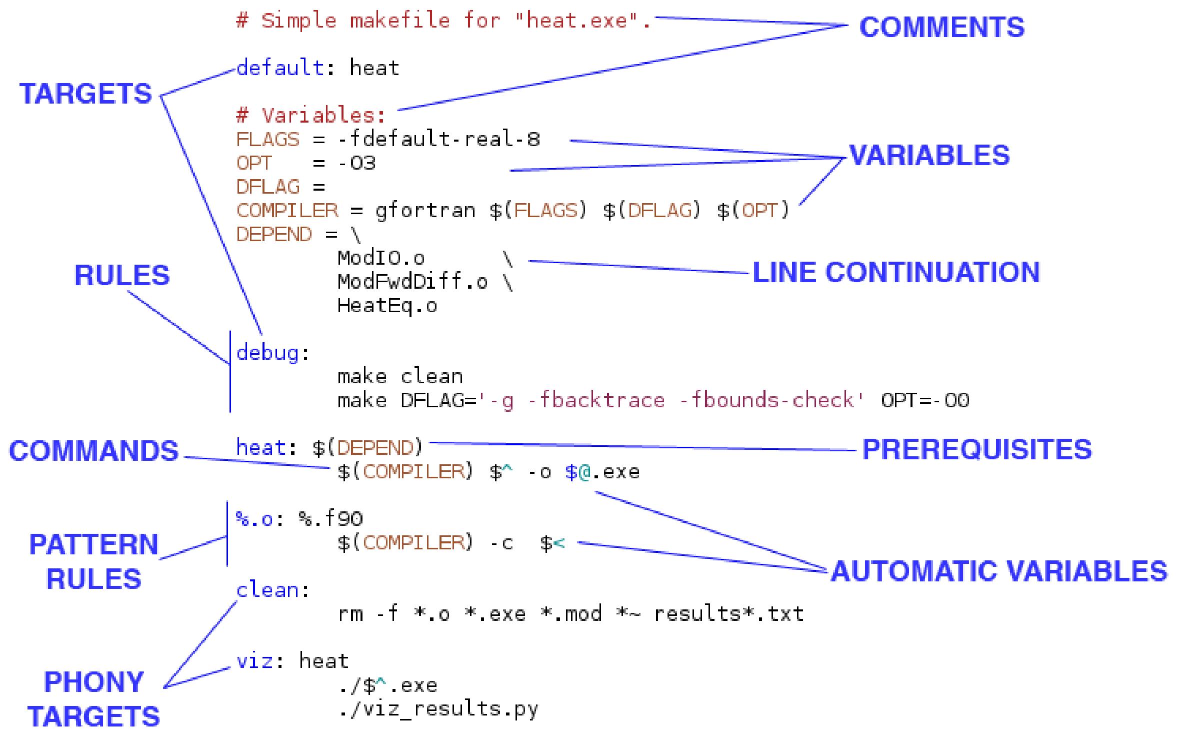
GNU Make: A Crash Course

D. T. Welling (dwelling@umich.edu)

GNU Make is a utility to *build* complicated things, like programs or LaTeXdocuments. It is designed to handle goals that have a clear hierarchy of tasks, i.e. each step has *prerequisites* that must be completed before advancing. Make is especially useful when developing code because at each step, it determines if all prerequisites need to be re-built or if some are *up-to-date*. To use make, a user must first write a *Makefile*, which must be called `Makefile`. Then, the user simply invokes Make by typing `make` at the command prompt.



```
# Simple makefile for "heat.exe".

default: heat

# Variables:
FLAGS = -fdefault-real-8
OPT   = -O3
DFLAG =
COMPILER = gfortran $(FLAGS) $(DFLAG) $(OPT)
DEPEND = \
        ModIO.o      \
        ModFwdDiff.o \
        HeatEq.o

debug:
        make clean
        make DFLAG='-g -fbacktrace -fbounds-check' OPT=-O0

heat: $(DEPEND)
        $(COMPILER) $^ -o $@.exe

%.o: %.f90
        $(COMPILER) -c  $<

clean:
        rm -f *.o *.exe *.mod *~ results*.txt

viz: heat
        ./$^.exe
        ./viz_results.py
```

Rules: The most basic part of a Makefile is a *rule*. This is a block of text that specifies a goal, or *target*, any *prerequisites* required to build the target, and what *commands* must be executed to create the target. The first target in the file becomes the *default* and is the one that is made by typing `make` with no arguments. Other targets can be selected by using arguments. For example, typing `make clean` will build the target “clean” instead of “default” in the above example.

Targets: *Targets* are the end goal of each individual rule. Typically, a target is the name of a file that will be created once the rule’s commands are executed, but *phony targets*, targets that do not refer to explicit files, are also acceptable. When a rule is executed, Make investigates to see if the target is *up-to-date*. This is true if the file the target refers to exists, is more recent than any prerequisites that are files, and all of the prerequisites that are also targets are up-to-date. This checking system is how Make minimizes the work performed when building targets. Note that phony targets are always considered out-of-date.

There are many “standard” phony targets, i.e. targets that users of your Makefile will expect to exist. The most important of these is `clean`. It is standard that `make clean` will remove all intermediate files and return the status of the build to its original state. You should always include `clean` in your file and give it meaningful clean-up tasks to perform.

Prerequisites: *Prerequisites* (alternatively, *dependencies*) can be explicitly listed files or other targets. If it is a file that does not exist, Make will search for a target that has the same file name or matches a *pattern rule* and execute that rule. All prereqs of a target must be up-to-date before the target can be built.

Commands: *Commands* follow the `target:prereq` syntax in a Makefile. They must be tabbed to the right of the target declaration. Commands are just that: any command that you would explicitly type at your terminal's prompt. Be sure to use *line continuation characters* if a single command wraps over more than one line. Make commands are most powerful when combined with regular and automatic *variables.*

Variables: *Variables* are key to Make's flexibility. They are declared very simply: `VAR = value`; it is standard to capitalize variable names. Their values can be any string of characters, there are very few limits. To use the value of a variable (known as *dereferencing* a variable), the syntax `$(VAR)` is used. It is possible to use curly braces instead of parentheses based on preference. When a variable is dereferenced, the value of variable takes the place of the dereferencing syntax. In our example, the value of `DEPEND` is a list of object files (`*.o`-files), split over several lines using the *line continuation character* for clarity. This variable is used later, as a prerequisite to the `heat` target. Because it is dereferenced there, the list of files is placed in its stead and each of those files is understood by Make to be a prerequisite to that target. It is proper to use variables to organize your makefile! Variables can be set at the command line as follows: `make DEPEND=otherfile.o`. This would override the value of `DEPEND` for the current execution of Make. Many Makefiles will require you to set variables at the command line to customize the build. Always read and include documentation!

Automatic Variables: Make has a set of special variables called *automatic variables.* Here are the most important ones:

Variable	Meaning
`$@`	The filename representing the target.
`$<`	The filename of the first prerequisite.
`$?`	A space separated list of all prerequisites newer than the target.
`$^`	A space separated list of all prerequisites, no duplicates.
`$*`	The target filename without its suffix (the stem).

Pattern Rules: It quickly becomes apparent that many different prerequisites will have nearly identical rules. *Pattern rules* (also known as *implicit rules* allow users to combine multiple rules together. In the example, the rule `%.o` applies to any file that ends in `.o` that does not have its own *explicit rule.* The `%` symbol acts like a Linux wildcard (*). The dependency for this rule is `%.f90`, which means the same file name as the target, but `.f90` instead of `.o`. Pattern rules make large builds with thousands of source files completely feasible with Make. Note that there are some built-in pattern rules, most notably `%.o: %.c`.

What does *this* Makefile do? The Makefile diagrammed above is designed to compile the source file `HeatEq.f90`, which depends on `ModFwdDiff.f90` and `ModIO.f90`. This code models the heat equation and prints output to an ASCII file. The output can be visualized with a Python script, `viz_results.py`. All files are assumed to be in the same directory as the Makefile.

When called for the first time with no targets and no variables (i.e., simply "`make`"), Make looks for the first target, which is `default`. It is a *phony target* with the *prerequisite* `heat`. There is no file called `heat`, so Make looks for a rule for this prereq and finds the target with the same name. The prereqs for target `heat` are given by the *dereferenced variable* `DEPEND`, which is a list of object files (files ending in `.o`). Make looks for these files in order, and, of course, will not find them. It also finds no *explicit rules* for any of the files. However, there is an *implicit pattern rule* for files that end in `.o` whose prereq is the same file that ends in `.f90`. This rule's commands are now executed once for each `*.o` file required by target `heat` in the order they are listed. The *variable* `$(COMPILER)` *dereferences* into `gfortran $(FLAGS) $(DFLAG) $(OPT)` which further dereferences into `gfortran -fdefault-real-8   -O3`. The *automatic variable* `$<` dereferences into the FORTRAN filename corresponding to the current target. By dereferencing all variables, it becomes clear that this rule is compiling each FORTRAN source file into individual object files.

Once all of the `heat` target's prereqs are ready, the `heat` rule's single command will be executed. `$(COMPILER)` dereferences the same as above, `$^` dereferences into a list of prerequisites as set by the variable `DEPEND`, and `$@.exe` becomes the name of our final executable, `heat.exe`. With the final program compiled and linked to its dependencies, Make is finished. Here are the set of commands explicitly executed:

```
gfortran -fdefault-real-8  -O3 -c  ModIO.f90
gfortran -fdefault-real-8  -O3 -c  ModFwdDiff.f90
gfortran -fdefault-real-8  -O3 -c  HeatEq.f90
gfortran -fdefault-real-8  -O3 ModIO.o ModFwdDiff.o HeatEq.o -o heat.exe
```

There are additional targets that add more functionality. If the user calls `make debug`, the phony target `debug` is called instead of the default. This target has no prereqs, so the commands are executed without detour. First, make calls itself (yes, you can do this!) with the phony target `clean`. This phony target merely removes all intermediate files produced by a single call to Make. Then, Make calls itself again, but this time two variables are set to non-default values. `DFLAG` is set from nothing to a list of compiler debug flags, and `OPT` is set to the no-optimization flag. The target `heat` is then remade using the new variable values. The commands executed are as follows:

```
rm -f *.o *.exe *.mod *~ results*.txt
make DFLAG='-g -fbacktrace -fbounds-check' OPT=-O0
gfortran -fdefault-real-8 -g -fbacktrace -fbounds-check -O0 -c  ModIO.f90
gfortran -fdefault-real-8 -g -fbacktrace -fbounds-check -O0 -c  ModFwdDiff.f90
gfortran -fdefault-real-8 -g -fbacktrace -fbounds-check -O0 -c  HeatEq.f90
gfortran -fdefault-real-8 -g -fbacktrace -fbounds-check -O0 \
    ModIO.o ModFwdDiff.o HeatEq.o -o heat.exe
```

Finally, there is the target `viz`. This target has `heat` as a prereq, so the code is compiled with default variable settings first. Then, `./$^.exe` dereferences to `heat.exe`, so the first command for rule `viz` is to execute the code that was just compiled. The next line executes the Python script, visualizing the results from the simulation. Within a single Makefile, a lot of functionality can be stored.

This guide is a very simple introduction. The material was taken from *Managing Projects from GNU Make, 3rd Ed.* by Robert Mecklenburg. There is much more to Make; users are encouraged to continue to explore this powerful utility.