Advanced Unix Tools	**Homework 3**
CS214 Spring 2004	**Due Date: Friday, March 19, 2004**

In previous versions of CS214, we used to study a tool called `make`, which is used mainly to manage softward projects (but can be used for other stuff too). Rather than study `make` in class, we will reproduce the essence of its functionality in this homework.

First, let's learn very quickly about `make`.

Overview of make

`Make` is a tool that is used to manage the construction of files from other files. The most common example of this phenomenon is the process of compilation for programming languages: you have a set of source files, you want to compile each of them into object files, and finally you want to link those object files together, possibly with some library files, into an executable program.

`Make` reads in a file containing a description of the dependencies between the files, and commands describing how to construct target files (i.e., object files) from dependent files (i.e., source files). The `make` tool then automatically figures out the best way to construct the new files. Among other things, if some of the target files already exist and are still valid (that is, if there has been no modification to the dependent files), then these target files don't need to be rebuilt.

The files that `make` reads and that contain the dependencies between files are called makefiles, and one typically stores the under the name `makefile`. Here is a sample makefile:

```
myapp : file1.o file2.o
      gcc -o myapp file1.o file2.o

file1.o : file1.c macros.h
      gcc -c file1.c

file2.o : file2.c macros.h
      gcc -c file2.c
```

This makefile describes the dependencies requires to compile the C source files `file1.c`, `file2.c` (and header file `macros.h`) into the executable called `myapp`. This makefile contains three rules, each rule describing a particular dependency. A rule is of the form:

target : *dep dep dep*
      cmd
      ⋮
      cmd

Such a rule states that to check if you need to rebuild the file *target*, you first check (recursively!) if any of the dependency files *dep* need to be rebuild. Once that is done, you can decide if *target* needs to be rebuild: this is the case either if *target* does not exist, or if it is older than any of the dependency files *dep*. (Here, older is taken to with respect to the time of last modification, not the time of creation.) If this indicates that *target* needs to be rebuilt, the commands specified after the rule are executed.

Consider the previous example. It states that you need to rebuild the file `myapp` if either it does not exist or if it's older than either `file1.o` or `file2.o`. To rebuild `myapp`, you invoke `gcc`, the C compiler, with the appropriate flags. Similarly for rebuilding `file1.o` and `file2.o` from `file1.c` and `file2.c`.

To "execute" such a makefile, you simply call `make` in the directory of the makefile. If the makefile is indeed called `makefile` (or `Makefile`), `make` will automatically find it and figure out if there are any files that need to be rebuild and if so rebuilds them. If the makefile is stored under another name, say *mfname*, you can still use `make`, with an option: `make -f` *makefilename*.

The Homework

In this homework, you will be reproducing the functionality of `make`. Well, not in all its full glory. Nor in most of the ways in which it solves problems. But we will implement the "essence" of what `make` does.

I want you to write a Perl script called `build` that takes a single argument, the name of a dependency file (to be described shortly), say *foo*, and that creates a *new script* named `build-`*foo* (where of course, *foo* is replaced by whatever name you gave `build` in the first place). This new script, when executed, should execute the commands specified in the dependency file in the appropriate order to rebuild everything that needs to be rebuilt. More on this later. The script that `build` creates can be a perl script or a shell script, whichever you prefer. Just make sure that:

1. you setup the right header `#!` ... as the first line;
2. you set the permissions correctly.

A dependency file is a file containing a sequence of *dependencies*. A dependency is written as follows:

```
build
  targetfile
ifchanged
  file1
  file2
  ⋮
```

```
do
  command1
  command2
  ⋮
end
```

Intuitively, such a dependency says that the file *targetfile* should be rebuilt if it does not exist, or if it is older than any of *file1*, *file2*, ... What do I mean by older? If the time of the last modification is less than the time of last modification of any of *file1*,*file2*,.... Intuitively, this means that one of the files on which *targetfile* depends has been modified, but *targetfile* does not reflect that change. (For example, a source file has been changed, but the resulting application has not been recompiled.) The dependency indicates how to rebuild *targetfile* if indeed it needs to be rebuilt: execute *command1*, *command2*, ..., in that order.

As I said, a dependency file is just a file containing a number of these dependencies. Of course, things get interesting if a file depends on a file which itself depend on antoher file, and so on. (Potential problems arise if there is looping in the dependencies, as you can imagine!)

To be more precise, the script `build` you have to create should take as input the name of a dependency file containing dependencies as above, and produce a new script that will do the following:

1. For every dependency listed in the dependency file, it checks if the *targetfile* does not exist or is older than any of its dependencies, and if so, executes the commands specified to rebuild the *targetfile*. After the commands have been executed, the script should check that *targetfile* has actually been updated, and report an error if not.

2. Note that a dependency file can change during script execution due to another dependency rebuilding it (and thus changing its modification time). Make sure you deal with that correctly. (See the following example for a simple test case.)

3. After your script finishes executing, it should be the case that *all* the files specified in the dependency file satisfy their dependency requirement.

Consider a directory containing three files `a`, `b`, `c`, each containing a single line, respectively, `"this line is from a"`, `"this line is from b"`, `"this line is from c"`, and the dependency file `test`, with content:

```
build
  final-output
ifchanged
  intermediary-output
  c
do
  cat c intermediary-output > final-output
```

```
end

build
  intermediary-output
ifchanged
  a
  b
do
  cat b a > intermediary-output
end
```

This is the behaviour I expect:

- If you call `build test`, your script `build` should create a new script `build-test` in the current directory. If you call this newly created script `build-test` (with no arguments), it should attempt to satisfy the dependencies specified in the original `test` dependency file. That is, it should build `intermediary-ouput` (since it doesn't exist), which will contain two lines (the one from `b`, followed by the one from `a`), and then build `final-output` (since it doesn't exist), which will contain the line from `c` followed by the content of `intermediary-output`, that is, the line from `b` followed by the line from `c`.
- If you execute `build-test` a second time, it should do nothing, since `final-output` and `intermediary-output` already satisfy all their dependencies.
- If you then edit file `a` to put in something like `this is some new line from a...`, and you execute `build-test` again (without calling `build`, of course), then `intermediary-output` should be rebuilt, since it does not satisfy its dependency requirement anymore (it is older than `a`). This means that `final-output` also will not satisfy its dependency after `intermediary-output` is rebuilt, and thus itself should be rebuilt. After that process, `final-ouput` should contain the three lines `"this line is from c"`, `"this line is from b"`, and `"this is some new line from a..."`.

Some remarks:

1. There are a number of ways of implementing the above, from topologically sorting the files according to the dependency order, to simply looping through the dependencies until they are all satisfied. It's okay if your solution is not efficient.
2. You have to somehow read in the dependency file and "process" that input. Essentially, you want to parse the dependency file and extract information from it. To make your life simple, you can assume that the syntax is such that keywords are all on their own line, and that there is no more than one file name or one command per script. There are potential problems when a file or command is named the same thing as a keyword (such as 'build', or 'ifchanged',

etc.) To solve this problem, assume the syntax is such that keywords need to start at the beginning of a line, while a command or filename never starts at the beginning of a line.

3. Do not worry about cyclic dependencies; it's fine if your script enters an infinite loop if it hits a cyclic dependency.[1]

4. The Perl function `lstat` will probably come in handy for determining the time of last modification.

5. The output of `build` should be an executable script, that means that you should be able to actually *execute* whatever is output by `build`...

To think about. If you have some time to kill, try to figure out how you could add variables to the above. Intuitively, I would like to be able to define and refer to variables in `build` dependency files. These variables should initially have the same value as any same-named environment variables, but can be redefined in the dependecy file via a line:

```
variable name = value
```

You can use the notation `$(name)` to refer to the value of a variable.

[1] Although, if it is easy for you to catch that case, and you do catch it, you'll make the grader happy, and a happy grader is a generous grader.