Reproducible commands and workflows with `make`

The GNU `make` program was created in 1976 to help build executable programs from source code files. While it was originally developed to assist with programming in the `c` language, it is not limited to that language or even to the task of compiling code. According to the manual, one "can use it to describe any task where some files must be updated automatically from others whenever the others change." The `make` program has gone far beyond it's role as a build tool to become a workflow system.

Makefiles are recipes

When you run the `make` command, it looks for a file called `Makefile` (or `makefile`) in the current working directory. This file contains recipes that describe discrete actions that combine to create some output. Think of how a recipe for a pie has discrete steps that need to be completed in such a way that the crust, filling, and meringue are all ready at the appropriate stages.

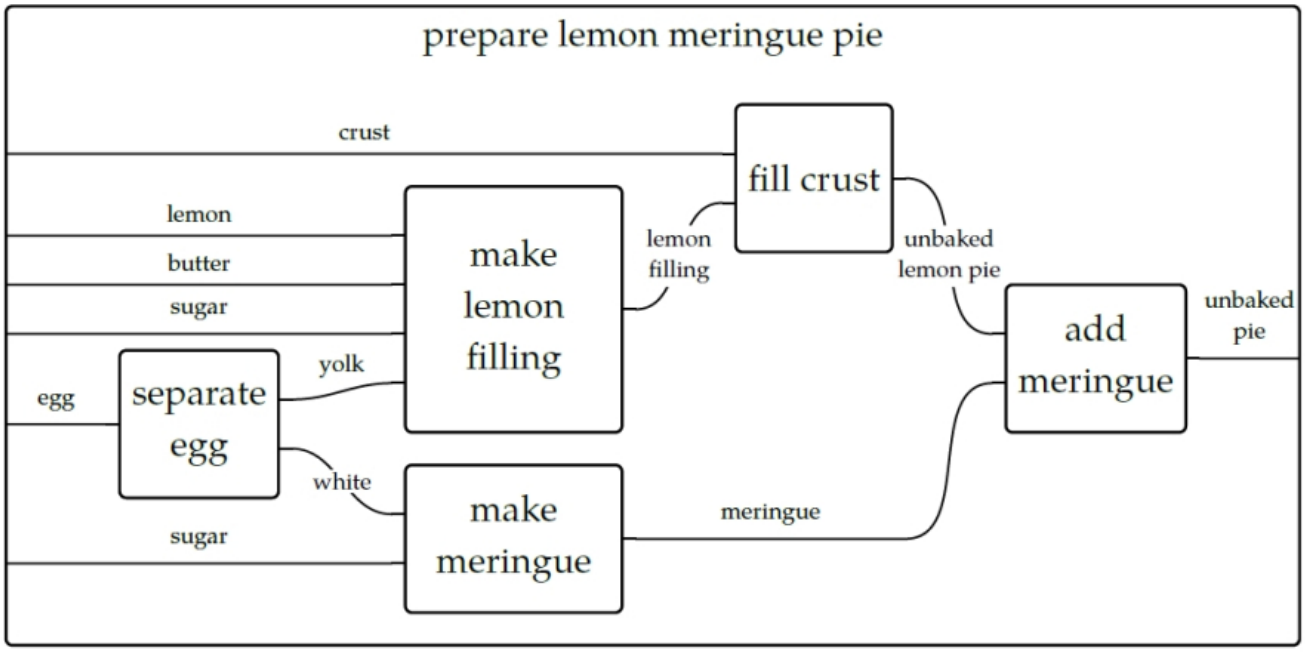

Figure 1. A string diagram describing how to make a pie. From Brendan Fong and David Spivak, Seven Sketches in Compositionality: An Invitation to Applied Category Theory.

It's not really important if you make the pie crust the day before and keep it chilled, and the same might hold true for the filling, but it's probably true that the crust needs to go into the dish before the filling, and the whole pie needs to be baked before you add the meringue.

Makefiles

When you run `make`, it looks for a `Makefile` text file in the current working directory. There is a Makefile in this directory that has a single target called `makefiles.pdf` that builds a PDF called "makefiles.pdf" from the source text "makefiles.adoc" using the `asciidoctor-pdf` program:

```
$ cat Makefile
makefiles.pdf:
    asciidoctor-pdf makefiles.adoc && open makefiles.pdf
```

The `Makefile` is like a file of recipes

```
all: crust filling meringue
    ./combine.py crust filling meringue
    ./cook.py 375 45

filling:
    ./combine.py cherries sugar water

meringue:
    ./combine.py eggwhites sugar

crust:
    ./combine.py flour butter water
```

describes "targets" of files that need to be created. Each target may have one or more dependencies, and those dependencies may, in turn, also have dependencies.

These actions create a graph structure where there is some starting point and paths through targets to finally create some output file(s). Many bioinformatics analysis pipelines are just that — a graph of some input like a FASTA sequence file and some transformations (trimming, filtering, comparisons) into some output (e.g., BLAST hits, gene predictions, functional annotations). You would be surprised at just how far `make` can be abused to document your work and even create fully functional analysis pipelines.

Using `make` to compile a `c` program

I believe it helps to use `make` for its intended purpose in order to really understand why it exists. Let's take a moment to write and compile our "Hello, World" example in `c`. In the `c-hello` directory, you will find a simple `c` program that will print "Hello, World!".

Here is the `hello.c` source code:

```
#include <stdio.h>            ①
int main() {                  ②
   printf("Hello, World!\n"); ③
   return 0;                  ④
}                             ⑤
```

Let's take a moment to learn just enough `c` to be dangerous going line-by-line:

① Like `bash`, the `#` character introduces comments in the `c` language, but this is a special comment that allows external modules of code to be used. Here, we want to use the `printf` (print-format that we saw in the previous chapter), so we need to `include` the standard I/O (input/output) module called `stdio`. We actually only need to include the "header" file, `stdio.h`, to get at the function definitions in that module. This is a standard module, and the `c` compiler will look in various locations for any included files to find it. There may come times when you are unable to compile `c` (or `c++` programs) from source code because some header file cannot be found. For example, the `gzip` library is often used to de/compress data, but it is not always installed in a libary form that other programs may `include` in this way. Therefore you will have to download and install the `libgz` program, being sure to install the headers into the proper `include` directories. Note that package managers like `apt-get` and `yum` often have `-dev` or `-devel` packages that you have to install to get these headers, e.g., you would install both `libgz` and `libgz-dev` or whatnot.

② This is the start of a function declaration in `c`. The `int` (an "integer") is the return value of the function called `main()`. The parentheses `()` list the parameters to the function. There are none, so the the parens are empty. The opening curly brace `{` shows the start of the code that belongs to the function. Note that `c` will automatically execute the `main()` function, and every `c` program must have a `main()` function where the program starts.

③ The `printf()` function will print the given string to the command line.

④ `return` will exit the function and will return the value `0`. Since this is the return value for the `main()` function, this will be the exit value for the entire program. The value `0` indicates that the program ran normally — think "zero errors." Any non-zero value would indicate a failure.

⑤ This curly brace `}` is the closing mate for the one on line 2 and marks the end of the `main()` function.

To turn that into an executable program you will need to have a `c` compiler on your machine. We can use the `gcc` (GNU c compiler) with this command:

```
$ gcc hello.o
```

That will create a file called `a.out` which is an executable file. On my Mac, this is what `file` will report:

```
$ file a.out
a.out: Mach-O 64-bit executable x86_64
```

And I can execute that:

```
$ ./a.out
Hello, World!
```

I don't like the name `a.out`, though, so I can use the `-o` option to name the output file called `hello`:

```
$ gcc -o hello hello.c
```

Run the resulting `hello` executable. You should see the same output.

Rather than typing `gcc -o hello hello.c` every time I modify the `hello.c`, I can put that as a "target" into a `Makefile`. A target is a single word (no spaces) on a line followed by a colon. The actions associated with that target must be indented by a `Tab` character (not spaces!):

```
hello:
    gcc -o hello hello.c
```

And now I can type `make hello` to explicitly run the "hello" target which will then run the shell command(s) listed. Since this is the first (and only) target, I could also run `make` and the first target will be executed.

This is clearly a trivial example, and you may be wondering how this is actually a time saver. A real-world project in `c` or any language would likely have multiple `.c` files with headers (`.h` files) describing their functions so that they could be used by other `.c` files. The `c` compiler would need to turn each `.c` file into `.o` ("out") files and then link them together into a single executable. Imagine you have dozens of `.c` files, and you change one line of code in one file. Do you want to type dozens of commands to recompile and link all your code? Of course not! You would build a tool to automate those actions for you.

`make` will actually go one step further and inspect the timestamps of the `.c` (input) files and the output file it should make. If an input file hasn't changed since the output file was last generated, it will skip the compilation step. Now imagine the inputs are dozens of FASTA files and the outputs are BLAST hits that might take several minutes to hours to generate. If you add or modify a FASTA file, there's no reason to re-run the previously existing files if their resulting hits already exist. `make` can see that it can skip those steps and move on to the next target in the graph!

We can add targets to the `Makefile` that don't generate new files. It's common to have a `clean` target that will clean up files and directories that we no longer need. Here I can create `clean` target to remove the `hello` executable.

```
clean:
    rm -f hello
```

If I want to be sure that the exeuctable is removed before every running the `hello` target, I can add

it as a dependency:

```
hello: clean
    gcc -o hello hello.c
```

It's good form to document for `make` that this is a "phony" target because the result of the target is not a new file to "make." We use the `.PHONY:` target and list all the phonies. Here is our complete `Makefile` now:

```
$ cat Makefile
.PHONY: clean

hello: clean
    gcc -o hello hello.c

clean:
    rm -f hello
```

If you `make` in the `c-hello` directory with this `Makefile`, you should see this:

```
$ make
rm -f hello
gcc -o hello hello.c
```

And there should now be a `hello` executable in your directory that you can run:

```
$ ./hello
Hello, World!
```

Notice that the `clean` target can be listed as a dependency to the `hello` target even *before* the target itself is mentioned. `make` will read the entire file and then use the dependencies to resolve the graph. If you were to put "foo" as an additional dependency to `hello` and then try to running `make` again, you would see this:

```
$ make
make: *** No rule to make target `foo', needed by `hello'.  Stop.
```

When we write `bash` programs, the program is executed from the top to the bottom, each statement one after the other. The `Makefile` allows us to write independent groups of actions that are ordered by their dependencies. They are essentially like *functions* in a higher-level language. We have essentially written a program who's output is ... a program.

I'd encourage you to `cat hello` to see what the `hello` program "looks" like. It's mostly binary information that will look like jibberish, but you will probably be able to make out some plain

English, too. You can also use `strings hello` to extract just the "strings" of text.

Using `make` for a shortcut

Let's start off with how to abuse Makefiles to create shortcuts for commands. Here we will say "Hello, World!" on the command line using the `echo` command:

```
.PHONY: hello            ①

hello:                   ②
    echo "Hello, World!" ③
```

① Since the `hello` target doesn't actually produce a file, we list it as a "phony" target.

② This is the `hello` target. The name of the target should be composed only of letters and numbers, should have no spaces before it, and is followed by a colon (`:`).

③ The command(s) to run for the `hello` target are listed on lines that are indented with a tab character.

Writing a workflow to find unclustered proteins

We'll revisit the exercise of finding unclustered proteins, this time using `make` to accomplish the task. Here is a `Makefile` I wrote to sequence the shell commands:

```
$ cat -n Makefile
     1  # Author: Ken Youens-Clark <kyclark@gmail.com>
     2  # Purpose: Find proteins that do not cluster with CD-HIT
     3
     4  .PHONY: clean report
     5  PROTEINS_FASTA = "proteins.fa"
     6  CD_HIT_CLUSTERS = "proteins.fa.cdhit.clstr"
     7  CLUSTERED_IDS = "clustered_ids"
     8  SORTED_IDS = "sorted_ids"
     9  UNCLUSTERED_IDS = "unclustered_ids"
    10
    11  all: clean report
    12
    13  clean:
    14      rm -f $(CLUSTERED_IDS) $(SORTED_IDS) $(UNCLUSTERED_IDS)
    15
    16  sorted_ids:
    17      grep -e '^>' $(PROTEINS_FASTA) | sed "s/^>//; s/|.*//" | sort >
$(SORTED_IDS)
    18
    19  clustered_ids: sorted_ids
    20      grep -v '^>' $(CD_HIT_CLUSTERS) | awk '{print $$3}' | sed "s/^>//; s/[^0-
9].*//" | sort | uniq > $(CLUSTERED_IDS)
    21
    22  unclustered_ids: clustered_ids
    23      comm -23 $(SORTED_IDS) $(CLUSTERED_IDS) > $(UNCLUSTERED_IDS)
    24
    25  report: unclustered_ids
    26      $(eval NUM_UNCLUSTERED=$(shell wc -l $(UNCLUSTERED_IDS) | awk '{print
$$1}'))
    27      @echo \"$(NUM_UNCLUSTERED)\" proteins in \"$(UNCLUSTERED_IDS)\"
```

Let's discuss:

- Lines 1-2: `make` uses the `#` just like `bash` to indicate text that should be ignored. Here I'm adding two comments so that future users know who wrote this, how to contact me if they have questions, and what this is supposed to do.
- Line 4: I create a `.PHONY` target to hold the two targets that do not actually "make" a file. They are the `clean` target (that removes intermediate files) and the `report` target (that tells the user the result of the analysis).
- Lines 5-9: I'm creating variables to hold the names of the various files that will be used and created in the analysis. Note that, unlike `bash`, Makefiles *requires* spaces around the `=` when you assign a value to a variable. Note that I'm hardcoding the names of the inputs files (`PROTEINS_FASTA` and `CD_HIT_CLUSTERS`) where in the `bash` script I took them from the positional arguments `$1` and `$2`. While it's possible to pass arguments to `make`, e.g. `make PROTEINS_FASTA=proteins.fa CD_HIT_CLUSTERS=proteins.fa.cdhit.clstr`, I think it borders on the unwise. What I'd rather you take from this example is how we are using `make` to document and reproduce a sequence of commands. When it comes to implementing more complex analyses

that need to be parameterized, I would recommend you write the pipeline in a language like Python or use an actual workflow system which we will discuss shortly.

- Line 11: Recall that the first target is the one run by default when no target is specified, e.g., you just run `make`. It's common to call the first target `all` meaning that it will run "all" the steps. My `all` target doesn't actually have any commands itself to run but is rather comprised of dependencies listed in the order I want them executed. First I want to `clean` the directory to ensure that all the intermediate files are removed. Recall that `make` may choose to not run a target if it sees that the output of the target already exists. Here I want to be sure that every target is run every time. After `clean`, I specify the last target `report` which itself has a dependency of `unclustered_ids` which itself has a dependency of `clustered_ids` and so on. Note that `$()` in `bash` was the syntax to call an external process in `bash`, but in `make` syntax this is how we deference (or interpolate) a variable's value. Also note that we are able to list targets that have not yet been defined by this point in the `Makefile`.
- Line 16-17: The target to created the "sorted_ids" file. Note that I chose to make the name of the target to be the same as the name of the file that is created, just as in a normal `Makefile` that we might use to build an executable.
- Line 19-20: The target to build the "clustered_ids" is a long one. The pipes make it difficult to break this onto multiple lines. Since `make` does not really care about line length, it was easiest to leave this as-is. Note that I had to add an extra `$` in the `awk` command so that `make` would not try to interpole the `$3` that needs to be passed literally to `awk`. Before this target can be run, we require `sorted_ids` to exist.
- Lines 22-23: The target to run find the `unclustered_ids` by running the `comm` command. I list the `clustered_ids` as a dependency so that it will be run first.
- Lines 25-27: The target to `report` on the number of proteins we found. I list `unclustered_ids` as a dependency that must be run first. Note the shenanigans required to run a `shell` command and capture the output into a variable. This is truly painful syntax that I show you not to convince you to use it but to make you hope that there must a better way. Just because you *can* do something doesn't mean you *should* do it. Would you really want to write and maintain this code?

Writing workflows and pipelines

We've seen how we can use the Unix pipe (`|`) to chain the output of one command as the input to another. This is the meaning of an analysis "pipeline" where we gradually transform some input into some desired output, often by using mulitiple steps and tools. Often the output from a program is almost, but not quite, entirely unlike that which is required for the next program we want to use, so we have to write out own code to massage the data. Another name for all this work is a "workflow" where we define, in some manner, the "flow" of the inputs through the transformations to some eventual output.

In chapter 1, we started out on the Unix command line learning how to issue single commands manually to effect some changes. We saw that we can combine multiple commands with pipes to create files which we used in subsequent actions. Then in chapter 2, we learned enough `bash` to be able to put all those commands into a single program which we could parameterize and reuse on new input files. Here in this chapter, we've seen that we can describe each step or transformation

as "targets" in a `Makefile` and let `make` figure out the order to execute the steps based on the dependencies for each target. (Conceptually, we moved from the idea of an "imperative" approach where we manually noted the actions and their order to a "declarative" approach where we noted the actions and their *dependencies* and let something else infer their *order*.)

As cool as `make` is, the syntax is a bit painful and easily muddled with that of `bash`. It's still worthwhile to understand how to use a `Makefile` to document and reproduce a workflow. If nothing else, I constantly use a `Makefile` to document how I ran some particular command for a given project. Maybe I want to document a long, complex command that I'm afraid I'll never remember, or maybe I want to run some analysis over mulitple input data sources. While I *could* write a `README` file where I document all my commands, I can easily *run* those commands using `make` if I go the step further to put them into a `Makefile`.

I would recommend you try writing one or two of your pipelines with a `Makefile` just so you can see what it's like. I will include an exercise in the GitHub repo that manipulates some yeast data. You may surprised yourself with just how far you can go in use `make` to run workflows. The commands for a target need not be limited to primitive shell commands. You can write full programs in `bash` and Python that get run alongside other programs like BLAST and whatnot. As you bump up against the limitations of `make`, you may choose to move to a workflow manager. There are literally hundreds to choose from including:

- Snakemake which extends the basic concept `make` with Python.
- The Common Workflow Language (CWL) defines workflows and parameters in a configuration file (in YAML), and you use tools like `cwltool` or `cwl-runner` (both implemented in Python) to execute the workflow with another configuration file that describes the arguments.
- The Workflow Description Language (WDL) takes a similar approach to describing workflows and arguments and can be run with the Cromwell engine.
- Pegasus allows you to use Python code to describe a workflow that then is written to an XML file that is the input for the engine that will run your code.
- Nextflow is similar in that you use a full programming language called "Groovy" (a subset of Java) to write a workflow that can be run by their engine.

All of these systems have the same basic ideas as `make`, so understanding how `make` works and how to write the pieces of your workflow and how they interact is the basis for any larger analysis workflow you may create.