Jaypee Institute of Information Technology

Lab Sheet-9 (C372.2)

ODD Semester 2019 **B.Tech CSE/IT 5th Semester**

Course Artificial Intelligence Lab **Course Code 15B17CI574**

Instructions:

- All students are required to wear uniform else no attendance will be given plus minus 5 marks in disciplinary grade for not wearing uniform.
- Students have to do a mini project apart from the Lab Assignments.
- The evaluative lab assignments must be evaluated 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**. In case a student, who is absent in Lab Test 2 will be considered as Fail in the lab course.
- All students are required to attend at least **80%** labs. 15 marks are reserved for attendance. The evaluative lab assignments must be evaluated as per the given deadline from time to time.

Jaypee Institute of Information Technology

Lab Sheet-9 (C372.2)

ODD Semester 2019 **B.Tech CSE/IT 5th Semester**

Course Artificial Intelligence Lab **Course Code 15B17CI574**

----

1. Download and Install prolog http://www.swi-prolog.org/download/stable

Write and test the following Prolog predicates. These are practice for an upcoming Prolog quiz.

Help for PROLOG: \\jiit128adc16\CSE_IT\5th Semester\AI\AI Lab\Prolog

```
1. Here are some simple clauses.

likes(mary,food).
likes(mary,wine).
likes(john,wine).
likes(john,mary).

The following queries yield the specified answers.

 | ?- likes(mary,food).
 yes.
 | ?- likes(john,wine).
 yes.
 | ?- likes(john,food).
 no.

How do you add the following facts?

1. John likes anything that Mary likes
2. John likes anyone who likes wine
3. John likes anyone who likes themselves

===========================================================================

2. Slightly more complicated family tree.


                              James I
                                 |
                                 |
                +----------------+-----------------+
                |                                  |
             Charles I                          Elizabeth
                |                                  |
                |                                  |
     +----------+------------+                     |
     |          |            |                     |
 Catherine   Charles II   James II               Sophia
                                                   |
                                                   |
                                                   |
                                                George I
```

```
Here are the resultant clauses:
-------------------------------
  male(james1).
  male(charles1).
  male(charles2).
  male(james2).
  male(george1).

  female(catherine).
  female(elizabeth).
  female(sophia).

  parent(charles1, james1).
  parent(elizabeth, james1).
  parent(charles2, charles1).
  parent(catherine, charles1).
  parent(james2, charles1).
  parent(sophia, elizabeth).
  parent(george1, sophia).
```

Here is how you would formulate the following queries:

```
     Was George I the parent of Charles I?
              Query: parent(charles1, george1).
     Who was Charles I's parent?
              Query: parent(charles1,X).
     Who were the children of Charles I?
              Query: parent(X,charles1).
```

Now try expressing the following rules:

```
     M is the mother of X if she is a parent of X and is female
     F is the father of X if he is a parent of X and is male
     X is a sibling of Y if they both have the same parent.
```

Furthermore add rules defining:

```
     "sister", "brother",
     "aunt", "uncle",
     "grandparent", "cousin"
```

```
====================================================================
```

3. Recursion: Towers of Hanoi

The 3-disk setup is like this:

```
           |        |         |
          xxx       |         |
         xxxxx      |         |
        xxxxxxx     |         |
     _________________________________
```

Here's a sample:

```
% move(N,X,Y,Z) - move N disks from peg X to peg Y, with peg Z being the
%                 auxilliary peg
%
% Strategy:
% Base Case: One disc - To transfer a stack consisting of 1 disc from
%    peg X to peg Y, simply move that disc from X to Y
% Recursive Case: To transfer n discs from X to Y, do the following:
         Transfer the first n-1 discs to some other peg X
         Move the last disc on X to Y
         Transfer the n-1 discs from X to peg Y

     move(1,X,Y,_) :-
         write('Move top disk from '),
         write(X),
         write(' to '),
         write(Y),
         nl.
     move(N,X,Y,Z) :-
         N>1,
         M is N-1,
         move(M,X,Z,Y),
         move(1,X,Y,_),
         move(M,Z,Y,X).
```

- note the use of "anonymous" variables _

Here is what happens when Prolog solves the case N=3.

```
     ?-  move(3,left,right,center).
     Move top disk from left to right
     Move top disk from left to center
     Move top disk from right to center
     Move top disk from left to right
     Move top disk from center to left
     Move top disk from center to right
     Move top disk from left to right

     yes
```

```
====================================================================
```

4. An example using lists:

(a) length of a list

```
size([],0).
size([H|T],N) :- size(T,N1), N is N1+1.
%  or size([_|T],N) :- size(T,N1), N is N1+1.

| ?- size([1,2,3,4],N).

N = 4

yes
| ?- size([bill,ted,ming,pascal,nat,ron],N).

N = 6
```

```
yes
| ?- size([a, [b, c, d], e, [f | g], h], N).

N = 5

yes


 (b) summing elements of a list of numbers

sumlist([],0).
ssumlist([H|T],N) :- sumlist(T,N1), N is N1+H.

 (c) list membership

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

 (d) reversing a list

reverse(List, Reversed) :-
          reverse(List, [], Reversed).

reverse([], Reversed, Reversed).
reverse([Head|Tail], SoFar, Reversed) :-
          reverse(Tail, [Head|SoFar], Reversed).

| ?- myreverse([a,b,c,d],X).

X = [d,c,b,a];     <- note semicolon (more solns?)

no

| ?- myreverse([a,b,c,d],[d,b,c,a]).

no
| ?- myreverse([a,b,c,d],[d,c,b,a]).

yes

- note difference between reverse/2 and reverse/3
- reverse/3 probably should be called reverseHelper or
  something else for clarity
```

`member(`*`Thing`*`, `*`List`*`)`

Succeeds if the *`Thing`* occurs in the *`List`*, fails otherwise.
Remember the declarative nature of Prolog. What will `member(X, [a, b, c])` do? or `member(abc, X)`?

`last(`*`NonemptyList`*`, `*`Thing`*`)`

`Thing` is the last value in the *`NonemptyList`*.

`last([1, 2, 3], X).` should unify `X` with `3`, and `last([1, 2, 3], 3).` should succeed without unifying anything. However, what do you think `last(X, 3).` would or should do? (Hint: There are multiple answers.)

`extend(List, Value, LongerList)`
LongerList is like *List*, but with one extra *Value* at the end. Or: *List* is like *LongerList*, but is missing the last *Value*.

`atomsof(List, ShorterList)`
ShorterList is like *List*, except that it has no sublists, only "atoms." (You can check if *X* is an atom with `atomic(X)`).

`setof(List, Set)`
Set is a set of all the things in *List*--that is, *Set* has no duplicated values.

`reverse(List1, List2)`
List1 and *List2* have the same top-level elements, but in the reverse order. Note: Sublists are not reversed, so`reverse([a,[b,c,d],e], [e,[b,c,d],a]).`

`reverseall(List1, List2)`
List1 and *List2* are complete reversals of each other. Sublists B reversed, so`reverseall([a,[b,cat,dog],e], [e,[dog,cat,b],a]).`

`remove(Thing, List,ShorterList)`
ShorterList is like *List*, except that it has no top-level occurences of *Thing* in it.

`removeall(Thing, List,ShorterList)`
ShorterList is like *List*, except that it has no occurences of *Thing* anywhere at all inside it, at any level.

`collapse(List, FlatList)`
FlatList contains all of the atoms of *List*, but the division into sublists has been discarded. That is, `[a,[b,c],d]` becomes `[a,b,c,d]`.

These predicates are roughly in order of increasing difficulty. Once you have written a predicate, use it in other predicates as appropriate--if you don't, some predicates will be very difficult!

I wrote `member`, `last`, `extend`, and `reverse` in two lines (two clauses) each. The other predicates each took three lines.

Hints:

1. **Use unification in the argument list (before the `:-`)!** For example, `member(H, [H | T]).` is simpler than something like `member(X, [H | T]) :- X = H.` That's not a big deal in this example, but if you get into the

habit of using "pattern matching" to select which clause to use, that will make this kind of programming a lot simpler. In fact, any time you are tempted to use `=`, you are probably making the predicate more difficult than it needs to be.

2. Don't forget to handle the simplest cases--empty lists and (for some predicates) plain atoms.
3. Prolog does backtracking, and most of these predicates, when backtracked into, will produce additional answers. The error is probably not in the predicate, but in something after it.
4. `extend`. You extend a list by extending the tail of the list, right? But don't forget about the empty list.
5. `reverse`. Don't forget that you can use functions you have already written (like `extend`).
6. `reverseall`. Atoms aren't reversible. The reverse of an atom is the same atom.
7. `setof`. Don't forget about the empty list.
8. `removeall`. You need a case for ordinary atoms.
9. `collapse`. Use `append`.
10. When all else fails, use `trace` and `notrace`.