Metaprogramming

Metaprogramming 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.

Build Systems

Build 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. These are usually called **build system**, and there are many of them.

`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 workd quite wiell for simple-to-moderate projects. When you run `make`, it consults a file called `Makefile` in the current directory. All the targets, their dependencied, and the rules are difined in the file. Let's take a look at once:

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

paper-%.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`.

Dependency management

With semantic versioning, every version number is of the form: `major.minor.patch`. The rules are:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards compatible manner, and

- PATCH version when you make backwards compatible bug fixes.

Continuous integration systems

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. 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.

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.