Dependencies

`make` decides that certain **target** files need to be re-built when their **dependencies** have changed.

`make` will consider dependencies recursively.

```
wltest: wltest.o wordlist.o

wltest.o: wltest.c wordlist.h

wordlist.o: wordlist.c wordlist.h
```

The command

```
make wltest
```

will attempt to bring `wltest` up to date.

It will first make certain that `wltest.o` and `wordlist.o` are up to date.

A non-existent file always needs to be brought up to date.

A file with no dependencies, such as `wordlist.h`, is always up to date.

The default target is the first target in the `Makefile`.

Actions

`make` will use **actions** to re-build targets when they are not up to date.

An action should produce a fresh version of the target file.

```
wltest: wltest.o wordlist.o
        gcc -Wall wltest.o wordlist.o -o wltest

wltest.o: wltest.c wordlist.h
        gcc -Wall -c wltest.c

wordlist.o: wordlist.c wordlist.h
        gcc -Wall -c wordlist.c
```

The command `make wltest` will bring `wltest.o` and `wordlist.o` up to date, and then use the linker command to link them.

The command `make wltest.o` will bring `wltest.c` and `wordlist.h` up to date, and then use the compiler command to compile it.

The command `make wordlist.h` will do nothing unless `wordlist.h` doesn't exist, which is an error.

False Targets

False targets can be used to allow a single make invocation to produce several different targets.

```
all: client server
```

The command

```
make all
```

will bring both `client` and `server` up to date.

False targets can also be used to provide an action that will always be taken.

```
clean:
        rm -f client server client.o server.o \
        common.o util.o
```

The command

```
make clean
```

will always execute the action

```
rm -f client server client.o server.o common.o util.o
```

Variables

`make` provides **variables** which can be assigned a string.

An undefined variable is considered to contain an empty string.

```
OBJS=util.o prog.o foo.o bar.o quux.o

all: prog

clean:
        rm -f $(OBJS) $(STUFF) prog

prog: $(OBJS)
        gcc -Wall $(OBJS) -o prog
```

The variable `OBJS` will contain `util.o prog.o foo.o bar.o quux.o`. The variable `STUFF` is undefined and hence contains the empty string.

Substituting as indicated above, we obtain,

```
all: prog

clean:
        rm -f util.o prog.o foo.o bar.o quux.o prog

prog: $(OBJS)
        gcc -Wall util.o prog.o foo.o bar.o quux.o -o prog
```

Default Rules

`make` provides certain **default rules** that provide default dependencies and actions for certain common patterns.

These default rules are made more flexible by the use of variables such as `CC` to hold the name of the C compiler.

Often, for some arbitrary *foo*, the file *foo*`.o` is compiled from *foo*`.c`.

```
foo.o: foo.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) foo.c
```

Similarly, a program *foo* is often built from *foo*`.o` and some other objects such as *bar*`.o`, *baz*`.o`, and *quux*`.o`.

```
foo: foo.o bar.o baz.o quux.o
        $(CC) $(LDFLAGS) foo.o bar.o baz.o \
        quux.o $(LOADLIBES) -o foo
```

`Makefile` example

```
CC=gcc
CFLAGS=-Wall -g
CPPFLAGS=

LDFLAGS=-Wall -g

OBJS=wordlist.o wltest.o wl.o

PROGS=wl wltest

all: $(PROGS)

clean:
        rm -f $(OBJS) $(PROGS)

wordlist.o: wordlist.h

wltest.o: wordlist.h

wl: wordlist.o wl.o

wltest: wordlist.o wltest.o
```