cps721: Assignment 5 (100 points).
Due date: Electronic file - Thursday, November 29, 2018, 16:00 (sharp).

You can get extra 10% by submitting this assignment before Sunday, November 25, 16:00.
You must work in groups of TWO, or THREE, you cannot work alone.
YOU SHOULD NOT USE ";" , "!" AND "–>" IN YOUR PROLOG RULES.
ALSO, YOU CANNOT USE SYSTEM PREDICATES OR LIBRARIES NOT MENTIONED IN CLASS.

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 solve planning problems using the situations and fluents approach. For each of Parts 1 and 2, you should use a file containing the rules for `solve_problem`, `reachable`, `max_length(List,Bound)` as explained in class, and then add your own precondition and successor state axioms using terms and predicates specified in the assignment. Read your lecture notes for examples: we discussed in class tiles-on-grid, coins-on-desk and sorting examples implemented using situations and fluents approach. (Download the planner from the Assignments page). As you will see from this assignment, there are both everyday and CS problems that can be fomulated as the deterministic AI Planning problem.

1 (50 points). In the not-too-distant future, the robot assistants will be helping humans both at homes and at work. We hope that giving instructions to them will be easy. Instead of programming every detailed step of every motion that the robot has to do, the future robots will simply take verbal orders such as "do laundry" when they are given the piles of dirty clothes. In this part, the task is to figure out how the robot will have to solve this planning problem.

We consider below terms (actions) and predicates (fluents) that are general enough to deal with any collections of clothes, washing and drying machines. In particular, we consider the following ten actions (they are **terms**).

- $fetch(O, C)$: fetch object O from cupborad C. After fetching, you are holding O, and it is no longer in C. This action is possible if you are not currently holding anything.
- $putAway(O, C)$: put away object O into cupborad C. After doing this action, you are no longer holding O, but it is now in C. This action is possible if you are currently holding O.
- $addSoap(P, W)$: after adding soap P to washer W, W has soap. This action is possible if you are holding P, and W does not currently have soap.
- $addSoftener(T, W)$: add fabric softener T to washer W. This action is possible if you are holding T and W does not currently have softener.
- $removeLint(D)$: remove lint from dryer D. After doing this action, D no longer has lint. It is possible if D currently has lint.
- $washClothes(C, W)$: wash clothes C in washer W. In the situation resulting from execution of this action, C is clean and wet, and W has neither soap nor softener. This action is possible if C is in W, C is not clean, and W has soap and softener.
- $dryClothes(C, D)$: dry clothes C in dryer D. In the situation resulting from execution of this action, C is not wet, D has lint. The action is possible if C is in D, C is wet, and D does not have lint.
- $fold(C)$: fold clothes C. This action is possible if C is not folded, clothes are clean and dry, and you are not currently holding anything.

- $wear(C)$: wear clothes C. After doing this action, C is neither clean nor folded. This action is possible if C is folded.

- $move(C, F, T)$: move clothes C from a location F to a location T. After doing this action, C is in T, but it is no longer in F. You can do this action if you are not currently holding anything, C is in F, both F and T are containers. The container can be a dresser, or a hamper, or a washer, or a dryer that is empty.

We also consider the following fluents (they are **predicates**).

- $in(O, C, S)$: object O is inside C in situation S.

- $holding(O, S)$: you are holding object O in situation S after fetching O from somewhere. You are no longer holding an object, if you put it away, or if you add it to a washer machine, in case if the object is a soap or a fabric softener.

- $hasSoap(W, S)$: washer W has soap in situation S.

- $hasSoftener(W, S)$: washer W has fabric softener in situation S.

- $hasLint(D, S)$: dryer D has lint in situation S.

- $clean(C, S)$: clothes C are clean in situation S.

- $wet(C, S)$: clothes C are wet in situation S (after washing, and before drying).

- $folded(C, S)$: clothes C are folded in situation S.

Finally, there are auxiliary predicates that do not have situational argument. Their meaning should be self-explanatory: $cupboard(B)$, $washer(W)$, $soap(P)$, $softener(T)$, $dryer(D)$, $clothes(C)$, $hamper(H)$, $container(N)$.

Part (a). Write the following rules in your file **laundry.pl** (you can download this file from D2L):

1. Write precondition axioms for all actions in your domain. 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 to the predicate that mentions these variables.

2. Write successor-state axioms that characterize how the truth value 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: (a) rules that characterize when a fluent becomes true in the next situation as a result of the last action, and (b) rules that characterize when a fluent remains true in the next situation, unless the most recent action changes it to false. When you write successor state axioms, you can sometimes start bodies of rules with negation of a predicate, e.g., with negation of equality. This can help your program work a bit more efficiently.

3. Consider the following initial and goal configurations.

```
washer(w1).      dryer(d1).
clothes(cl1).    clothes(cl2).    hamper(h1).       hamper(h2).
soap(p1).        soap(p2).        cupboard(cbd1).   cupboard(cbd2).
softener(sft1).  softener(sft2).
in(cl1,h1,[]).   in(cl2,h2,[]).   in(p1,cbd1,[]).   in(p2,cbd2,[]).
in(sft1,cbd1,[]).       in(sft2,cbd2,[]).
/* Goal 1 */
goal_state(S) :- clean(cl1,S).
```

Solve this simple planning problem using the planner with the upper bound **6** on the number of actions (download the initial and goal configurations from the Assignments Web page: the file **laundryInit.pl**).

It should take no more than a few seconds for your planner to solve this problem. If you have to debug your rules, try a shorter sub-sequence of actions that should be consecutively executable and check for them if your program correctly identifies they are possible, and if your program computes correctly their effects. Avoid queries with the variable S, since they amount to solving the unbounded planning problem. Keep a copy of your session with Prolog in your file **laundry.txt**
Do not copy content of **laundryInit.pl** into your file **laundry.pl** because TA may use other initial and goal states to test your program in **laundry.pl** Request several plans using "more" command (keep all plans in file **laundry.txt**). Read them carefully to convince yourself that all of them are correct.

Part (b). 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 performed actions. If this predicate is defined using proper rules, then it helps to speed-up the search that your program is doing to find a list of actions that solves the problem. 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 solve the planning problems. However, any implementation of rules that define this predicate should **not** use any information related to the specific initial or goal situations. Your rules should be general 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 **laundry.pl**: it must include the program that you created in the previous part of this assignment. You have to write at least 4 different rules. Make sure they make sense. The specification of the goal and initial state must remain in the file **laundryInit.pl**, but you must make appropriate modifications (remove comments, etc) in that file. Once you have the rules for the predicate `useless(A,List)`, solve 2 other planning problems, this time using the modified planner:

```
/* Goal 2 */
goal_state(S) :- clean(cl1,S), not wet(cl1,S).
/* Goal 3 */
%goal_state(S):- clean(cl1,S), not wet(cl1,S), folded(cl1,S), in(cl1,dresser,S).
```

In the 2nd planning problem, we simply ask the robot to make sure clothes are dry. This problem can be solved with 8 actions in about 5sec or less. In the 3rd planning problem, we ask the robot additionaly to figure out what has to be done to store dry folded clothes in a dresser. When you solve this 3rd planning problem look for a plan that has no more then 10 actions. If your rules are creative enough, solving this problem should be fast, e.g., in less than 2 min. Request several plans using "more" button (or using ";" command). Once, you solved it, try to solve the 2nd or 3rd planning problems without using rules for the predicate `useless(A,L)`: put comments on the rule that defines the predicate `reachable(S,[A|L])` by using the predicate `useless(A,L)`. Warning: your program may take a few minutes to solve this version (depending on how fast is your computer).

Write a brief report (in the file **laundry.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. You have to write brief comments in

laundry.pl with explanations of your declarative heuristics. Note that TA who will be marking your assignment will use another specification of initial and goal states. You are expected to submit the fastest version of your program with the declarative heuristics.

Handing in solutions: An electronic copy of your files **laundry.pl** and **laundry.txt** must be included in your **zip** archive. Your Prolog file **laundry.pl** should contain only your precondition and successor state axioms and rules for the predicate `useless(A,List)`. The file **laundry.txt** should contain the queries to solve the problem, the computed plans and information about time that your program spent on finding a plan. It must include also your brief report about what you have observed. Explain what effect heuristics have on the running time. Write also on what computer (mention CPU, memory) you solved this planning problem.

2 (50 points). In computer science, the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. In this assignment, it is used as an instance of a planning problem. The following rendering of this problem is adapted from *Wikipedia*. Let n silent philosophers sit at a circular table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. Each philosopher can be in the following states: thinking, or waiting, or eating. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. After he finishes eating, he needs to put down both the forks so they become available to others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them. Eating is not limited by the amount of spaghetti left: assume an infinite supply. However, at most 2 philosophers can eat spaghetti from the bowl at the same time. If 2 philosophers are currently eating, and the third one is trying to eat, then he will not be able to eat, but he will have to wait.

You have to solve this planning problem using an approach considered in class. Specifically, you have to take the generic planner written in Prolog, but you have to provide missing rules as explained below. Subsequently, we consider a simplified version of this problem and several different goal states that are increasingly more difficult to reach from a given initial state. Start with writing the precondition axioms: these are rules (one rule per action) which say when actions are physically possible to execute in the current state. When you write the precondition axioms, concentrate on the common-sense meaning of actions (not on the fact whether an action is useful or not). There are the following actions in this application domain. (Recall that actions are **terms**: they can be only arguments of predicates or equality.)

- $pickUp(P, F)$: a philosopher P picks up an adjacent fork F provided this fork is currently available in situation S. This action is possible for P if either F is the fork on his left, or if F is the fork on his right. A philosopher P cannot pick up forks which are not adjacent to him. When P picks up F, the fork F is no longer (freely) available because P has it now in his hand.

- $putDown(P, F)$: this action will undo the effects of the pick up action. A philosopher P can execute this action if he currently holds F in his hand.

- $tryToEat(P)$: this action is possible if a philosopher P has two different adjacent forks in his hands.

There are several auxiliary predicates that are not fluents: their truth values do not change after executing any of the actions. In particular, $philosopher(P)$ means that P is a philosopher, $fork(F)$ means that F is a fork, $left(P, L)$ means that the philosopher L is seating to the left from P, $right(P, R)$ means that the philosopher R is seating to the right from P, $between(P1, F, P2)$ means that a fork F is located in between the philosophers $P1$ and $P2$. These predicates do not change from one situation to another.

To represent features of this domain that can change, it is sufficient to introduce the following predicates with situation argument (fluents). Recall that fluents are **predicates** with a situation S as the last argument, where situations are represented by lists of actions that have been already executed. Situation can be understood as a history, i.e., as a sequence of executed actions. Since fluents are predicates, action terms can occur as arguments in fluents.

- $available(F, S)$: a fork F is available in S. The fork becomes available if someone puts it down, and not available when someone picks it up.
- $has(P, F, S)$: a philosopher P holds a fork F in S. This fluent becomes true after doing $pickUp(P, F)$ action, and it remains true unless $putDown(P, F)$ action is executed.
- $thinking(P, S)$: a philosopher P is thinking in situation S. This is true if he had the only one fork and was waiting in the previous situation, but he put down the only fork in his possession, or if P was already thinking and did not pick up any fork.
- $eating(P, S)$: a philosopher P is eating in situation S if his latest action was a successful attempt to eat (there were no 2 other philosophers eating at that time), or if he was eating before and has not put down one of the two forks that he holds. If a waiting philosopher is trying to eat when 2 other philosophers are already eating, he cannot eat, but keeps waiting.
- $waiting(P, S)$: a philosopher P is waiting in situation S if he is neither thinking nor eating. He switches from thinking to waiting by picking up a fork, or from eating to waiting by putting down one of his two forks. The philosopher P remains waiting, if he was waiting in the previous situation and could not start eating after attempting to eat (because spaghetti in the bowl was consumed by 2 other philosophers), or if he was waiting in the previous situation with at least one fork in his hands, and he did not put down the last fork. Recall that if any waiting philosopher has only one fork, he starts thinking after he puts down this fork.
- $happy(P, S)$: a philosopher P becomes happy after eating in the previous situation as soon as he puts down at least one of his forks. If the philosopher P is happy, he remains happy no matter what he or other philosophers do subsequently.

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 **philosophers.pl** provided for you. Write precondition axioms for all 3 actions. Write successor-state axioms that characterize how the truth value of all 6 fluents change from the current situation S to the next situation $[A|S]$. You will need two types of rules for each fluent: (a) rules that characterize when a fluent becomes true in the next situation as a result of the last action, and (b) rules that characterize when a fluent remains true in the next situation, unless the most recent action changes it to false. To make sure that there are no errors in your rules, test them on solving simple planning problems as recommended below. Consider the following description of initial situation []:

```
philosopher(p1).  philosopher(p2).  philosopher(p3). fork(f1). fork(f2). fork(f3).
left(p1,p2). left(p2,p3). left(p3,p1). right(p1,p3). right(p3,p2). right(p2,p1).
/*              p2
               /  \
              f1  f2
             /      \
          p1--f3--p3
*/
between(p1,f1,p2). between(p2,f2,p3). between(p3,f3,p1).
between(p2,f1,p1). between(p3,f2,p2). between(p1,f3,p3).
thinking(p1, []). thinking(p2, []). thinking(p3, []).
available(f1, []). available(f2, []). available(f3, []).
```

```
goal_state(S) :- happy(p1, S), waiting(p2,S).
/*  goal_state(S) :- happy(p1,S), happy(p2,S). */
```

Download the file **philosophersInit.pl** with descriptions of the initial and goal states from the Assignments Web page. Keep both this file and the file **philosophers.pl** in the same folder. When ECLiPSe Prolog compiles the file **philosophers.pl** it loads the file **philosophersInit.pl** automatically.

Do not copy content of **philosophersInit.pl** into your file **philosophers.pl** because a TA may use other initial and goal states to test your program in **philosophers.pl** Do not submit the file **philosophersInit.pl**

Solve several simple planning problems using the planner with the upper bound 8 on the number of actions: use goal states mentioned above, or any intermediate states. Solving these planning problems does not need lots of search, so your program should solve each of them in a few seconds (less than 1 minute even if your CPU is slow). If your program cannot solve any of these simple problems in 1 minute or sooner, or if it computes a wrong plan, then there is a bug in your program. You can either rely on Prolog's tracer to debug your program, or you can try queries testing intermediate states of computation that your program does (you can use *poss* or fluents to formulate testing queries), or you can try another simpler goal state. Testing your program with simple queries can help you to locate a bug faster than trying to trace it with a debugger. Request several plans using ";" command. Discuss briefly your results in the file **philosophers.txt**

Handing in solutions: An electronic copy of your files **philosophers.pl** as well as a copy of **philosophers.txt** must be included in your **zip** archive. The file **philosophers.txt** must include a copy of session(s) with Prolog, showing all the queries you submitted and the answers returned. Write also what computer (manufacturer, CPU, memory) you use to solve this planning problem. If you are working in lab, check system info or ask a system administrator about CPU and memory installed on your machine. It is recommended to use Linux servers instead of Windows machines: read the handout about running ECLiPSe Prolog in labs.

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,ListOfPastActions)` is true if an action A is useless given the list of previously executed actions. The predicate `useless(A,ListOfPastActions)` helps to solve the planning problem by providing *declarative heuristics* (advises) to the planner. If this predicate is correctly defined using a few rules (one or more rules per action A), then it helps to speed-up the search that your program is doing to find a list of actions solving a problem. Write as many rules as you can to implement this predicate: think about useless repetitions that should be avoided, and about order of execution (i.e., use common sense properties of the application domain). Your rules should never use any constants mentioned in the initial or goal states because your rules have to be domain specific, but they should be independent of a particular instance of the planning problem that you are solving. Consequently, your rules defining *useless* can use only variables. Write your rules for the predicate `useless` in the file **bonus.pl**: it must include the program **philosophers.pl** that you created in Part 2 of this assignment. Your rules have to be general enough to be applicable to any number of philosophers. In other words, they have to speed-up solving a planning problem for any instance (i.e., any initial and goal states), not just those which are given to you. Once you have wrote rules for `useless(A,ListOfPastActions)`, test them on same planning problem as above, or any similar simple planning problems.

Once you have convinced yourself that your rules implementing *useless* are correct, try to solve the planning problem with the following more challenging goal:

```
goal_state(S) :- happy(p1,S), happy(p2,S), happy(p3,S).
```

Use 13 as the upper bound on the number of actions. **Warning**: solving an instance of this planning problem without declarative heuristics can take 5-7 hours of CPU time (or longer if you work in a lab, or if your CPU is slow). However, if you have a clever set of heuristics, then solving this or similar problems should take at most a few minutes. In other words, well-thought heuristics should provide 100 times speed-up. You have to try several different goal states to deserve full mark for bonus work. Include your testing results and a **brief discussion** of your results in the report **bonus.txt**

Handing in solutions: An electronic copy of your files **bonus.pl** as well as a copy of **bonus.txt** must be included in your **zip** archive. The file **bonus.txt** must include a copy of session(s) with Prolog, showing all the queries you submitted and the answers returned. Write also what computer (manufacturer, CPU, memory) you use to solve this planning problem. If you are working in lab, check system info or ask a system administrator about CPU and memory installed on your machine. It is recommended to use Linux servers instead of Windows machines: read the handout about running ECLiPSe Prolog in labs.

How to submit this assignment. Read regularly *Frequently Answered Questions* and replies to them that are linked from the Assignments Web page 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 `nlu.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 on `moon`:

```
   zip yourLoginName.zip laundry.pl laundry.txt philosophers.pl philosophers.txt
[bonus]
```

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 ZIP** file only (**No individual files!**) **yourLoginName.zip** into the "Assignment 5" folder on D2L.

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.