cps721: Assignment 5 (100 points).
Due date: Electronic file - *Monday evening*, November 25, 2019, 21:00 (sharp).

YOU MAY NOT USE ";" AND "!" AND "–>" IN YOUR PROLOG RULES

You must work in groups of TWO, or THREE. You can discuss this assignment only with your CPS721 group partners or with the CPS721 instructor. By submitting this assignment you acknowledge that you read and understood the course Policy on Collaboration in homework assignments stated in the CPS721 course management form.

This assignment will exercise what you have learned about problem solving and planning using Prolog. More specifically, in this assignment, you are asked to use the situations and fluents approach to reasoning about effects of actions.

1 (20 points). This question is related to the previous assignment about natural language processing, and you are not asked to solve any planning tasks in this question. However, you need to use the situations and fluents approach considered in class to implement a modification of your program from the previous assignment. Previously, we could formulate queries with noun phrases about a static scene only, but now we imagine a robot that can move blocks around. We would like to query in English after moving a few blocks in the initial scene. More specifically, the task is to handle simple imperative sentences of the form "Put NP on NP" where NP is a noun phrase. For example, the following are syntactically well-formed imperative sentences:

Put a green pyramid on the small red cube.
Put an orange wedge on the block beside a red cube.
Put any small block on a green cube on a big red cube.

We model sentences such as these using the predicate do([put | List],S_1,S_2), where List is the list of words from your lexicon representing a phrase of the form "NP1 on NP2", S_1 is the current situation and S_2 is the situation resulting from moving an object described by NP1 on top of another object described by NP2 (if this is possible). We also would like to answer queries that use English noun phrases, but now all of them will have an additional argument S (*situation*) to represent changes in the scene resulting from blocks being moved from one location to another.

Make the following modifications in your program **nlu.pl** that you developed in the previous assignment.

1. In your database, write *precondition axioms* for the action putOn(*Block*,*X*), where X can be either a cube or an unoccupied area on a table. This action is impossible if there is another block on the top of *Block* or on the top of *X*; in other words, if *Block* or *X* are not clear. In addition, this action is impossible if *X* is a block, but not a cube, or if *Block* is already on *X*. The rules that you write must use the fluent locatedOn(*Block*,*Loc*,*S*) which is similar to the predicate from the previous assignment, but it has an additional argument S. Your rules may also use other predicates from your database. If you like, you can also introduce another fluent clear(X,S) that is true if there is nothing on top of X (a block or an area) in a situation S.
   Next, write *successor state axioms* for the fluent(s).

2. Add a situational argument S to the predicates beside(X, Y), above(X, Y), leftOf(X, Y) rightOf(X, Y) everywhere in your database. Add [ ]-argument to all atomic statements locatedOn(*Block*,*Loc*) in your database: they represent your scene in the initial situation. If you use the predicate clear(X,S), then you need also to write which blocks and areas on the table are unoccupied in the initial situation. All your rules in the database must use the fluent locatedOn(*Block*,*Loc*,*S*) instead of the two-argument predicate locatedOn(*Block*,*Loc*) that you had before. Because the action putOn(*Block*,*X*) can have effect only on locations of blocks, all remaining predicates in the database remain without changes.

3. Make changes in your lexicon (in the predicate preposition only) and in your parser. More specifically, you will need a rule that implements the predicate do([put | List],S_1,S_2) as well as changes in other parts of your parser (grammar) that will allow you to parse queries like

   ```
   ?- what( [any,yellow,pyramid,on,a,big,cube], B, []).
   ?- what( [the,pink,wedge,above,a,block,beside,a,medium,blue,cube], B, [putOn(b1, b8)]).
   ```

   where the last argument represents an initial situation or the situation resulting from doing a sequence of actions. Keep your modified program in the file **put.pl**

4. Test your do(Command,S_1,S_2) predicate on a variety of imperative sentences like those above, including some that are syntactically well-formed but impossible to execute, showing that "do" is capable of computing whether performing an action is possible. (If a command cannot be executed then do(Command,S_1,S_2) returns "no".) Test the "what" predicate on a variety of noun phrases, showing that "what" is capable of identifying the blocks being referred to in your scene after performing possible actions. It is up to you to choose noun phrases for testing, but you must convincingly demonstrate that

your program works properly (try about 10 different noun phrases with words "above", "beside" about blocks that were moved to new locations). Copy all results of your tests into another file **put.txt** For example, you can consider queries similar to the following:

```
?- do([put, an, orange, wedge, on, the, medium, blue, cube], [],S),
        do([put,any,yellow,block,on,a,table],S,Snext).
?- do([put, an, orange, wedge, on, the, medium, blue, cube], [],S),
         what([a,wedge,above,a,blue,block],B,S).
```

Handing in solutions: An electronic copy of your file **put.pl** and a copy of **put.txt** must be included in your **zip** archive. The file **put.txt** must include a copy of session(s) with Prolog, showing the queries you submitted and the answers returned.

For the question 2, download a file containing the rules for `solve_problem`, `reachable`, `max_length(List,Bound)` as given in class: they are generic and can be used to solve any planning problem. Your task is to add your own precondition and successor state axioms using terms and predicates specified in the assignment. (Download the planner from D2L).

2 (80 points). This question asks you to implement a simplified airport traffic management system. In a nutshell, the problem consists of coordinating the movements of airplanes on the ground so that they reach their planned destinations (runway or parking position) as soon as possible whereby collisions shall be, of course, avoided. The airplanes move on the airport infrastructure, which consists of runways, taxi ways, and parking positions. Airplanes are generally divided into the three categories: light, medium, and heavy, which classify them according to their engine exhaust. We use the predicate $hasType(Airplane, Category)$ to say that $Airplane$ belongs to one of these categories. A moving airplane can either be in-bound or out-bound. In-bound airplanes are recently landed and are on their way from the runway to a parking position, usually a gate. Out-bound airplanes are ready for departure, meaning they are on their way to the departure runway. Since airplanes are not able to move backwards, they need to be pushed back from the gate on the taxiway where they start up their engines. For simplicity, we assume that an airplane always needs to be pushed back. The ground controller has to communicate to the airplanes which ways they shall take, where they can take off, when they can start its engines and when to park. The planner is given an initial configuration that consists of in-bound and/or out-bound airplanes and a goal configuration that specifies which airplane have to depart and which have to park. The task of a planner is to find a shortest sequence of actions that leads from the initial to the goal configuration. Of course, the planner has to compute this sequence of actions as quickly as possible. For simplicity, we assume that both planning and subsequent execution of a plan will take little time, so that no new airplanes arrive and no vehicles block traffic In other words, we assume there are no exogenous actions.

The airport infrastructure is built out of segments. An airplane always occupies one segment and may block several others depending on its type. Our assumption here is that medium and heavy airplanes block the segment behind them whereas light airplanes only block the segment they occupy. Blocked segments cannot be occupied by another airplane. To handle notions like "behind" we need to introduce direction in segments. Since our segments are directed edges represented by a term $seg(V_1, V_2)$ we identify directions with vertices. Every segment has 2 end vertices so it becomes possible to talk about *direction* in segment. Namely, the vertex V_1 represents the direction when V_1 is at front, the vertex V_2 represents the opposite direction when V_1 is behind. Thus, the "direction" of a segment is simply understood as the vertex of the segment.

We need several predicates to describe the airport infrastructure. These predicates specify static properties: truth values of each of the following predicates will not change after doing actions, i.e., they are set once in the initial state and never change.

The $canMove(Seg_1, Seg_2, Dir_1)$ predicate states that an airplane may move from segment Seg_1 to segment Seg_2 if it is facing direction Dir_1 on Seg_1. The $canPushback(Seg_1, Seg_2, Dir_1)$ predicate describes the possible backward movement which is similar to the "canMove" predicate. In our encoding, the "canMove" and "canPushback" predicates hold only for pairs of segments that belong to the standard routes on the airport – this is common practice in reality (as reroutes are likely to cause trouble or at least confusion).The closely related $moveDir(Seg_1, Seg_2, Dir_2)$ and $moveBackDir(Seg_1, Seg_2, Dir_2)$ predicates state the airplanes heading after moving from segment Seg_1 to segment Seg_2. For example, when an airplane at segment Seg_1 facing direction Dir_1 moves to segment Seg_2, its new direction Dir_2 is determined by $moveDir(Seg_1, Seg_2, Dir_2)$. That means that in a correct airport domain every "canMove" ("canPushback") predicate has its "moveDir" ("moveBackDir") counterpart: rules for them are given to you.

The $segIsBlocked(Seg, Type, Seg_1, Dir_1)$ predicate is used to handle all static safety distances due to engine exhaust. It says that segment Seg is blocked by an airplane of category $Type$ at segment Seg_1 facing into direction Dir_1. This predicate is static in a sense that it holds no matter whether any airplane is physically present at Seg_1, Dir_1 or not. In other words, it specifies purely geometric constraints between connected segments. This predicate imposes conditions when the "move" and "startup" actions can be executed: these actions are not possible if the airplane is going to segment that is blocked or if another airplane can be endangered. There is another closely related predicate that handles how blocking between segments changes when airplanes move around (see below).

The last predicate, $isStartRunway(Seg, Dir)$, states that segment Seg with direction Dir is the right location to takeoff. This is an essential predicate as we cannot allow an airplane to takeoff wherever it (or better: the planner) wants.

All rules implementing the airport infrastructure are given to you together with the facts about the initial and goal configurations. These rules use the predicates mentioned above. You can download them: see the file **airport.pl**

This application domain is represented using five action **terms**: $move(Airplane, Type, Dir1, Seg1, Seg2, Dir2)$, $pushback(Airplane,Type,Dir1,Seg1,Seg2,Dir2)$, $takeoff(Airplane,Seg,Dir)$, $park(Airplane,Type,Seg,Dir)$, $startup(Airplane)$. The meaning of these actions should be clear from their names, but precise conditions when these actions are possible are stated below. The changes in this domain are represented with the following seven **fluents** (fluents are predicates with situation as the last argument): $blocked(Seg,Airplane,S)$, $atSegment(Airplane,Seg,S)$, $facing(Airplane,Dir,S)$, $airborne(Airplane,Seg,S)$, $isParked(Airplane,Seg,S)$, $isMoving(Airplane,S)$, $isPushing(Airplane,S)$. Note that the actions and the fluents are general enough to deal with arbitrary configurations of airplanes in this domain. The last four fluents introduce an airplane state. An airplane can either be moving, be pushed, be parked (at segment), or be airborne. We want to make sure that an airplane only moves backwards while being pushed from its park position and only moves forward if not. The parked state is necessary since a parked airplanes engines are off, and therefore the parked airplane does not block any segments except the one it occupies, unlike when it is moving. If an airplane is airborne, i.e. it took off already, then that means this plane is not relevant to the ground traffic anymore.

- The fluent $blocked(Segment, Airplane, S)$ is the predicate that we use to characterize blocked airport segments in the current situation S. For example, when $Airplane$ moves from Seg_1 to Seg_2, then Seg_2 becomes blocked by this airplane in the resulting situation. The action "takeoff" has the opposite effect: airborne airplanes do not block any segments. All successor state axioms for this fluent are given to you.

- The fluent $isParked(Airplane, Seg, S)$ is the predicate with a second parameter representing the segment where the airplane is parked at. Airplanes remain parked unless they are pushed back.

- The second parameter of the $airborne(Airplane, Seg, S)$ predicate just states from which segment the airplane took off. This is useful in case an airport provides several departure runways and we want to force an airplane to use a specific one.

- The predicate $isMoving(Airplane, S)$ holds once $Airplane$ has started its engines and remains true unless it parks or takes off.

- The fluent $isPushing(Airplane, S)$ holds in situation S if $Airplane$ is being pushed back from a gateway, but if $Airplane$ starts its engines, it is no longer pushing, but we say that it starts moving by itself.

- Apart from the airplane state, we also need to describe the current position of an airplane $atSegment(Airplane,Segment,S)$, and its heading $facing(Airplane,Direction,S)$. When an airplane moves or when it is pushed back, in the resulting situation it will be located at another segment. Movements of other airplanes have no effect. Similar, moving or pushed airplanes will face a new direction in the resulting situation. Recall that a direction is simply a vertex of a segment.

A correct state of an airplane is defined by the following facts:

1. An airplane is at exactly one segment or is airborne.
2. If airplane is airborne it neither occupies nor blocks any segments.
3. An airplane is facing in exactly one direction or is airborne (then it is not facing any direction).
4. If airplane is moving it only blocks the segments determined by the $segIsBlocked$ predicate and the one it occupies.
5. If airplane is parked it only blocks the segment it occupies.
6. An airplane never blocks airplane segment occupied by another airplane.
7. An airplane never occupies airplane segment blocked by another airplane.

All actions are **terms** with the following meaning.

- The action $move(Airplane, Type, Dir1, Seg1, Seg2, Dir2)$ moves $Airplane$ of category $Type$ facing direction $Dir1$ from $Seg1$, where it is located in the current situation, to $Seg2$ where its new direction is $Dir2$. This action is possible only if blocking constraints are followed. Namely, the destination segment should not be blocked by another airplane in the current situation, and $Airplane$ moving to $Seg2$, $Dir2$ should not block anyone else. The latter means that no other segment with another airplane at that segment is blocked from $Seg2$, $Dir2$. Most of the effects of the move action should be self-explanatory. Namely, $move$ results in updating of the occupied segment, changing the heading if necessary, and blocking the occupied segment. When an airplane is moving, it cannot turn 180 degrees in place.

- The action $pushback(Airplane, Type, Dir1, Seg1, Seg2, Dir2)$ is similar, but it is simpler than the previous action. Because an airplane is transported from a segment where it is parked without starting its engines, it does not block any other segments (except the one it occupies) once it has been pushed somewhere. Only parked airplanes can be pushed back, and once an airplane has been pushed from a segment, it can start up its engines. An airplane that is being pushed back can turn 180 degrees, and then $Seg1 = \langle V1, V2 \rangle \neq Seg2 = \langle V2, V1 \rangle$ and $Dir1 \neq Dir2$, or it can move back one segment without turning around (if a destination segment is not occupied by another airplane), or it can move and turn at an angle.

- The action $startup(Airplane)$ represents the process of starting airplanes engines after it has been pushed back from its park position. That is why its precondition should contain the $isPushing$ predicate. It is intended only for out-bound airplanes, but any airplane that is not moving can execute this action. If the airplane starts its engines, then it begins blocking segments. So we need the exact same check for other segments as in the $move$ action: it is possible to start up only when no segment with another airplane is blocked from the segment where $Airplane$ is currently located. Apart from that we only update the airplane state to $isMoving$.

- The action $park(Airplane, Type, Seg, Dir)$ does the opposite of the $startup$ action by unblocking all segments except the occupied one. It is possible to park at a tuple (Seg, Dir) in situation S only if $Airplane$ is moving, it is located at Seg, it is facing Dir, and if Seg is not one of runway segments. Note that a parking action has 4 arguments, but the related fluent predicate $isParked(Airplane, Seg, S)$ has as its parameters only $Airplane$ and $Segment$ where it is parked, apart from situation S.

- The $takeoff(Airplane, Seg, Dir)$ action makes sure the airplane is completely removed from the airport – meaning it does not block or occupy any segments anymore. This action is possible in S, if $Airplane$ is at a runway Seg, if it is facing Dir, and if it is moving.

You have to solve planing problems in this application using the situations and fluents approach considered in class. The task of your planner will be to find a shortest list of actions such that after doing actions from this list a goal situation will be reached starting from the initial situation.

Before you can solve planning problems in this domain, you have to write precondition axioms and successor state axioms. Write all your Prolog rules in the file **airport.pl** More specifically, you have to do the following.

(A) Write precondition axioms for all five actions. Recall that to avoid potential problems with negation in Prolog, you should not start bodies of your rules with negated predicates. Make sure that all variables in a predicate are instantiated by constants before you apply negation "not" to the predicate that mentions these variables.

(B) Write successor-state axioms that characterize how the truth values of all fluents change from the current situation S to the next situation $[A|S]$. You will need two types of rules for each fluent: (1) rules that characterize when a fluent becomes true in the next situation as a result of the last action, and (2) rules that characterize when a fluent remains true in the next situation, unless the most recent action changes it to false.

(C) Once you are done, you have to test your precondition and successor state axioms using the following initial and goal situations. Download the file **initAirport.pl** with descriptions of your initial and goal states from the Assignments Web page. Do not copy content of **initAirport.pl** into your file **airport.pl** because TA will use other initial and goal states to test your program in **airport.pl**

```
/*           Objects in a simple airport domain
%% the airplanes: out-bound boeing347 - medium  and  in-bound md25 - light
%% the segments        v0
                       |
                       v8
                       |
               v5--v7--v2--v9--v1
                |      |
                |      v6
                |      |
               v4------v3                                */
adjacent(v0,v8).                              adjacent(v1,v9).
adjacent(v8,X) :- member(X,[v0,v2]).           adjacent(v9,X) :- member(X,[v1,v2]).
adjacent(v7,X) :- member(X,[v5,v2]).           adjacent(v6,X) :- member(X,[v3,v2]).
adjacent(v2,X) :- member(X,[v8,v9,v6,v7]).    adjacent(v3,X) :- member(X,[v6,v4]).
adjacent(v4,X) :- member(X,[v3,v5]).           adjacent(v5,X) :- member(X,[v7,v4]).
```

```
isStartRunway(seg(v3,v4), v4).    isStartRunway(seg(v5,v4), v4).

atSegment(boeing347,seg(v8,v0),[]).  blocked(seg(v8,v0),boeing347,[]).
blocked(seg(v0,v8),boeing347,[]).
hasType(boeing347,medium). facing(boeing347,v0,[]). isParked(boeing347,seg(v8,v0),[]).

atSegment(md25,seg(v4,v3),[]). blocked(seg(v4,v3),md25,[]). blocked(seg(v3,v4),md25,[]).
hasType(md25,light).  facing(md25,v3,[]).   isMoving(md25,[]).

%% First planning problem to solve: use this goal state
goal_state(S) :- airborne(boeing347,Seg,S).

%% Second planning problem to solve: use this goal state
%  goal_state(S) :-  isParked(md25,seg(v9,v1),S).

%%This 3rd planning problem needs 12 steps and takes about 10min without heuristics.
%goal_state(S) :-
% isParked(md25,seg(v9,v1),S),
% airborne(boeing347,Seg,S).
```

Solve this simple planning problems given to you using the planner with the upper bound 10 on the number of actions. It should take no more than a few seconds for your planner to solve each problem. Comment-out the rule for the first *goal_state* predicate (with out-bound boeing347) and remove comments from the rule for the second *goal_state* predicate (with in-bound md25) to test the 2nd planning problem. Once you are satisfied with the results you obtained, comment-out the 1st and 2nd *goal_state* rules, and test the 3rd planning problem (with both planes). Collect the results of your tests in your file **airport.txt** If you would like to debug your program, you can try a simpler goal state, or check if consecutive sequences of actions are possible as they should be, and if they have effects you expect. Request a few plans using ";" command (there is no need to request all of them). Why are they different? Write a brief summary of your results.

Handing in solutions: An electronic copy of your file **airport.pl** and a copy of **airport.txt** must be included in your **zip** archive. The file **airport.txt** must include copies of all your session(s) with Prolog, showing all the queries you submitted and the answers returned. Write also what hardware (CPU, memory) you used to solve this planning problem.

3. Bonus work (**20 points**):

To make up for a grade on another assignment or test that was not what you had hoped for, or simply because you find this area of Artificial Intelligence interesting, you may choose to do extra work on this assignment. *Do not attempt any bonus work until the regular part of your assignment is complete.* If your assignment is submitted from a group, write whether this bonus question was implemented by all people in your team (in this case bonus marks will be divided evenly between all students) or whether it was implemented by one person only (in this case only this student will get all bonus marks).

Consider the modified version of the generic planner:

```
reachable(S2, [M | ListOfActions]) :- reachable(S1,ListOfActions),
                        legal_move(S2,M,S1),
                        not useless(M,ListOfActions).
```

The predicate `useless(A,ListOfActions)` is true if an action A is useless given the list of previously executed actions. If this predicate is defined using proper rules, then it helps to speed-up the search that your program does by filtering out the redundant or irrelevant actions. This predicate provides (application domain dependent) declarative heuristic information about the planning problems that your program solves. The more inventive you are when you implement this predicate, the less search will be required to find a list of actions that solves the planning problems. However, any implementation of rules that define this predicate should not use any information related to the specific initial situation. Your rules should be good enough to work with any initial and goal states. When you write rules that define this predicate use common sense properties of the application domain. Write your rules for the predicate `useless` in the file **bonus.pl**: it must include the program **airplane.pl** that you created in Part 2 of this assignment. You have to use the file **initBonus.pl** with the specifications of the new goal and initial states. As before, do not include this file in your file **bonus.pl** Once you have the rules for the predicate `useless(A,List)`, solve another planning problem, this time using the modified planner:

```
atSegment(boeing347, seg(v8,v0),[]).   blocked(seg(v8,v0),boeing347,[]).
blocked(seg(v0,v8),boeing347,[]).
hasType(boeing347,medium). facing(boeing347,v0,[]). isParked(boeing347,seg(v8,v0),[]).

atSegment(md25, seg(v9,v1),[]). blocked(seg(v9,v1),md25,[]). hasType(md25,light).
facing(md25,v1,[]). isParked(md25,seg(v9,v1),[]). blocked(seg(v1,v9),md25,[]).

% The planning problem when both planes are out-bound: it is computationally
% difficult without declarative heuristics.
goal_state(S) :- airborne(md25,Seg2,S), airborne(boeing347,Seg1,S), not Seg1=Seg2.
```

When you solve this planning problem look for a plan that has no more then 14 actions. If your rules are creative enough, solving this problem should be fast (a few minutes or faster). Next, take the 3rd planning problem from Part 2 of this assignment. Solve it again, but now using you modified planner. How much time you save? *Explain briefly why you believe that your rules help to cut useless parts of search space.* Once, you solved this problem, try to solve the same planning problem without using rules for the predicate `useless(A,L)`: put comments on the rule that defines the predicate `reachable(S,[A|L])` and calls the predicate `useless(A,L)`. Warning: your program may take much longer to solve this version (depending on how fast is your computer). Request several plans using ";" command.

Finally, write a brief report (in the file **bonus.txt**): all queries that you have submitted to your program (with or without heuristics) and how much time your program spent to find several plans when you added more heuristics. Discuss briefly your results and explain what you have observed. Mention on which computer you did testing: what CPU and memory are available to your program. Note that TA who will be marking your assignment will use another specification of initial and goal states.

Handing in solutions: An electronic copy of both files **bonus.pl** and **bonus.txt** with your session with Prolog must be included in your **zip** archive. You have to write brief comments in **bonus.pl** with explanations of your declarative heuristics. You lose marks if you do not explain. Your Prolog file **bonus.pl** should contain only your precondition and successor state axioms and rules for the predicate `useless(A,List)`, but should not include anything related to the initial and goal states.

How to submit this assignment. Read regularly *Frequently Answered Questions* and answers at
`http://www.scs.ryerson.ca/~mes/courses/cps721/assignments.html`
If you write your code on a Windows machine, make sure you save your files as plain text that one can easily read on Linux machines. Before you submit your Prolog code electronically make sure that your files do not contain any extra binary symbols: it should be possible to load `put.pl` or `airport.pl` into a recent release 6 of ECLiPSe Prolog, compile your program and ask testing queries. TA will mark your assignment using ECLiPSe Prolog. If you run any other version of Prolog on your home computer, it is your responsibility to make sure that your program will run on ECLiPSe Prolog (release 6 or any more recent release), as required. For example, you can run a command-line version of *eclipse* on moon remotely from your home computer to test your program (read handout about running *ECLiPSe Prolog*). To submit files electronically do the following. First, create a **zip** archive:

```
zip yourLoginName.zip put.pl put.txt airport.pl airport.txt [bonus files]
```

where `yourLoginName` is the login name of the person who submits this assignment from a group. Remember to mention at the beginning of each file *student, section numbers* and *names* of all people who participated in discussions (see the course management form). You may be penalized for not doing so. Second, upload your file **yourLoginName.zip** (make sure it includes **all** files) to D2L into "Assignment 5" folder.

Improperly submitted assignments will **not** be marked. In particular, you are **not** allowed to submit your assignment by email to a TA or to the instructor.

Revisions: If you would like to submit a revised copy of your assignment, then run simply the submit command again. (The same person must run the submit command.) A new copy of your assignment will override the old copy. You can submit new versions as many times as you like and you do not need to inform anyone about this. Don't ask your team members to submit your assignment, because TA will be confused which version to mark: only one person from a group should submit different revisions of the assignment. The time stamp of the last file you submit will determine whether you have submitted your assignment on time.