Questions

Here are some detailed questions about *Make – A Program for Maintaining Computer Programs* by Stuart I. Feldman who *invented* Make. Therefore, this paper was the first paper ever about Make.

1. Easy questions; you should be able to answer these within 5 minutes.
   (a) What is the problem being solved by Make?
   (b) A beginner is tempted to write the following code in order to compile a file `foo.c` that includes `foo.h`:

   ```
   foo.o:  foo.c
       cc -o foo.o foo.c
   foo.c:  foo.h
   ```

   This is wrong—why?
   (c) How does Make detect that a file was changed?
   (d) Does Make check that a rule does what it says? That is, can you lie to Make?
2. List Make's key concepts. Just identify them, you don't need long explanations. For example, one key concept are rules, which make Makefiles declarative (opposed to what?). Find more such concepts.
3. What abstraction mechanisms does Make provide? How can you make a Makefile shorter? (Compare these with abstraction mechanisms in programming languages.)
4. Does a Make know all the files that it needs to check from the Makefile (closed universe), or does it have to discover what files are out there that might be relevant (open universe)? What are the implications of this design?
5. Does Make analyze a situation completely before it issues any build actions, or are analysis and building interleaved? Can you write a Makefile for an experiment to answer this question? (If you have a laptop around, try this!.) What are the consequences?
6. After Make many tools where designed to replace it, but none became as universally accepted as Make. Why do you think Make is such a success?
7. What are you missing in Make? You don't need to agree on this. Just prepare a list of reasons for additional features.
8. As a paper, do you find *Make – A Program for Maintaining Computer Programs* convincing?

Advanced Questions

You might want to think about the following questions; however, we are not going to discuss them in class.

1. Some Make replacements implement dependency scanning: the tool scans source files to determine what other files they depend on. Typically, this works for C and C++. The main argument is, that manually specified dependencies are often wrong and result in hard to find bugs. How do you like this idea?

2. Are rules first-class? That is, can you store rules in variables or construct them using variables?

3. What is the role of file-name suffixes in Make? Suppose you have two sets of C files and you want to use two different compiler commands for the files in each set. Can you use generic rules for this?

4. Can you express in Make that a command creates a file only in some cases, but not in others? For example, how would you express the effect of this shell-command sequence?

   ```
   gzip -c foo.ml > foo.cmo
   if [ -f foo.cmi ]; then
       cat foo.cmi > /dev/null; else touch foo.cmi
   fi
   ```

   The command sequence simulates the behavior of the Objective Caml ML compiler, compiling a file `foo.ml` to an object file `foo.cmo`. If an interface file `foo.cmi` exists, it is checked against the implementation; otherwise an interface file `foo.cmi` is derived from the implementation `foo.ml`.

Small-group work

As a reminder, here is our procedure to tackle questions during our meetings.

1. Divide into groups. We will try to form groups that are diverse, so that each group contains people with different degree of experience and research interests. Once the groups have been formed, introduce yourselves to each other.

2. Choose a recorder. Agree on one person to record the views expressed in your group, in particular

   - decisions the group makes collaboratively,
   - significant dissent.

3. Discuss questions; use the black board, a laptop, or whatever you deem necessary to structure your answers.