little languages

lecture 28:

make and Makefiles

VM day today!
Pull 590-material from upstream.
Run: sudo apt install ghostscript

Today's Goal

1. What is a build system?

2. Case Study: make The Original Build System

3. Reflect: Imperative vs. Declarative Approaches
   - Shell Script vs. Makefile

Cargo Example

- In your most recent problem set, try running the following command twice:
  - cargo build --verbose
- Count the number of times you see "Running" and a command
- Now, clean the project:
  - cargo clean --verbose
- Then, try building again:
  - cargo build --verbose

Build Systems

- Machine code program files are complex digital artifacts to produce
  - Many tools are required to compile high level programs into machine code
- Tools in a compilation process may include:
  - linters to check and fix deviations from a style guide
  - running of test harnesses to verify lack of regressions
  - optimization of assets (images, language resource files, and so on)
  - compilation of source code to an intermediate representation
  - compilation of intermediate representations to machine code
- Carrying out each step manually is tedious and error prone.
- During development only small parts of a program change.
  - Why repeat the whole process from scratch when most steps have same results?
  - Downside for using only a shell script to automate your software build:
    - At worst: naive. Repeats the whole process from scratch.
    - At best: complex and fragile. Keeping track of all dependencies and taking min actions on change is tough.

Enter: `make` and Makefiles (1976 - Bell Labs)

"Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected).

As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make **that weekend**.

Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff."

— *Stuart Feldman*

The `make` Build System's Big Idea

- Early steps in a build will run a commands taking ***source*** files to produce ***target*** files.
- Later steps' commands use ***target*** files as ***source*** files and produce more ***target*** files.

- In a `Makefile`, you specify each step's:

  1. Prerequisite "Source" files
  2. Recipe of Command(s) to process those files
  3. The "Target" file produced by the recipe

- **`make`** reads the Makefile and then figures out which target files are missing or outdated and run *only* the commands needed to build exactly those targets.

- **`make`** was designed for software projects but works much more generically
  - This is evidence of a *good abstraction.* Does it generalize beyond intent?

Makefile - Rule Specification

```
<target-file>: <source-file>*
[tab-character]<recipe-to-produce-target-from-source>*
```

<u>Example:</u>

```
ch1.pdf: ch1.md
	pandoc -o ch1.pdf ch1.md
```

Makefiles with multiple final targets

- By default, make treats the first rule of a Makefile as the "default goal"
  - This rule is considered the final target of the build.
- To run multiple steps that produce multiple targets, it's common to have an "all" default goal with sub-targets as prereqs and no recipe.
- For example:

```
all: ch1.pdf ch2.pdf
```

Running **make** with specific goals

- With the **make** command you can specify a goal (name of target)
- For example:
```
$ make ch1.pdf
$ make ch2.pdf
```
- This causes **make** to focus on a subtarget
- This feature is commonly used to add build tasks to a project *outside* of the compilation process. For example: cleaning generated files up.

Adding a **`clean`** goal

- Add a rule *at the end of the Makefile* to delete the produced PDF files:

```
clean:
    rm *.pdf
```

- This rule can be run as a goal:

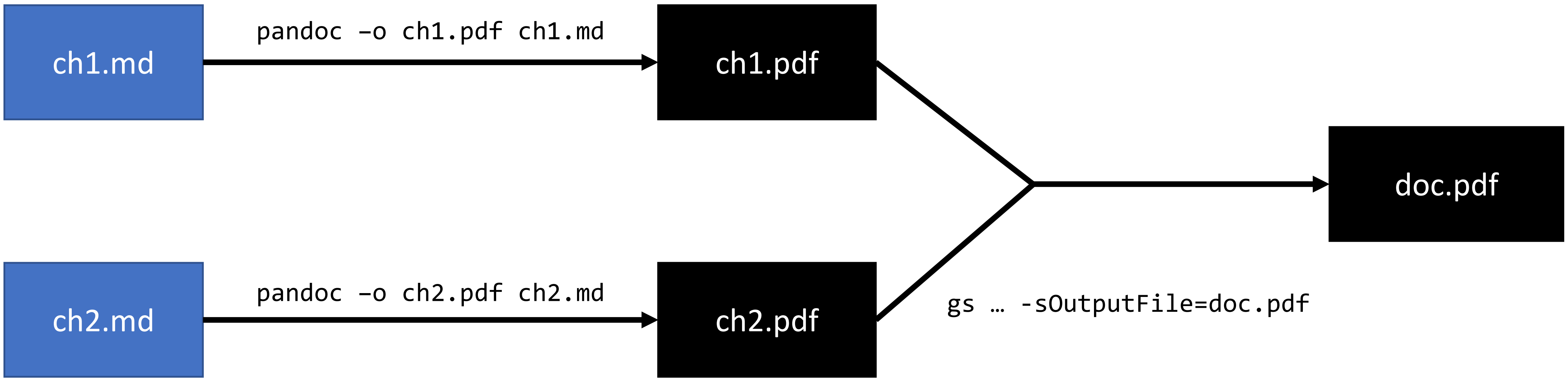
```
$ make clean
```

Targets that build on one another

- Let's use ghostscript to merge multiple PDFs into a single PDF
  - ghostscript is "An interpreter for the PostScript language and for PDF."
  - Not installed by default: `sudo apt install ghostscript`
- Replace the 'all' rule with:

```
doc.pdf: ch1.pdf ch2.pdf
	gs -q -dNOPAUSE -dBATCH -sDEVICE=pdfwrite \
	   -sOutputFile=doc.pdf \
	   ch1.pdf ch2.pdf
```

Dependency Visualization



- Notice your Makefile describes the structure of a directed acyclic graph
  - The nodes of the graph are files and the edges are build steps
- If a node is target determined to be missing, make can backtrack to the missing prerequisites and execute the commands of each edge in order.
- This is an example application of partial ordering and topological sort!

Automatic Variables

- It's common you want to reference your target file or prerequisite file(s) as part of the recipe.

- Automatic variables are available:
  - $@ - The filename of the target of the rule.

  - $^ - The names of all the prerequisite source files with spaces between them.

  - $? - The names of all prerequisite source files that are newer than the target.

- And many more...

`make` is a much deeper subject than this tutorial

- As a 40-year old tool it has accumulated many capabilities
- Many features try to avoid redundancy and verbosity of the `Makefile`
  - The downside is this leads to cryptic, non-obvious Makefiles
- Special features to use `make` for building specific kinds of projects
  - i.e. C projects or archives
- Modern build systems like CMake will *generate a Makefile* specific to the system the project is being built on.
  - Eases portability between operating systems and versions.
- The documentation for **`make`** is generally very good:
  - https://www.gnu.org/software/make/manual/make.html

Case Study: Compiling C Projects

- Open example for lec28 / c /

- Notice three files:
  - main.c - Entry point of program, includes helper functions.
  - helpers.h - Header file with helper function declarations.
  - helpers.c - Helper function definitions.

- Let's explore the Makefile of this project which looks a little more like a Makefile you'll commonly see in the real world.

```makefile
# Common Variables

# The shell recipes should be interpretted in.
SHELL = /bin/sh

# The C compiler to use.
CC = gcc
# Flags for the C compiler.
CFLAGS = -I. -g

# A list of the object files of our program
objects = main.o helpers.o

# The default goal is a `factorial` program. This links object files.
factorial: $(objects)
    $(CC) $(CFLAGS) -o $@ $^

# Each c file is composed into an object file compiled from a source file.
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

# PHONY rules are ones that do not produce target files.
.PHONY: clean
clean:
    rm -f factorial $(objects)
```

Many build tools are `make`-inspired

- Nothing stops you from using make for any project, but many ecosystems revolve around tooling custom suited for their environment.
- C/C++ – make, Cmake, bazel
- Rust – cargo
- Java – Ant, Maven, Gradle, bazel
- Node.js / JavaScript / TypeScript – npm, webpack, gulp, grunt
- Python – Scons, Waf

Imperative vs. Declarative Languages

- **Imperative** languages describe *how* a task should be accomplished
  - Most general-purpose languages (GPLs) are imperative by default
  - Often concerned with how to effectively mutate state and actual algorithms
  - You are writing the algorithm(s).
- **Declarative** languages describe *what* a task should accomplish
  - Most Little Languages are declarative by design and limitation
  - The *how* is left to the implementor of the Little Language
- This is a false dichotomy in that you can take a declarative approach in general-purpose programming languages *by using good abstractions.*
  - Functional example: higher-order functions like filter, map, and reduce.
  - Java example: sorting methods.
  - Rust example: macros.
- The existence of declarative solutions suggests good abstractions to a generalized problem.
  - Declarative solutions can be improved for free by improving the underlying system!