Metaprogramming

Table of Contents

Build systems

Build systems run a build process. Combination of - targets - dependencies - rules 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. - typically won't build if already up to date

`make`

- available on Windows, Linux, MacOS
- suitable for small-to-medium projects
- running `make` consults `Makefile` in current directory

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

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

- each directive is a rule for how to produce LHS (target) from RHS (dependencies)
- indented block is sequence of programs to produce the target
- first directive defines default goal; running `make` with no arguments will build this target
- alternatively `make plot-data.png` will build that target instead
- `%` will match the same string on LHS and RHS
- must use `TAB` not spaces

Other build systems

- `cmake`: for C projects
- `maven`: for Java projects

Dependency Management

Semantic versioning

`major.minor.patch` Release increments number of: - *patch*: API hasn't changed - *minor*: API has new, backwards-compatible changes - *major*: API has backwards-incompatible change

e.g. python 2.7 vs 3.6 Semantic versioning allows you to specify dependencies via a major version and project should still work with later versions

Lock files

- list of dependencies and current version you are using
- good for
  - reproducible builds
  - avoiding unnecessary recompiles
  - not automatically updating to latest version
- vendoring: copy full project dependency into your project
  - full control over any changes, but means you need to pull in updates from maintainers manually

Continuous Integration

- cloud build system
- execute event-triggered actions e.g. linting, build project, load to PyPI, run test suite
- add *recipe* to your repository with events and actions
- e.g. Travis CI, Azure Pipelines, Github Actions
  - (Dependabot)[https://dependabot.com/]

Tests

- *test suite*: collection of tests usually run as a unit, composed of
  - *unit tests*: small, self-contained, testing single particular feature
  - *integration tests*: test interaction between sub-systems
  - *mocking*: replace parts of system with dummy version that behaves in controlled way
    * e.g. a tool to do file copying over ssh: mock the network so you don't need a network to run test suite

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.

   - Solution:

   ```
   PHONY: clean
   clean:
   	git ls-files --other | xargs rm
   ```

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.

   - Solution:
     - `^`: allows patch and minor updates from specified version, disallows major updates
       * e.g. `^1.2.3 := >=1.2.3, < 2.0.0`
       * use case: probably typical for dependencies that aren't critical or heavily used - you will get latest patches and minor versions which should be compatible. Will require deliberate upgrade to next major version to ensure functionality isn't broken.
     - `~`: minimal version with some ability to update. if major+minor+patch specified only patch level changes are allowed. If major+minor specified only minor changes allowed:

    * e.g. `~1.2 := >=1.2.0, <1.3.0`
    * use case: you may use `~1.2` when dependency is critical to project, while ensuring that patches are installed
  - `*`: allows any version in position of wildcard
    * use case: if dependency is mature and fairly stable, and not used heavily in your project, you may specify a major version and wildcard the minor version to get the latest compatible version
  - `<, <=, >, >=, =`: manual specification of version range or exact version
    * use case: if dependency is unstable and critical you may want to lock a particular version and manually update patches to ensure it functions as expected
  - multiple requirements e.g. `>= 1.2, < 1.5`
    * use case: tests may pass on a range of dependencies but fail after a certain version, so you may limit requirements to a specific range

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 un-buildable version of the paper.

   `.git/hooks/pre-commit`: “‘ #!/bin/sh # # Pre-commit script to prevent commit if the make fails

   # Redirect output to stderr. exec 1>&2

   if make

   then echo “Make successful” else cat « EOF Error: could not make pdf EOF exit 1 fi
   “‘

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.