Makefiles

yosef?

Problem: automating builds

Most software you write at CMU is only a few files and only needs to work on the Andrew machines. But what about when:

- The software is dozens or hundreds or thousands of files
- The software depends on third-party libraries or frameworks
- The software needs to run on multiple operating systems or servers

SubProblem: The software is very big

It’s not feasible to type all of the commands by hand. You could write a bash script to compile it, but…

- … you don’t want to recompile every file every time
- … Bash is hard
- … do we really need arbitrary loops and conditionals to compile code?

SubProblem: Third-party dependencies

- Where are they installed?
- Can you assume that they’re installed?

SubProblem: The software needs to be portable

Rather than just the Andrew machines (Ubuntu Linux), you'd like to be able to compile your code on your laptop (???), or AWS (Amazon Linux), or Solaris for some reason, but …

- … all of the paths are different! Is it /usr or /usr/bin or /opt or what?
- … Bash isn't even installed???
- … writing portable code for all Unixes is hard!!!

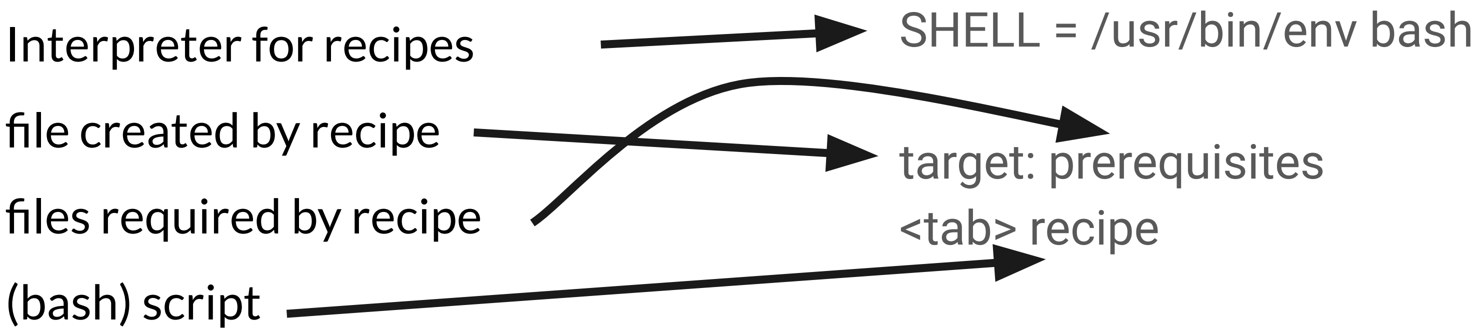
Makefile syntax

```
SHELL = /usr/bin/env bash

target: prerequisite1 rerequisite2
<tab> recipe
<tab> recipe continued
```

The tab character is important!

Makefile Semantics



```
SHELL = /usr/bin/env bash

target: prerequisites
<tab> recipe
```

Makefile Semantics

Makefiles essentially describe a Directed Acyclic Graph (DAG) of dependencies between how to create targets from prerequisites using recipes.

Make inspects the modification time of each file and rebuilds a target if its modification time is earlier than that of one of its prerequisites.

The .PHONY Tag

Make is fundamentally concerned with creating files. If you just want to run a recipe always regardless of whether the target file exists, you need to create a .PHONY rule.

The “make clean” rule should always be .PHONY

Automatic Variables

$@	Target file name
$<	First prerequisite name
$^	All prerequisite names

Implicit Rules

For convenience, Make includes a lot of pre-defined rules based on file extension. Example:

```
%.o: %.c
	gcc $< -o $@
```

Compiles all C files

The make command

“make <target>” will run the recipes for <target> and all its dependencies in an efficient way. “make” will default to the first <target> listed in Makefile.

-f <file>: use the following file instead of Makefile
-j <job_num>: run Make in parallel using this many cores

Different Interpreters

Make is almost always used with either bash or sh. It’s technically possible to write recipes using any interpreter using the SHELL variable (we set it to bash for consistency).

Alternative Build Systems

- Scons
  - Written in Python. Fairly slow. Uses cached file hashes instead of modification time.
- Ninja
  - Very fast. Basically just the DAG part of Make without any bells and whistles.

Build Configuration Systems

These solve the problem of compatibility across Unixes and machines.

- GNU Autotools
- CMake

Package Managers

These solve the problem of third-party dependencies. They also typically make use of the build system and build configuration systems to simplify package creation.