Assignment Sheet
EVEN Semester 2021
B.Tech CSE 6th Semester

Course: Artificial Intelligence Lab
CourseCode: 15B17CI574

Instructions:

- Students have to do a mini project apart from the Lab Assignments.
- The evaluative lab assignments must be submitted as per the given deadline.
- The total weightage of all day-to-day work is 60 Marks.
- There will be two lab tests of 20 marks each.
- Absence in LabTest-2 means Fail in the lab course.
- All students are required to attend atleast 80% labs. 15 marks are reserved for attendance.

Week-8&9 April 12-24, 2021

Prolog has a built-in backward chaining inference engine which can be used to partially implement some expert systems. Prolog rules are used for the knowledge representation, and the Prolog inference engine is used to derive conclusions. Other portions of the system, such as the user interface, must be coded using Prolog as a programming language.

The Prolog inference engine does simple backward chaining. Each rule has a goal and a number of sub-goals. The Prolog inference engine either proves or disproves each goal. There is no uncertainty associated with the results.

- You need to represent the knowledge base in online Prolog engine and analyse the inference results.
- Try out the different problem statements and run your queries on the following engine: https://swish.swi-prolog.org/example/prolog_tutorials.swinb
- Read the documentation on the above link mentioned

Example 1:

Predicate definitions that calculate the factorial function:

```
factorial(0,1).

factorial(N,F) :-
  N>0,
  N1 is N-1,
  factorial(N1,F1),
  F is N * F1.
```

This program consists of two *clauses*. The first clause is a unit clause, having no body. The second is a rule, because it does have a *body*. The body of the second clause is on the right side of the ':-' which can be read as "if". The body consists of literals separated by commas ',' each of which can be read as "and". The head of a clause is the whole clause if the clause is a unit clause, otherwise the *head* of a clause is the part appearing to the left of the colon in ':-'. A declarative reading of the first (unit) clause says that "the factorial of 0 is 1" and the second clause declares that "the factorial of N is F if N>0 and N1 is N-1 and the factorial of N1 is F1 and F is N*F1".

The Prolog goal to calculate the factorial of the number 3 responds with a value for W, the goal variable:

?- factorial(3,W).
W=6

Knowledge base Representation	Inference	

/* Two Factorial Definitions */
 	
factorial(0,1).
 	
factorial(N,F) :-
  N>0,
  N1 is N-1,
  factorial(N1,F1),
  F is N * F1.
	?- factorial(3,W).	W=6

Example 2:

A famous problem in mathematics concerns coloring adjacent planar regions. Like cartographic maps, it is required that, whatever colors are actually used, no two adjacent regions may not have the same color. Two regions are considered adjacent provided they share some boundary line segment. Consider the following map.

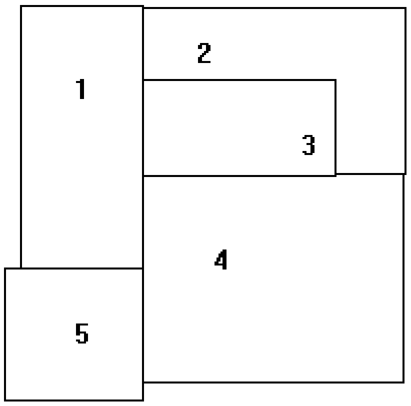


Fig. 1

We have given numerical names to the regions. To represent which regions are adjacent, consider also the following graph.

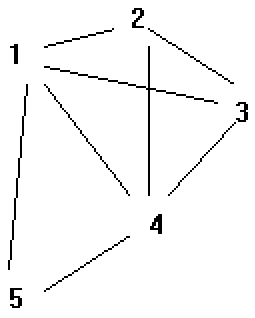


Fig. 2

Here we have erased the original boundaries and have instead drawn an arc between the names of two regions, provided they were adjacent in the original drawing. In fact, the adjacency graph will convey all of the original adjacency information. A Prolog representation for the adjacency information could be represented by the following *unit* clauses, or facts.

```
adjacent(1,2).      adjacent(2,1).
adjacent(1,3).      adjacent(3,1).
adjacent(1,4).      adjacent(4,1).
adjacent(1,5).      adjacent(5,1).
adjacent(2,3).      adjacent(3,2).
adjacent(2,4).      adjacent(4,2).
adjacent(3,4).        adjacent(4,3).
adjacent(4,5).        adjacent(5,4).
```

If these clauses were loaded into Prolog, we could observe the following behavior for some goals.

```
?- adjacent(2,3).
yes
?- adjacent(5,3).
no
?- adjacent(3,R).
R = 1 ;
R = 2 ;
R = 4 ;
no
```

One could declare colorings for the regions in Prolog also using unit clauses.

```
color(1,red,a).    color(1,red,b).
color(2,blue,a).   color(2,blue,b).
color(3,green,a).  color(3,green,b).
color(4,yellow,a). color(4,blue,b).
color(5,blue,a).   color(5,green,b).
```

Here we have encoded 'a' and 'b' colorings. We want to write a Prolog definition of a conflictive coloring, meaning that two adjacent regions have the same color. For example, here is a Prolog clause, or rule to that effect.

```
conflict(Coloring) :-
   adjacent(X,Y),
color(X,Color,Coloring),
color(Y,Color,Coloring).
```

For example,

```
?- conflict(a).
no
?- conflict(b).
yes
?- conflict(Which).
Which = b
```

Here is another version of 'conflict' that has more logical parameters.

```
conflict(R1,R2,Coloring)                                             :-
   adjacent(R1,R2),
color(R1,Color,Coloring),
color(R2,Color,Coloring).
```

Prolog allows and distinguishes the two definitions of 'conflict'; one has one logical parameter ('conflict/1') and the other has three ('conflict/3'). Now we have

```
?- conflict(R1,R2,b).
R1 = 2   R2 = 4
?- conflict(R1,R2,b),color(R1,C,b).
R1 = 2   R2 = 4   C = blue
```

The last goal means that regions 2 and 4 are adjacent and both are blue. Grounded instances like 'conflict(2,4,b)' are said to be consequences of the Prolog program. One way to demonstrate such a consequence is to draw a *program clause tree* having the consequence as the root of the tree, use clauses of the program to branch the tree, and eventually produce a finite tree having all true leaves. For example, the following clause tree can be constructed using fully grounded instances (no variables) of clauses of the program.

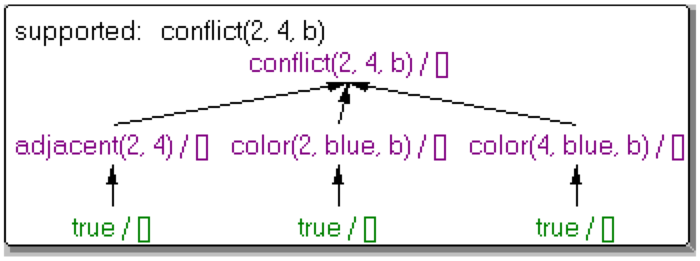


Fig. 3

The bottom leftmost branch drawn in the tree corresponds to the unit clause
adjacent(2,4).

which is equivalent in Prolog to the clause

adjacent(2,4) :- true.

Now, on the other hand, 'conflict(1,3,b)' is not a consequence of the Prolog program because it is not possible to construct a finite finite clause tree using grounded clauses of P containing all 'true' leaves. Likewise, 'conflict(a)' is not a consequence of the program, as one would expect. We will have more to say about program clause trees in subsequent sections.

```
/* Map Colorings */

adjacent(1,2).         adjacent(2,1).
adjacent(1,3).         adjacent(3,1).
adjacent(1,4).         adjacent(4,1).
adjacent(1,5).         adjacent(5,1).
adjacent(2,3).         adjacent(3,2).
adjacent(2,4).         adjacent(4,2).
adjacent(3,4).         adjacent(4,3).
adjacent(4,5).         adjacent(5,4).

/*-------------------------------------
*/

color(1,red,a).    color(1,red,b).
color(2,blue,a).   color(2,blue,b).
color(3,green,a).  color(3,green,b).
color(4,yellow,a). color(4,blue,b).
color(5,blue,a).   color(5,green,b).

/*-------------------------------------
*/

conflict(Coloring) :-
   adjacent(X,Y),
color(X,Color,Coloring),
color(Y,Color,Coloring).

/*-------------------------------------
-*/

conflict(R1,R2,Coloring) :-
   adjacent(R1,R2),
color(R1,Color,Coloring),
color(R2,Color,Coloring).
```

```
?- conflict(R1,R2,b).
R1 = 2   R2 = 4
?- conflict(R1,R2,b),color(R1,C,b).
R1 = 2   R2 = 4   C = blue
```

A *Boolean expression* is one of:

0	false
1	true
variable	unknown truth value
atom	universally quantified variable
~ *Expr*	logical NOT
Expr + *Expr*	logical OR
Expr * *Expr*	logical AND
Expr # *Expr*	exclusive OR
Var ^ *Expr*	existential quantification
Expr =:= *Expr*	equality
Expr =\= *Expr*	disequality (same as #)
Expr =< *Expr*	less or equal (implication)
Expr >= *Expr*	greater or equal
Expr < *Expr*	less than
Expr > *Expr*	greater than
card(Is,Exprs)	*see below*
+(Exprs)	*see below*
*(Exprs)	*see below*

where *Expr* again denotes a Boolean expression.

The Boolean expression card(Is,Exprs) is true iff the number of true expressions in the list *Exprs* is a member of the list *Is* of integers and integer ranges of the form From-To.

+(Exprs) and *(Exprs) denote, respectively, the disjunction and conjunction of all elements in the list *Exprs* of Boolean expressions.

Atoms denote parametric values that are universally quantified. All universal quantifiers appear implicitly in front of the entire expression. In residual goals, universally quantified variables always appear on the right-hand side of equations. Therefore, they can be used to express functional dependencies on input variables.

Exercises:

1. **Represent the following in Prolog**:

- Butch is a killer.
- Mia and Marsellus are married.
- Zed is dead.
- Marsellus kills everyone who gives Mia a footmassage.
- Mia loves everyone who is a good dancer.
- Jules eats anything that is nutritious or tasty.

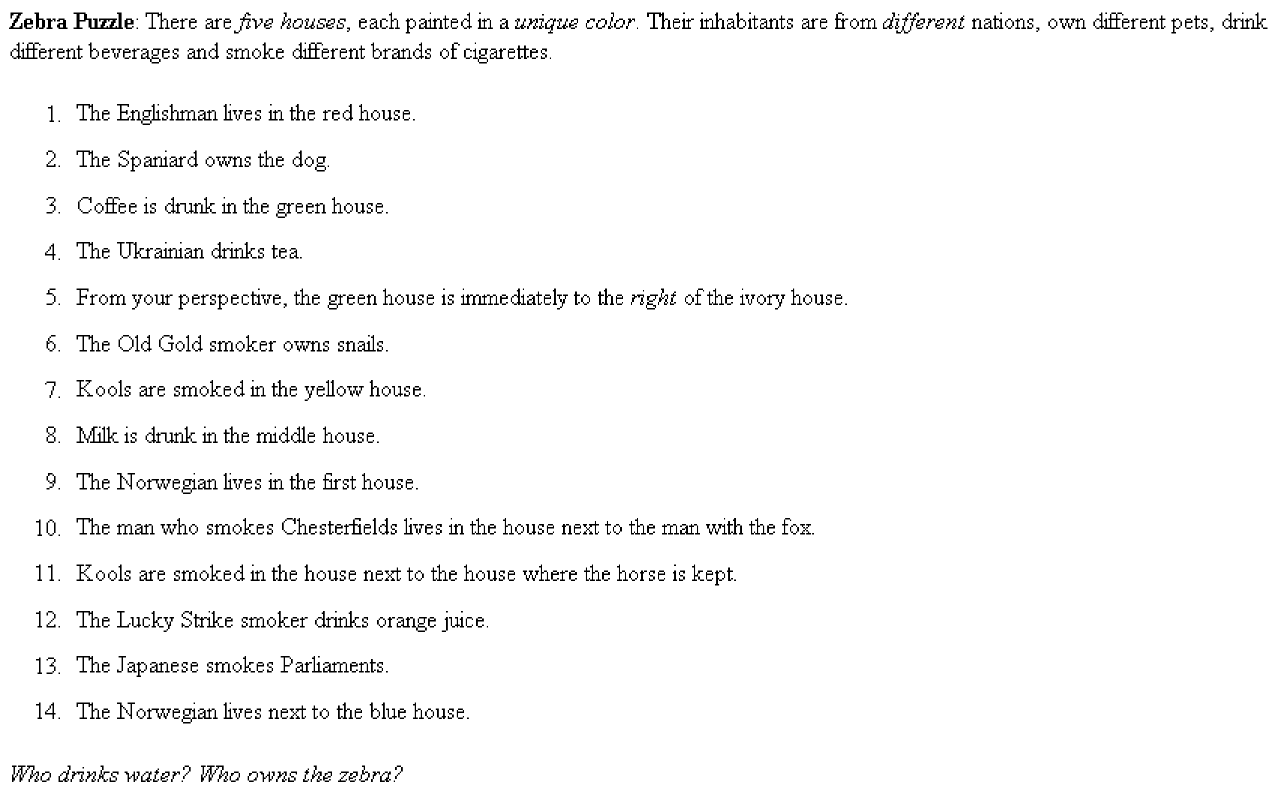
2. **Suppose we are working with the following knowledge base:**

```
wizard(ron).

hasWand(harry).

quidditchPlayer(harry).

wizard(X):- hasBroom(X), hasWand(X).

hasBroom(X):- quidditchPlayer(X).
```

How does Prolog respond to the following queries?

1. wizard(ron).
2. witch(ron).
3. wizard(hermione).
4. witch(hermione).
5. wizard(harry).
6. wizard(Y).
7. witch(Y).

3. Design the Knowledge base and infer the puzzle:



Zebra Puzzle: There are *five houses*, each painted in a *unique color*. Their inhabitants are from *different* nations, own different pets, drink different beverages and smoke different brands of cigarettes.

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. From your perspective, the green house is immediately to the *right* of the ivory house.
6. The Old Gold smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house.
10. The man who smokes Chesterfields lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

Who drinks water? Who owns the zebra?

4. Consider a group of ten friends who want to visit a new city somewhere in the world. They vote on seven potential destinations:

- Cairo
- London
- Beijing
- Moscow
- Mumbai
- Nairobi
- Jakarta

One city received four votes, two cities received two votes each, two cities received one vote each, and the remaining two cities received zero votes. How many votes did each of the cities receive?

- Beijing and Cairo got different numbers of votes.
- Moscow either got the most votes, or it got zero votes.
- Cairo got more votes than Jakarta did.

- In the list of cities above, each of the two cities that got two votes has a city that got no votes immediately above it in the list.
- Either Jakarta got one fewer votes than London did, or it got one fewer vote than Beijing did.

Provide a solution in Prolog.

5. There are 4 students: Carrie, Erma, Ora und Tracy. Each has one scholarship and one major subject they study. The goal is to find out which student has which scholarship and studies which subject (with all scholarships and majors being different from each other) from the hints provided. The available scholarships are: 25000, 30000, 35000 and 40000 USD. The available majors are: Astronomy, English, Philosophy, Physics. The following hints are given to solve the problem:

- The student who studies Astronomy gets a smaller scholarship than Ora.
- Ora is either the one who studies English or the one who studies Philosophy.
- Erna has a 10000 USD bigger scholarship than Carrie.
- Tracy has a bigger scholarship than the student that studies English.

6. Implement Wumpus World problem.