Program 3: Partial Order Planner
Due date: Tuesday, March 23.

The problem: A partial order planner works by systematically searching the space of partially ordered plans until it finds a plan that satisfies a particular goal.

Assignment: You will develop and test a partial order planner. It will use plan operator schemas as defined below. The use of variable constraints in operator schemas (see “Variables in Partially Ordered Plans” in Dean et al., section 7.2, for details) is optional, but it should be possible to ultimately recognize and deal with conflicts involving variables (e.g. At(?someplace) and not(At (home)) conflicts if ?someplace is bound to home). You may find it easier to not notice these conflicts until necessary.

A partial plan is defined by a structure with the following components:
1. Steps (the plan steps), identified by an identifier and its value, where a value is a plan operator schema.
2. A set of step ordering constraints: a set of pairs (Si, Sj) with the interpretation that Si precedes Sj.
3. A substitution (as with unification): variables and their bindings. This substitution is global over the partial plan.
4. A set of links (referred to as Causal Links by R&N); each link is a triple (C, Si, Sj), where condition C is in the effects of step Si and in the preconditions of step, Sj. We refer to this as *C is established in Sj by Si.* For each of these, there should also be a step ordering constraint (Si, Sj) in 2.
5. (optional) Conflicts. A set of pairs of the form (link, step), where step may be between the two steps in the link, and has the negation of the link’s condition.
6. (optional) Requirements. A set of pairs of the form (condition, step), where condition is a condition in the preconditions of step, and condition, step (respectively) are not the first and third element of a link; that is, this condition of this step is not supported by a link in the plan.
7. (optional) Binding constraints of the form (not (equal <var> <term>)) where <var> is a variable and <term> is a variable or constant. These restrict what substitutions should be allowed by unify, and correspond to posted constraints as used in Dean et al, as mentioned above.

Options 5. and 6. will need to be computed if not explicitly in the plan representation; 7. is only needed if you use binding constraints as mentioned above.

Plan operator schemas: Each plan operator schema will have the following parts:
1. Description: a descriptive statement, possibly with variables (e.g. (move ?thing ?place)) for an action corresponding to moving ?thing to ?place).
2. Preconditions: a set of non-negated literals, possibly including variables.
3. Postconditions: a set of literals, possibly including variables. Optionally this can be expressed as two separate sets: those added (non-negated), and those deleted (negated literals).
4. (Optional): Constraints on variable bindings (if using binding constraints, as in Figure 7.14 in Dean, et al.).

**Operation:** You should write a routine `plan(initial, goal, operators)`, where

- `initial` is a set of non-negated variable-free literals corresponding to facts that are true before taking the first step.

- `goal` is a set of non-negated literals corresponding to facts that should be true after the plan is executed.

- `operators` is a structure containing all of the operator schemas that can be used in this plan.

`plan` returns a partial plan where there are no conflicts, and each element of each precondition is established by another step (i.e. in a link).

**Assignment specifics:** For this to work, `plan` needs to produce an initial partial plan with two steps (Start, with postconditions equal to initial; End, with preconditions equal to goal, with ordering constraint (Start, End)). Your program will then need to search systematically from there until it finds a solution.

Some of your code from before should be able to be reused. The applicability of an operator schema can be detected by unifying one of its postconditions with a requirement. You will need to carry along a substitution list with each partial plan. You will want to rename variables whenever you use an operator schema.

Read the material in the book (either one) carefully to get a feel for what you are trying to do. You should pay special attention to the notions of *sound*, *complete*, and *systematic*. Dean et al. propose a best-first search approach, with best being the one with the fewest requirements (unestablished preconditions); this is certainly a case where (blind) depth-first search is *not* an option. In this project it is very important that you partition the problem into modules that are small enough to build and test independently, as the whole process is rather complex.

Early on you should make it simple to 1) read in problems, and 2) observe the behavior of your program in a way to support debugging.

This program can be written in a (small) group if you prefer. If so, you should spend some time on decomposing the task, determining any common representations, and defining the interfaces between the modules.
Report due:
- Specification of the textual syntax of input and output.
- Documented source code, so it could be compiled and run by someone else (hard copy plus electronic version, zipped and submitted to robert via email, named Project3_300_<your name>. Documentation should identify all classes or types defined and used.
- Test results (include test files in zipped stuff, plus enough hard copy to illustrate).
- Short discussion of design, including choices made and alternatives, plus a critique of the design (e.g. how could it be improved). Specifically, did you use “not equal” constraints; if so, how did they work, if not, what did you do instead?
- “Comparable” results using common problems (see below).

For your extra excitement:

One way to evaluate these programs is to compare their behavior on common problems. Each person should provide at least one planning problem (that is, set of operators, initial, and goal) that they think is a good test. Around the due date we will choose a number of these that all planners should run. Evaluation will be based on 1) whether the planner succeeds, 2) how many refinements were generated on the way to the solution, and 3) how much processor time was required to solve the problem (so your program will need to be able to collect statistics for 2 and 3).