CSCE 625
Programing Assignment #9
due: Fri, May 8 (by noon)

Goal Regression

The goal of this assignment is to implement a simplified version of the goal regression algorithm for doing planning in the Blocksworld. 'Simplified' means that we are going to do this with ground versions of the pickup() and puton() operators (where variables have been replaced by all combinations of blocks names), to avoid having to do unification.

Overview of the program

The main program will take two command-line arguments, a file of operators and a file of facts about the initial state. The main program will be interactive, such that it gives you a command prompt, and you can type goals into it. When given a goal (or list of goals), it will try to find a plan to achieve the goal from the initial state. If it succeeds, it will print the plan as a sequence of actions that achieves the goal. Here is a working example. The initial state looks like this:

```
    a   c
    b   d
  -------------
```

Suppose we wanted to find a plan for on(a,c). Running the program would look like this, ultimately producing a 2-step plan, pickup(a,b) followed by puton(a,c):

```
287 compute-linux1> python goal_regression2.py blocksworld.opers init.kb
> on(a,c)
initiating goal regression...

iter=1, queue=1
 context:
 goal stack: on(a,c)
 considering using puton(a,c) to achieve on(a,c)

iter=2, queue=1
 context: puton(a,c)
 goal stack: holding(a) clear(c)
 considering using pickup(a,b) to achieve holding(a)
 considering using pickup(a,c) to achieve holding(a)
 considering using pickup(a,d) to achieve holding(a)
 considering using pickup(a,table) to achieve holding(a)
 considering using pickup(a,c) to achieve clear(c)

iter=3, queue=4
solution found!
plan:
```
```
puton(a,c)
pickup(a,b)
```

```
> quit
```

File Formats

The initial state can be described in a kb file (init.kb) that looks like this:

```
clear(a)
on(a,b)
on(b,table)
clear(c)
on(c,d)
on(d,table)
gripper_empty()
```

Note the syntax. Each fact (or predicate) is represented by a symbol, followed by a comma-separated list of args in parentheses. Blank lines and lines starting with '#' will be assumed to be comments and ignored. You will have to write a simple parser to read in facts and store them in a suitable data structure (perhaps as class that consists of a list of strings...).

For operators (file blocksworld.opers), we will use the following format:

```
OPER pickup(a,b)
precond: clear(a) on(a,b) gripper_empty()
addlist: holding(a) clear(b)
dellist: clear(a) on(a,b) gripper_empty()
conflict: on(b,a) on(a,b) on(c,a) on(a,c) on(d,a) on(a,d) holding(b) +
 holding(c) holding(d)
END
```

(Note that the 'conflict:' line is supposed to be all on one line but is broken into 2 lines by '+' in this document.) The first word in each line is required, followed by a space-separated list of facts. You will have to write a parser to read in the file of operators. You will probably want to make a class that stores the operator name, and lists of pre-conditions, add list, and delete list. Note that all of these like like predicates, so you can use the same class as for facts above.

The semantics of these fields is like discussed in class. 'precond:' is the list of pre-conditions, 'addlist:' is the set of effects that will become true as a consequence of the action, and 'dellist:' is the set of things that will become false (typically the pre-conditions). '**conflict:**' is a new field which is needed for goal regression (as you will see below). It gives a list of facts that would be inconsistent with the outcome of the action. For example, in pickup(a,b), the usual delete list would be {clear(a), on(a,b), gripper_empty()}, since these are pre-conditions that would get negated directly by the action. However, if we are doing goal-regression and another goal on the subgoal list happens to be on(a,c) or holding(d), then we need to be able to detect this inconsistency. Thus, you will have to use 'conflict:' to list all the additional facts (mainly involving on() and holding() of other blocks) that would be inconsistent with the outcome of the action. (Sorry for this complication. Enforcing these constraints, like holding(X)→¬on(X,Y)∧

¬on(Y,X) would be easier to do with variables, but then we would need to unification and inference; hence the need for making an explicit list of conflicts.)

Generating Ground Operators

In a typical application of goal regression, you would have one generalized operator for pickup(X,Y) and one for puton(X,Y). However, because I want to avoid doing unification for variables, we will just generate all the instances of operators from these schemas by filling in all combinations of blocks. So you will write a pre-processor program that generates operator definitions for pickup(a,b), pickup(a,c), pickup(a,d), pickup(b,a), pickup(b,c)...puton(a,b), puton(b,a), puton(a,c), puton(c,a)... This can be achieved by writing a simple script to enumerate over all combinations of blocks (assume they are limited to: a,b,c,d). Also, you will have to include special version of the operators for pickup and puton of objects to the table, e.g. pickup(a,table), puton(a,table), etc. These have slightly different pre-conditions and effects, because we assume the table is always clear.

Goal Regression Algorithm

Effectively, goal regression performs a backwards search in the state space from the goal to the initial state. This is enabled by the regress() function, which takes an initial list of goals and regresses them through an operator to form the set of weakest preconditions. There are many ways to control the order of nodes searched during goal regression. Similar to other projects you did earlier this semester, we are going to use a queue-based mechanism. The queue will maintain a list of <subgoals,plan_suffix>, where subgoals are conjunction of facts such that if they were true, then the utlimate goal could be achieved by executing the remaining steps of the plan_suffix (i.e. rest of the plan). We initialize the queue with the given goal list and empty plan: <goals,∅>. We keep the queue sorted on the length of plan_suffix, which will effectively produce a breadth-first search from the goal backwards.

With each iteration, we pull the next node out of the queue. We test the list of goals to see if they are all satisfied by the initial state; if they are, then we return the plan_suffix. Otherwise, for each goal that is unsatisfied, we go through all the operators that could achieve this goal as an effect (on the addlist), and regress the goal list through the operator. Remember that the regress(goals,oper) function consists of removing any goals that are on addlist (effects of the action) and then appending the pre-conditions. (This step is simplified by using ground predicates, and would be considerably harder if variables are involved, because you would have to use unification.) For each of the applicable operators, we append them to the current plan_suffix and add them to the queue, along with the updated (regressed) goal list. Here is some pseudo-code:

```
goal_regression(goals,opers,kb)
  queue = new Queue // use a priority queue, or just keep sorted on plan len
  queue.insert(<goals,[]>) // could make a Node class for this
  visited = new HashTable()

  while True:
    <goals,plan> ← queue.pop()
    if all goals are satisfied by kb, return plan
    for each goal in goals:
      for each oper in opers:
        if goals ∩ oper.addlist ≠ ∅: // oper relevant; effect achieves a goal
          if goals ∩ (oper.dellist ∪ oper.conflicts) = ∅: // is consistent
            newgoals ← regress(goals,oper)
            newplan = plan.append(oper)
            ps ← planstring(newplan)
            if ps not in visited:
              visited[ps] = 1 // hash it
              queue.insert(<newgoals,newplan>) // keep sorted on plan len

regress(goals,oper)
  return (goals \ oper.addlist)  ∪ oper.preconds // '\' is set-difference
```

(You will probably want to add some print statements each iteration to show things like the popped goal list, plan_suffix, iteration count, and queue size for tracing/debugging purposes.)

There are two important subtleties here. First, note that before we select an operator as a way of achieving a goal, we check to see that it is *consistent* with the rest of the goals. This is where the 'conflict:' list comes in (along with predicates on the dellist). Even if an operator has a member of the addlist that can achieve one of the subgoals, it is not used if there is another subgoal that appears on the conflict list.

Second, in order to keep the queue size manageable, it is helpful to check for and discard visited states. There are several ways to do this, but one suggestion is to make a "print string" out of each plan_suffix and put it in a hash table, so you can quickly look up whether you have generated it before, and if so, avoid putting a second copy in the queue.

What to Turnin

1. Submit your **source code** via the Turnin program on CSNet.
2. Submit your generated file of operators, and the script used to generate it.
3. Also include a **transcript** showing your solutions to the following problems:

on(a,c)
on(b,a)
on(a,c) holding(b)
on(a,d)
on(a,d) on(d,b)