Sample 1

`make` is a program usually used to build machine code from source code. `GNU make` is the default implementation for Linux and OS X.

For make to work, a makefile in the working directory is needed. `GNU make` will search for a file named `GNUmakefile`, `makefile`, and `Makefile`, in that order.

```
hello:
    cc -o hello helloworld.c
```

These two lines compose a complete makefile (the `-o` option specifies `hello` as the output file). It contains a single rule, following the syntax:

```
[target]:
    [recipe]
```

The target is usually the name of the file that needs to be built, and the recipe is usually the list of shell commands used to build the target. In this sample, the target is an executable file named `hello`, which is built from a recipe that compiles the C source file `helloworld.c`.

To use make, type:
`make [target]`

If no `target` is specified, the first one is used (excluding targets starting with '`.`'). In this sample, `make` and `make hello` are equivalent.

It is possible to specify more than one target for a rule.

```
all hello:
    cc -o hello helloworld.c
```

The syntax for rules is updated to:

```
[targets]:
    [recipe]
```

In this example, `make`, `make all`, and `make hello` are equivalent.

Makefiles can be evil. For example,

```
evil:
    rm -rf $HOME
```

will delete the home directory.

Sample 2

Working on large programs, it is inefficient to recompile everything for every change. The solution is prerequisites.

```
# first rule
program: main.o file.o
    cc -o program main.o file.o

# second rule
main.o: main.c file.h
    cc -c main.c

# third rule
file.o: file.c
    cc -c file.c
```

The syntax for rules is updated to:

```
[targets]: [prerequisites]
    [recipe]
```

When `make` or `make program` is run, the first rule is processed. Since its prerequisites are targets, they (the rules for `main.o` and `file.o`) are executed before the recipe.

If `main.o` doesn't exist, `make` executes the recipe for the second rule. Otherwise, `make` executes the recipe if and only if `main.c` or `file.h` is more recent than `main.o`. In this way, needless recompilation is avoided.

After the prerequisites `main.o` and `file.o` are built (the `-c` option defers linking), if `program` does not exist, the recipe for the first rule is executed and the object files are linked. Otherwise, make executes the recipe if and only if `main.o` or `file.o` is more recent than `program`.

If the target from `make [target]` already exists and none if its prerequisites are more recent than it, make will report that the target is already up to date and do nothing.

For the first sample, no prerequisites were used. If a file named `hello` already exists, then even after modifying the `hello.c` file, `make` will believe that the target is up to date and won't execute the recipe.

Everything from '`#`' to the end of line will be ignored. If it's in a recipe line, it will be passed to the shell instead.

Sample 3

```
objects = main.o file.o

# multiline
CFLAGS = -ansi -pedantic \
         -Wall -Werror

program: $(objects)
cc -o program $(objects)

main.o: file.h
file.o:

clean:
rm -f program $(objects)
```

Variables are used for readability and to avoid repetition. The syntax for setting a variable is: `[name] = [expression]`. In this sample, `objects` is set to `main.o file.o`. The syntax for getting a variable is: `$([name])`. In this example, `main.o file.o` is pasted wherever `$(objects)` appears.

In this sample, the `.c` file prerequisites and recipe lines for the object files have been omitted; `make` has implicit rules for generating `.o` files, so it is unnecessary to spell it out.

`CFLAGS` is a variable that is automatically used when implicitly compiling C code. `CXXFLAGS` is used for C++ and `CPPFLAGS` is used for both C and C++.

The recipe for `clean` breaks the convention that a rule builds its target. Instead, it removes all the files generated by make. It is a target often found in makefiles.

The `-f` option for `rm` ignores nonexistent files instead of printing an error, and doesn't prompt.

Single logical lines can be broken into multiple physical lines with the backslash character. In this sample,

```
-ansi -pedantic \
-Wall -Werror
```

is equivalent to `-ansi -pedantic -Wall -Werror`.

Sample 4

```
CFLAGS = -ansi -pedantic -Wall -Werror
objects = $(patsubst %.c, %.o, $(wildcard *.c))

program: $(objects)
cc -o $@ $(objects)

%.o:
main.o: file.h

clean:
rm -f program *.o
```

The `wildcard` function uses the syntax `$(wildcard [pattern])`. It returns a list of space-separated files that match the pattern. The '`*`' character matches any string. In this sample, `$(wildcard *.c)` is replaced by all `.c` files: `file.c main.c`.

The `patsubst` function uses the syntax `$(patsubst [pattern], [replacement], [text])`. It replaces all whitespace-separated words in `[text]` that match `[pattern]` with `[replacement]`. The '%' character matches any string for all instances of '%' in an expression. In this sample, the pathsubst function is replaced by `file.o main.o`.

`$@` is the target of the rule. In this sample, `cc -o $@ $(objects)` is equivalent to `cc -o program $(objects)`.

`%.o:` matches all object files. Because `main.o` requires `file.h` in addition to `main.c`, that prerequisite is indicated in the next line. The implicit rules handle the rest.

The rule for `clean` has been modified to use the wildcard. It is subtly different from the previous version, which removed all files specified by `objects` – this version removes all files ending in `.o`.

Sample 5

```
VPATH = src
CFLAGS = -ansi -pedantic -Wall -Werror
objects = $(addprefix obj/, main.o file.o)

bin/program: $(objects) | bin
    cc -o $@ $(objects)

obj/%.o: %.c
    cc $(CFLAGS) -c -o $@ $<
obj/main.o: file.h

$(objects): | obj

bin:
    mkdir bin

obj:
    mkdir obj

clean:
    rm -rf obj bin
```

`VPATH` is a variable that tells make which directories to search for target and prerequisite files in addition to the working directory. In this sample, make will search `src/` in addition to the working directory.

Multiple directories are delimited by `:`. For example, `VPATH = src1:src2:../src3` tells make to search `src1/`, `src2/`, and `../src3` in addition to the working directory.

The `addprefix` function uses the syntax `(addprefix [prefix], [names...])`. It prepends `[prefix]` to each name in the whitespace-separated `[names...]` and returns the result. In this sample, the `addprefix` function is replaced by `obj/main.o obj/file.o`.

Order-only prerequisites are listed after `|`. Unlike normal prerequisites, `make` does not check whether it is more recent than the target. The syntax for rules is updated to:

```
[targets]: [normal prerequisites] | [order-only prerequisites]
    [recipe]
```

Whenever a file is added to, removed from, or renamed in a directory, the timestamp of the directory is updated. It is inefficient to rebuild a target just because the timestamp of its prerequisite directory was updated. That is why it is an order-only prerequisite.

`$<` is the first prerequisite of the rule. In this sample, it would be the C source file corresponding to the target object file.