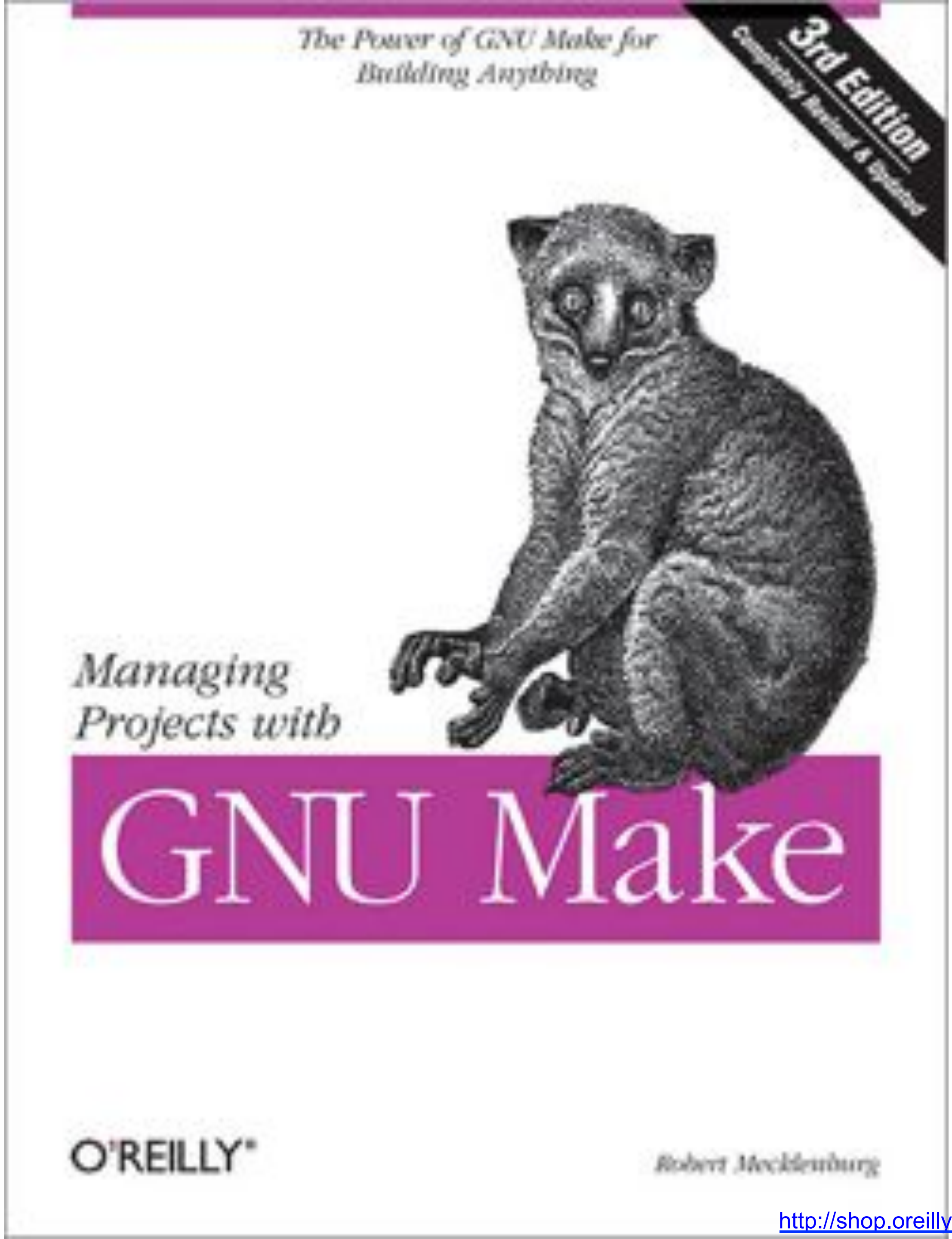

The Power of GNU Make for Building Anything
3rd Edition
Completely Revised & Updated
Managing Projects with
GNU Make
O'REILLY
Robert Mecklenburg

http://shop.oreilly.com/product/9780596006105.do

GNU make

GNU Make is a tool which controls the workflow of generating target/result files from the dependencies (source files). Target/result files may be text files, standalone programs, packages

Capabilities of Make

- Make is for more than just a tool for compiling software
- The path from raw data to final results
- Automates/documents a workflow
- Intelligently handles the dependencies among data files, code
- Accounts for the updates in data, code
- Re-runs only the necessary code, based on what has changed

https://www.gnu.org/software/make/

Makefile structure

Makefile contains **recipes** in the form of:

```
target: dependencies
   <code>
```

– **`target`**	the outcome
– **`dependencies`**	the necessary parts to build the outcome
– **`code`**	outlines the rules to build target using dependencies

- All commands must be **tab**-indented
- Dependencies, if more than one, must be **space**-separated

Makefile example

```
# An example of obtaining counts and types of the cytobands
all:  cytoband_counts.txt cytoband_types.txt

# Download the raw data
cytoBand.txt.gz:
	wget http://hgdownload.cse.ucsc.edu/goldenPath/hg19/database/$@

# Obtain counts of the cytobands
cytoband_counts.txt:	cytoBand.txt.gz
	zcat < $< | cut -f1 | sort | uniq -c | awk '{OFS="\t"} {print $$2,$$1}' | sort -
k2 -nr > $@

# Obtain types of the cytobands
cytoband_types.txt:  cytoBand.txt.gz
	zcat < $< | cut -f5 | sort | uniq -c | awk '{OFS="\t"} {print $$2,$$1}' | sort -
k2 -nr > $@

clean:
	rm *.gz
```

Makefile structure

Typical recipes

- **`clean`** **–** commands to clean up the working directory from temporary files
- **`test`** **–** runs a series of tests
- **`install`** **–** installs a software
  - **`./configure`**
  - **`make`**
  - **`make install`**

How to use make

- If you name your make file **`Makefile`**, then just go into the directory containing that file and type **`make`**

- If you name your make file **`something.else`**, then type

```
make -f something.else
```

- By default, **`make`** builds the first target listed in the **`Makefile`**. Generally, the first target generates all other targets

```
all: target1 target2 target3
```

- To build a specific target, type **`make target`**. For example,

```
make cytoband_counts.txt
```

Make variables

- A variable is a name defined in a makefile to represent a string of text, called the variable's value. Variables are used to simplify recipes

- Defining internal Makefile variable

```
DB = "/home/genomerunner/db_2.00_06.10.2014"
```

- Using a variable

```
${DB} or $(DB)
```

https://www.gnu.org/software/make/manual/make.html#Using-Variables

Using shell variables in Make

Shell variables, e.g. **`$HOME`**, need to be prefixed by **`$`**

`awk '{print $0}'`	within shell variable use
`awk '{print $$0}'`	within Makefile variable use

Capturing output of shell commands into a variable:

```
TXT_FILES = $$(shell find . -type f -name "*.txt")
```

TIP!

- The content of a Makefile runs in its own shell environment. The default shell environment is **`/bin/sh`**. To set shell environment to **`bash`**, use

```
SHELL=/bin/bash
```

Why bother?

- Variable **`$SECONDS`** exists in **`bash`**, but not in **`sh`**
- Other syntax incompatibilities, e.g., **`if-else-fi`** syntax

Automatic variables

Makefile contains **recipes** in the form of:

```
target: dependencies
   <code>
```

`$@`	the name of the target of the rule
`$<`	the name of the first dependency
`$?`	the names of all the dependencies
`$(<F)`	the file part of the first dependency

Example:

```
   COMPILER = g++               # Define compiler
   COMPILER_FLAGS=-c -Wall      # Define flags

   hello.o: hello.c hello.h     # Recipe
           $(COMPILER) $(COMPILER_FLAGS) $< -o $@
```

https://www.gnu.org/software/make/manual/make.html#Automatic-Variables

Patterns

A pattern rule allows “wildcard” matching between the target and the dependencies. The ‘`%`’ wildcard is similar to the ‘`*`’ wildcard in bash

- Existing files:
  - `module0_induction.Rmd`
  - `module1_basics.Rmd`
  - `module2_managingR.Rmd`
- Makefile recipe:

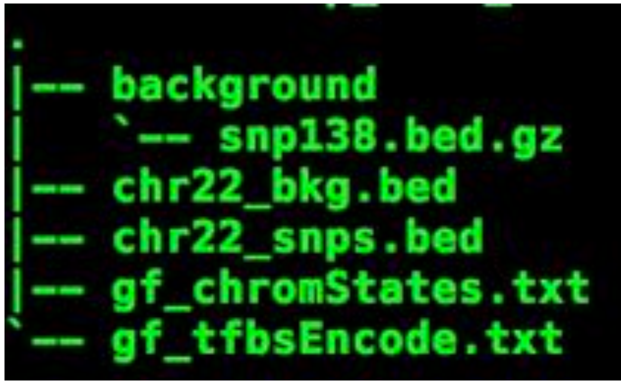
```
%.html:  %.Rmd
        echo $(@)
        ./compile_slides $(basename $(@))
```

- Results:
  - `module0_induction.html`
  - `module1_basics.html`
  - `module2_managingR.html`

https://www.gnu.org/software/make/manual/make.html#Pattern-Rules

Canned recipes (functions)

```
# Prefix for the '_snps.bed' and '_bkg.bed' file names
INP = chr22

# Path to the genomic background
BKG = background/snp138.bed.gz

# Types of analyses to run
AN1 = chromStates
AN2 = tfbsEncode

# Variable/function to execute the $@ analysis
define hypergeom4
python -m grsnp.hypergeom4 --output_dir $@ $(INP)_snps.bed gf_$@.txt $
(INP)_bkg.bed
endef

all:          .$(AN1) .$(AN2)

.$(AN1):      $(INP)
              $(hypergeom4)

.$(AN2):      $(INP)
              $(hypergeom4)
```



```
.
|-- background
|   `-- snp138.bed.gz
|-- chr22_bkg.bed
|-- chr22_snps.bed
|-- gf_chromStates.txt
`-- gf_tfbsEncode.txt
```

https://www.gnu.org/software/make/manual/make.html#Canned-Recipes