that to generate those results, and something else to produce the final paper. As with so many of the things we have seen in this class, you are not the first to encounter this annoyance, and luckily there exist many tools to help you!

These are usually called “build systems”, and there are *many* of them. Which one you use depends on the task at hand, your language of preference, and the size of the project. At their core, they are all very similar though. You define a number of *dependencies*, a number of *targets*, and *rules* for going from one to the other. You tell the build system that you want a particular target, and its job is to find all the transitive dependencies of that target, and then apply the rules to produce intermediate targets all the way until the final target has been produced. Ideally, the build system does this without unnecessarily executing rules for targets whose dependencies haven’t changed and where the result is available from a previous build.

`make` is one of the most common build systems out there, and you will usually find it installed on pretty much any UNIX-based computer. It has its warts, but works quite well for simple-to-moderate projects. When you run `make`, it consults a file called `Makefile` in the current directory. All the targets, their dependencies, and the rules are defined in that file. Let’s take a look at one:

```
paper.pdf: paper.tex plot-data.png
        pdflatex paper.tex

plot-%.png: %.dat plot.py
        ./plot.py -i $*.dat -o $@
```

Each directive in this file is a rule for how to produce the left-hand side using the right-hand side. Or, phrased differently, the things named on the right-hand side are dependencies, and the left-hand side is the target. The indented block is a sequence of programs to produce the target from those dependencies. In `make`, the first directive also defines the default goal. If you run `make` with no arguments, this is the target it will build. Alternatively, you can run something like `make plot-data.png`, and it will build that target instead.

The `%` in a rule is a "pattern", and will match the same string on the left and on the right. For example, if the target `plot-foo.png` is requested, `make` will look for the dependencies `foo.dat` and `plot.py`. Now let's look at what happens if we run `make` with an empty source directory.

```
$ make
make: *** No rule to make target 'paper.tex', needed by 'paper.pdf'.  Stop.
```

`make` is helpfully telling us that in order to build `paper.pdf`, it needs `paper.tex`, and it has no rule telling it how to make that file. Let's try making it!

```
$ touch paper.tex
$ make
make: *** No rule to make target 'plot-data.png', needed by 'paper.pdf'.  St
```

Hmm, interesting, there *is* a rule to make `plot-data.png`, but it is a pattern rule. Since the source files do not exist (`data.dat`), `make` simply states that it cannot make that file. Let's try creating all the files:

```
$ cat paper.tex
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\includegraphics[scale=0.65]{plot-data.png}
\end{document}
$ cat plot.py
#!/usr/bin/env python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-i', type=argparse.FileType('r'))
parser.add_argument('-o')
args = parser.parse_args()

data = np.loadtxt(args.i)
plt.plot(data[:, 0], data[:, 1])
plt.savefig(args.o)
$ cat data.dat
1 1
2 2
3 3
4 4
5 8
```

Now what happens if we run `make`?

```
$ make
./plot.py -i data.dat -o plot-data.png
pdflatex paper.tex
... lots of output ...
```

And look, it made a PDF for us! What if we run `make` again?

```
$ make
make: 'paper.pdf' is up to date.
```

It didn't do anything! Why not? Well, because it didn't need to. It checked that all of the previously-built targets were still up to date with respect to their listed dependencies. We can test this by modifying `paper.tex` and then re-running `make`:

```
$ vim paper.tex
$ make
pdflatex paper.tex
...
```

Notice that `make` did *not* re-run `plot.py` because that was not necessary; none of `plot-data.png`'s dependencies changed!

Dependency management

At a more macro level, your software projects are likely to have dependencies that are themselves projects. You might depend on installed programs (like `python`), system packages (like `openssl`), or libraries within your programming language (like `matplotlib`). These days, most dependencies will be available through a *repository* that hosts a large number of such dependencies in a single place, and provides a convenient mechanism for installing them. Some examples include the Ubuntu package repositories for Ubuntu system packages, which you access through the `apt` tool, RubyGems for Ruby libraries, PyPi for Python libraries, or the Arch User Repository for Arch Linux user-contributed packages.

Since the exact mechanisms for interacting with these repositories vary a lot from repository to repository and from tool to tool, we won't go too much into the details of any specific one in this lecture. What we *will* cover is some of the common terminology they all use. The first among these is *versioning*. Most projects that other projects depend on issue a *version number* with every release. Usually something like 8.1.3 or 64.1.20192004. They are often, but not always, numerical. Version numbers serve many purposes, and one of the most important of them is to ensure that software keeps working. Imagine, for example, that I release a new version of my library where I have renamed a particular function. If someone tried to build some software that depends on my library after I release that update, the build might fail because it calls a function that no longer exists! Versioning attempts to solve this problem by letting a project say that it depends on a particular version, or range of versions, of some other project. That way, even if the underlying library changes, dependent software continues building by using an older version of my library.

That also isn't ideal though! What if I issue a security update which does *not* change the public interface of my library (its "API"), and which any project that depended on the old version should immediately start using? This is where the different groups of numbers in a version come in. The exact meaning of each one varies between projects, but one relatively common standard is *semantic versioning*. With semantic versioning, every version number is of the form: major.minor.patch. The rules are:

– If a new release does not change the API, increase the patch version.
– If you *add* to your API in a backwards-compatible way, increase the minor version.

– If you change the API in a non-backwards-compatible way, increase the major version.

This already provides some major advantages. Now, if my project depends on your project, it *should* be safe to use the latest release with the same major version as the one I built against when I developed it, as long as its minor version is at least what it was back then. In other words, if I depend on your library at version `1.3.7`, then it *should* be fine to build it with `1.3.8`, `1.6.1`, or even `1.3.0`. Version `2.2.4` would probably not be okay, because the major version was increased. We can see an example of semantic versioning in Python's version numbers. Many of you are probably aware that Python 2 and Python 3 code do not mix very well, which is why that was a *major* version bump. Similarly, code written for Python 3.5 might run fine on Python 3.7, but possibly not on 3.4.

When working with dependency management systems, you may also come across the notion of *lock files*. A lock file is simply a file that lists the exact version you are *currently* depending on of each dependency. Usually, you need to explicitly run an update program to upgrade to newer versions of your dependencies. There are many reasons for this, such as avoiding unnecessary recompiles, having reproducible builds, or not automatically updating to the latest version (which may be broken). An extreme version of this kind of dependency locking is *vendoring*, which is where you copy all the code of your dependencies into your own project. That gives you total control over any changes to it, and lets you introduce your own changes to it, but also means you have to explicitly pull in any updates from the upstream maintainers over time.

Continuous integration systems

As you work on larger and larger projects, you'll find that there are often additional tasks you have to do whenever you make a change to it. You might have to upload a new version of the documentation, upload a compiled version somewhere, release the code to pypi, run your test suite, and all sort of other things. Maybe every time someone sends you a pull request on GitHub, you want their code to be style checked and you want some benchmarks to run? When these kinds of needs arise, it's time to take a look at continuous integration.

Continuous integration, or CI, is an umbrella term for "stuff that runs whenever your code changes", and there are many companies out there that provide various types of CI, often for free for open-source projects. Some of the big ones are Travis CI, Azure Pipelines, and GitHub Actions. They all work in roughly the same way: you add a file to your repository that describes what should happen when various things happen to that repository. By far the most common one is a rule like "when someone pushes code, run the test suite". When the event triggers, the CI provider spins up a virtual machines (or more), runs the commands in your "recipe", and then usually notes down the results somewhere. You might set it up so that you are notified if the test suite stops passing, or so that a little badge appears on your repository as long as the tests pass.

As an example of a CI system, the class website is set up using GitHub Pages. Pages is a CI action that runs the Jekyll blog software on every push to `master` and makes the built site available on a particular GitHub domain. This makes it trivial for us to update the website! We just make our changes locally, commit them with git, and then push. CI takes care of the rest.

A brief aside on testing

Most large software projects come with a "test suite". You may already be familiar with the general concept of testing, but we thought we'd quickly mention some approaches to testing and testing terminology that you may encounter in the wild:

- Test suite: a collective term for all the tests
- Unit test: a "micro-test" that tests a specific feature in isolation
- Integration test: a "macro-test" that runs a larger part of the system to check that different feature or components work *together*.
- Regression test: a test that implements a particular pattern that *previously* caused a bug to ensure that the bug does not resurface.
- Mocking: to replace a function, module, or type with a fake implementation to avoid testing unrelated functionality. For example, you might "mock the network" or "mock the disk".

Exercises

1. Most makefiles provide a target called `clean`. This isn't intended to produce a file called `clean`, but instead to clean up any files that can be re-built by make. Think of it as a way to "undo" all of the build steps. Implement a `clean` target for the `paper.pdf` `Makefile` above. You will have to make the target phony. You may find the `git ls-files` subcommand useful. A number of other very common make targets are listed here.
2. Take a look at the various ways to specify version requirements for dependencies in Rust's build system. Most package repositories support similar syntax. For each one (caret, tilde, wildcard, comparison, and multiple), try to come up with a use-case in which that particular kind of requirement makes sense.
3. Git can act as a simple CI system all by itself. In `.git/hooks` inside any git repository, you will find (currently inactive) files that are run as scripts when a particular action happens. Write a `pre-commit` hook that runs `make paper.pdf` and refuses the commit if the `make` command fails. This should prevent any commit from having an unbuildable version of the paper.
4. Set up a simple auto-published page using GitHub Pages. Add a GitHub Action to the repository to run `shellcheck` on any shell files in that repository (here is one way to do it). Check that it works!
5. Build your own GitHub action to run `proselint` or `write-good` on all the `.md` files in the repository. Enable it in your repository, and check that it works by filing a pull request with a typo in it.