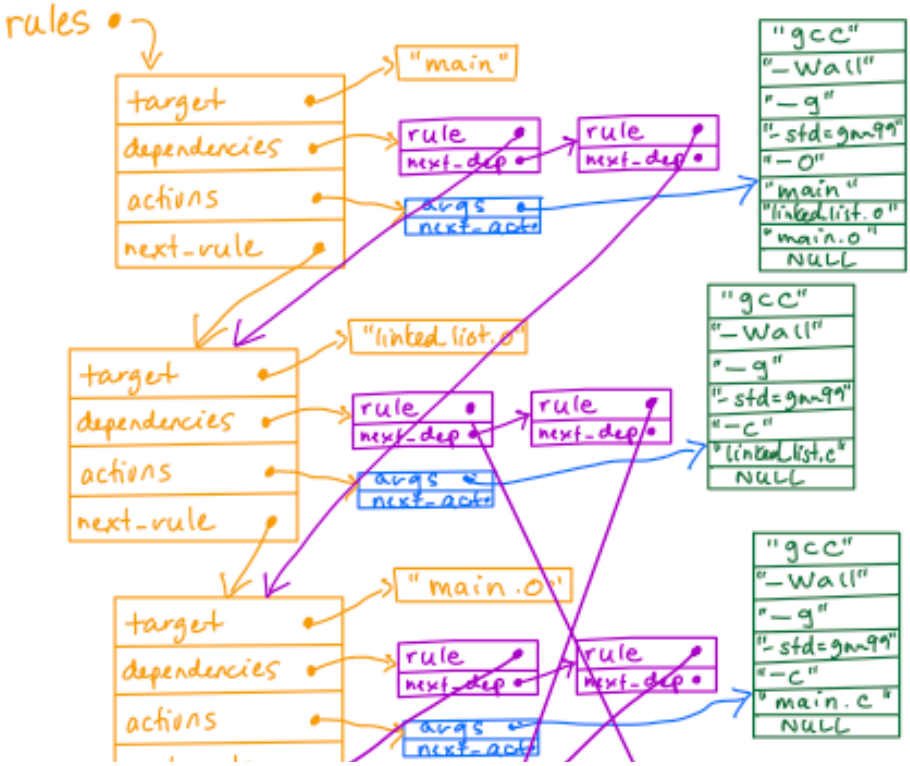
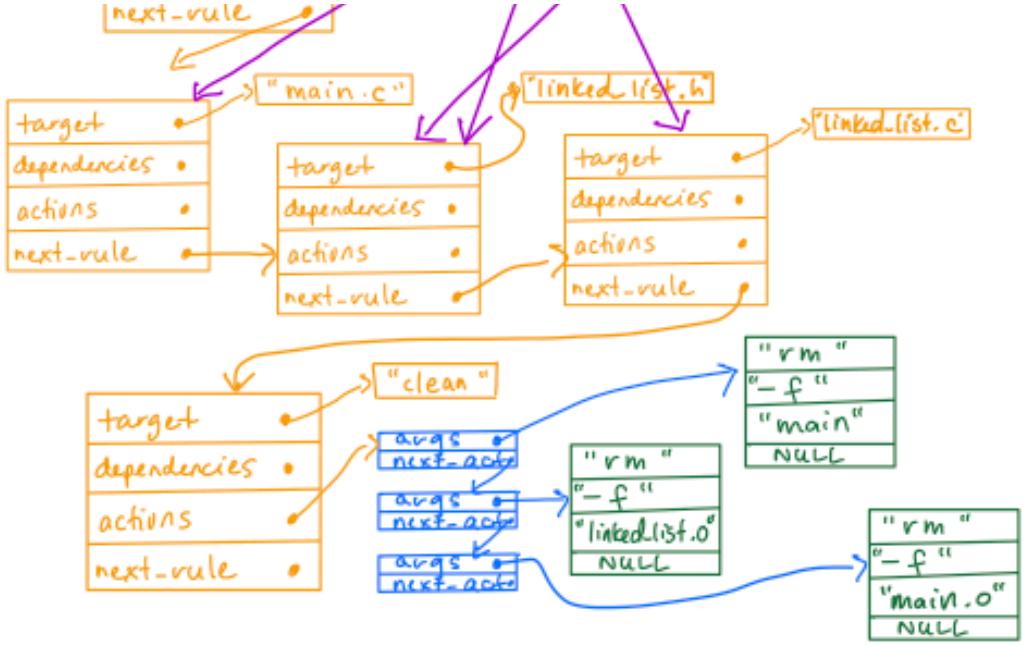
We have divided the development of your solution into three parts. You should fully implement each part and add and commit to your repository before moving on to the next part. The starter code includes the complete `pmake.c` file which you must not modify. Read this file to see how to run your program and how it makes use of the functions that you will write.

Part 1: Parsing a Makefile

Your first task is to implement `parse_file`, so that it reads a makefile and constructs a corresponding linked data structure. The necessary structs are defined in `pmake.h`, and the comments at the top of the file explain what each line of a makefile will contain. You may not change the struct definitions. `pmake.h` also contains function prototypes for functions that are either already provided, or that you will need to write.





The starter code also contains a makefile named `handout.mk`. (This file and the required source and header files are in the subdirectory `handout_example`.) The picture above shows a data structure that could result from parsing this file. Notice that there is one `struct rule_node` for each rule in the original makefile and an additional `struct rule_node` for each dependency that is not also a target. You must follow this design. You are are also required to have the head of the list be the first rule in the original makefile. However, depending on how you do your parsing, the `struct rule_node` elements may come in a different order in your linked list of rules.

Before reading any more of this handout, spend time to make sure you understand this figure and how it connects to the original makefile and the structs defined in `pmake.h`.

You will see that the actions are stored in a `struct action_node` that has a member `args`. This array has the format required by the second parameter to `execvp`. Use the man page for `execvp` to understand this format. In particular, notice that the first element is the executable name, subsequent elements are the arguments for that executable, and these are followed by an extra `NULL` pointer element.

Makefile format

The real make program supports a fairly complex syntax for writing makefiles. For the purposes of this assignment, your implementation of `pmake` only needs to support makefiles with the following simplified syntax.

A line from a makefile is either a target line, an action line, or a comment or line to ignore.

- Target line
  - Starts with a target word, followed by a colon, followed by a space-separated list of dependencies.
  - The colon has space on each side of it.
- Action line
  - Begins with a tab
  - After the tab is a command that can be executed in the shell.
  - Must follow a target line or another action line (possibly with comments/empty lines in between).
- Comment or empty line
  - Contains only spaces and/or tabs.
  - Or, contains 0 or more spaces/tabs followed by a `#`. Any characters following the `#` are ignored.

You may make the following **assumptions** about the format and contents of the Makefile:

- The Makefile syntax is valid (i.e., follows the structure described above).
- Each target name, dependency name, and action word contains **no spaces**. This allows you to tokenize each line by splitting on spaces.
- Every line in the file contains **at most `MAXLINE` characters**, where `MAXLINE` is a macro defined in `pmake.h`. This limit includes the newline character `\n` at the end of a line, if present. It does not include a null-terminator character.
- Does not contain variables (e.g., `$@`), wild cards (e.g., `%.o`), pattern rules, `.PHONY`, or special characters at the start of an action line (e.g., `@` or `-`).
- The Makefile contains at least one rule.
- The Makefile does not contain any circular dependencies (e.g., where target A depends on B, and target B depends on A).

When you have completed Part 1, the `print_rules` function should print the `rules` data structure as a makefile. In other words, we should be able to save the output to a file and use it with the regular `make` program. The `print_rules` function has been given in the starter code and you must not change it. To run your `pmake` program on the handout example and see the output generated from `print_rules` you should `cd` into `handout_example` and then run `../pmake -f handout.mk -o`. All of your work for this part of the assignment will be done in `parse.c`.

(Added Feb 27.) When your `pmake` program is called with a target that does not exist in the makefile, your program should print a helpful error message and exit with a non-zero return code.

Part 2: Evaluating a Makefile Rule

The second task is to implement function `run_make` in `run_make.c`. `run_make` takes a target as its first argument. It finds the rule in the `rules` data structure corresponding to the target, and evaluates it. If `target` is `NULL`, then run_make will evaluate the first target in the rules list.

A rule is evaluated using the following steps:

1. Update each of the dependencies. In other words, recursively evaluate each dependency rule.
2. Compare the last modified time for each of the dependencies to the target.
3. If the target does not exist, or at least one of the dependencies refers to a file with a last modified time more recent than the target, execute the rule's actions.

Last modified time

Use `stat` to get the last modified time (mtime) of a file. Read the man page (using `man 2 stat`) to learn how to call `stat`. The field of the `stat struct` that you want is called `st_mtim` on Linux. It has different names on Mac OSX, so if you have a Mac, you will need to either work on the Linux machines or remember to change your code when you do the final tests on the teach.cs machines. It may also be different on Windows Ubuntu, so you should check.

The `struct timespec st_mtim;` has two fields, one for seconds (`tv_sec`) and one for nanoseconds (`tv_nsec`). You will need both of these values to find out which time is more recent.

Executing actions

You will need to use `fork` and `execvp` to the run the command specified by an action. Fortunately in Part 1, you already prepared the array of arguments to pass into `execvp`. The parent process should wait for each child to complete. If the child terminates with a non-zero exit status, the parent should also terminate with a non-zero exit status. This means that `pmake` will stop when it encounters an error.

The actions for one rule are executed sequentially (one after another), not in parallel.

Your implementation must also print each action line to stdout immediately before it is executed, just like the real `make` program. (`make` has a way to disable this using the `@` symbol, but we will not allow those symbols in our input makefiles for this assignment.)

Part 3: Parallelizing pmake

When `pmake` is run with the `-p` option, a child should be created to update each dependency. The parent will create one child process for each dependency and after it has created all of them, the

parent will wait for all children to terminate successfully before determining whether it needs to execute the actions.

As when executing actions in Part 2, if a child terminates with a non-zero exit status, the parent should also terminate with a non-zero exit status. In this part, this means that the `pmake` will stop when evaluating one of its dependencies fails.

Submitting your work

For marking we will only be collecting your `parse.c` and your `run_make.c`. We will test these with the starter code versions of the other files. If you write a helper function for `parse.c` that you wish to call from `run_make.c`, you will need to put its prototype in `run_make.c` (since you are not permitted to create additional files or add to `pmake.h`).

If you design your solution carefully, there is not a lot of code to write for this assignment. Our solution adds approximately 300 lines to the starter code (counting many comments). You should use good design principles to make use of helper functions to avoid unnecessary repeated code. Your program should be well-documented and readable.