KOMMURI PRATAP REDDY INSTITUTE OF TECHNOLOGY

(COLLEGE OF ENGINEERING)

PROLOG LAB MANUAL

B.TECH(II YEAR–II SEM) R22 REGULATION (Common for CSE, CSE(AI&ML))

2024-2025

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Aim:

1. **Write simple fact for following:A. Ram likes mango.B. Seema is a girl.C. Bill likes Cindy.D. Rose is red.E. John owns gold**

% Facts

1. Ram likes mango.

2. Seema is a girl.

3. Bill likes Cindy.

4. Rose is red.

5. John owns gold.

% Clauses

likes(ram ,mango).

girl(seema).

red(rose).

likes(bill ,cindy).

owns(john ,gold).

Output :

```
% Queries

 ?-likes(ram,What).
 What= mango
 ?-likes(Who,cindy).
 Who= cindy

 ?-red(What).
 What= rose
 ?-owns(Who,What).
 Who= john
 What= gold
```

2. Write predicates one converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

```
% Production rules:

c_to_f
f is c * 9 / 5 + 32
freezing f < = 32

% Rules:

c_to_f(C,F) :-
    F is C * 9 / 5 + 32.

freezing(F) :-
    F =< 32.
```

Quaries:

```
?- c_to_f(0, F).

F = 32.

?- freezing(30).

true.
```

How It Works

1. **`c_to_f(C, F)` Rule:**
   - Takes **Celsius (`C`)** as input.
   - Converts it to **Fahrenheit (`F`)** using the formula: F=C×95+32F = \frac{C \times 9}{5} + 32F=5C×9+32
2. **`freezing(F)` Rule:**
   - Checks if **Fahrenheit (`F`)** is **less than or equal to 32°F** (freezing point of water).

3. Write a program to solve the Monkey Banana problem

```
% Facts: Objects present in the room
in_room(bananas).
in_room(chair).
in_room(monkey).

% The monkey is intelligent
clever(monkey).

% The monkey can climb onto the chair
can_climb(monkey, chair).

% The chair is tall
tall(chair).

% The monkey can move the chair under the bananas
can_move(monkey, chair, bananas).

% Rules:
% The monkey can reach the bananas if it is clever and close to them
can_reach(X, Y) :-
   clever(X),
   close(X, Y).

% The monkey can get on something if it can climb
get_on(X, Y) :-
   can_climb(X, Y).

% The chair is under the bananas
under(Y, Z) :-
   in_room(Y),
   in_room(Z),
   can_move(monkey, Y, Z).

% The monkey is close to the bananas if it is on the chair and the chair is under the bananas
close(X, Z) :-
   get_on(X, Y),
   under(Y, Z).

% The monkey is also close to the bananas if the chair is tall enough
close(X, Z) :-
   tall(Y),
   in_room(Y).
```

Example Queries in Prolog

1. **Check if the monkey can reach the bananas:**

```
?- can_reach(monkey, bananas).
true.
```

2. **Check if the monkey can get on the chair:**

```
?- get_on(monkey, chair).
true.
```

3. **Check if the chair is under the bananas:**

```
?- under(chair, bananas).
true.
```

4. **WAP in turbo prolog for medical diagnosis and show the advantages and disadvantages of green and red cuts.**

```
% Tower of Hanoi Solution

% Rule to solve Tower of Hanoi
hanoi(N) :-
   move(N, left, middle, right).

% Base case: Move a single disk directly
move(1, A, _, C) :-
   inform(A, C), !.

% Recursive case: Move N disks
move(N, A, B, C) :-
   N1 is N - 1,  % Compute N-1
   move(N1, A, C, B), % Move N-1 disks to auxiliary peg
   inform(A, C), % Move the largest disk to the target peg
   move(N1, B, A, C). % Move the remaining N-1 disks to target peg

% Print the movement
inform(Loc1, Loc2) :-
   write("Move a disk from "), write(Loc1), write(" to "), write(Loc2), nl.
```

How It Works :
hanoi(N) starts the process for N disks.

move/4 follows the recursive logic:

Move N-1 disks from the source peg (A) to auxiliary peg (B).
Move the largest disk from source peg (A) to the target peg (C).
Move N-1 disks from auxiliary peg (B) to target peg (C).
inform/2 correctly prints the movement.

Example Queries in Prolog

```
?- hanoi(3).
Move a disk from left to right
Move a disk from left to middle
Move a disk from right to middle
Move a disk from left to right
Move a disk from middle to left
Move a disk from middle to right
Move a disk from left to right
true.
```

5. Write a program to solve 4-Queens problem

```
/* Description:

In the 4 Queens problem the object is to place 4 queens on a chessboar
   d in such a way that no queens can capture a piece. This means that
   no two queens may be placed on the same row, column, or diagonal.

*/
```

```
% Solve the N-Queens problem
nqueens(N) :-
   makelist(N, L),
   Diagonal is N * 2 - 1,
   makelist(Diagonal, LL),
   placeN(N, board([], L, L, LL, LL), Final),
   display_board(Final).

% Base case: All queens are placed
placeN(_, board(D, [], [], D1, D2), board(D, [], [], D1, D2)) :- !.

% Recursive case: Place a queen on the board
placeN(N, Board1, Result) :-
   place_a_queen(N, Board1, Board2),
   placeN(N, Board2, Result).

% Place a single queen
place_a_queen(N, board(Queens, Rows, Columns, Diag1, Diag2),
              board([q(R, C) | Queens], NewR, NewC, NewD1, NewD2)) :-
   nextrow(R, Rows, NewR),
   findandremove(C, Columns, NewC),
   D1 is N + C - R, findandremove(D1, Diag1, NewD1),
   D2 is R + C - 1, findandremove(D2, Diag2, NewD2).

% Remove an element from a list
findandremove(X, [X | Rest], Rest).
findandremove(X, [Y | Rest], [Y | Tail]) :-
   findandremove(X, Rest, Tail).

% Create a list from 1 to N
makelist(1, [1]).
makelist(N, [N | Rest]) :-
   N1 is N - 1,
   makelist(N1, Rest).
```

```
% Get the next row from the list
nextrow(Row, [Row | Rest], Rest).

% Display the board neatly
display_board(board(Queens, _, _, _, _)) :-
    write("Solution: "), nl,
    print_queens(Queens).

print_queens([]).
print_queens([q(R, C) | Rest]) :-
    format("Queen at Row ~w, Column ~w~n", [R, C]),
    print_queens(Rest).
```

Example Query and Output

```
?- nqueens(4).
Queen at Row 2, Column 4
Queen at Row 4, Column 2
Queen at Row 1, Column 3
Queen at Row 3, Column 1
true.
```

6. Write a program to solve traveling salesman problems.

```
% Road connections between towns with distances

% Facts: Direct road connections
road(tampa, houston, 200).
road(gordon, tampa, 300).
road(houston, gordon, 100).
road(houston, kansas_city, 120).
road(gordon, kansas_city, 130).

% Rules: Define route using direct and indirect connections
route(Town1, Town2, Distance) :-
   road(Town1, Town2, Distance).

route(Town1, Town2, Distance) :-
   road(Town1, X, Dist1),
   route(X, Town2, Dist2),
   Distance is Dist1 + Dist2.
```

<u>Example Queries and Outputs</u>

```
?- route(tampa, houston, D).
D = 200.
?- route(tampa, kansas_city, D).
D = 320 ;   % (Tampa -> Houston -> Kansas City)
D = 430.    % (Tampa -> Gordon -> Kansas City)
```

7. Write a program to solve water jug problems using Prolog.

```
:- dynamic visited_state/2.

% Water Jug Problem Solution

% State representation: (X, Y) where:
% X = Amount of water in 4-Gallon jug
% Y = Amount of water in 3-Gallon jug

state(2, 0) :-
    write("Goal state reached: (2,0)~n"),
    !.  % Stop further exploration

state(X, Y) :-
    X < 4,  % Rule R1: Fill 4-Gallon jug if not full
    not(visited_state(4, Y)),
    assert(visited_state(X, Y)),
    format("Fill the 4-Gallon Jug: (~w,~w) --> (4,~w)~n", [X, Y, Y]),
    state(4, Y).

state(X, Y) :-
    Y < 3,  % Rule R2: Fill 3-Gallon jug if not full
    not(visited_state(X, 3)),
    assert(visited_state(X, Y)),
    format("Fill the 3-Gallon Jug: (~w,~w) --> (~w,3)~n", [X, Y, X]),
    state(X, 3).

state(X, Y) :-
    X > 0,  % Rule R5: Empty the 4-Gallon jug
    not(visited_state(0, Y)),
    assert(visited_state(X, Y)),
    format("Empty the 4-Gallon Jug: (~w,~w) --> (0,~w)~n", [X, Y, Y]),
    state(0, Y).

state(X, Y) :-
    Y > 0,  % Rule R6: Empty the 3-Gallon jug
    not(visited_state(X, 0)),
    assert(visited_state(X, Y)),
    format("Empty the 3-Gallon Jug: (~w,~w) --> (~w,0)~n", [X, Y, X]),
    state(X, 0).

state(X, Y) :-
    X + Y >= 4, Y > 0,  % Rule R7: Transfer from 3G to 4G until full
    NEW_Y is Y - (4 - X),
    not(visited_state(4, NEW_Y)),
    assert(visited_state(X, Y)),
    format("Pour from 3G to 4G until full: (~w,~w) --> (4,~w)~n", [X, Y, NEW_Y]),
    state(4, NEW_Y).

state(X, Y) :-
    X + Y >= 3, X > 0,  % Rule R8: Transfer from 4G to 3G until full
    NEW_X is X - (3 - Y),
    not(visited_state(NEW_X, 3)),
    assert(visited_state(X, Y)),
    format("Pour from 4G to 3G until full: (~w,~w) --> (~w,3)~n", [X, Y, NEW_X]),
    state(NEW_X, 3).

state(X, Y) :-
    X + Y =< 4, Y > 0,  % Rule R9: Pour all from 3G to 4G
    NEW_X is X + Y,
    not(visited_state(NEW_X, 0)),
    assert(visited_state(X, Y)),
    format("Pour all from 3G to 4G: (~w,~w) --> (~w,0)~n", [X, Y, NEW_X]),
    state(NEW_X, 0).

state(X, Y) :-
    X + Y =< 3, X > 0,  % Rule R10: Pour all from 4G to 3G
    NEW_Y is X + Y,
    not(visited_state(0, NEW_Y)),
    assert(visited_state(X, Y)),
    format("Pour all from 4G to 3G: (~w,~w) --> (0,~w)~n", [X, Y, NEW_Y]),
    state(0, NEW_Y).


% Start solving from initial state (0,0)
start :-
    retractall(visited_state(_, _)), % Clear visited states before running
    state(0, 0).
```

How to Run the Program

Load the file into SWI-Prolog or any Prolog interpreter, and run the following query:

```
?- start.
```

Expected Output :

```
Fill the 3-Gallon Jug: (0,0) --> (0,3)
Pour from 3G to 4G until full: (0,3) --> (3,0)
```

```
Fill the 3-Gallon Jug: (3,0) --> (3,3)
Pour from 3G to 4G until full: (3,3) --> (4,2)
Empty the 4-Gallon Jug: (4,2) --> (0,2)
Pour 2 gallons from 3G to 4G: (0,2) --> (2,0)
Goal state reached: (2,0)
```

8. **Write simple Prolog functions such as the following. Take into account lists which are too short.-- remove the Nth item from the list. -- insert as the Nth item.**

```
 % Remove the Nth item from a list
   remove_nth(1, [_|Tail], Tail) :- !.  % Remove first element
   remove_nth(N, [Head|Tail], [Head|Result]) :-
      N > 1,
      N1 is N - 1,
      remove_nth(N1, Tail, Result).
   remove_nth(_, List, List).  % If N is out of bounds, return original list

   % Insert an element at the Nth position in a list
   insert_nth(1, Element, List, [Element|List]) :- !.  % Insert at the beginning
   insert_nth(N, Element, [Head|Tail], [Head|Result]) :-
      N > 1,
      N1 is N - 1,
      insert_nth(N1, Element, Tail, Result).
insert_nth(_, Element, [], [Element]).  % If list is too short, add at the end
```

__Example Queries__

```
?-start.
```

9. Assume the prolog predicate gt(A, B) is true when A is greater than B. Use this predicate to define the predicate addLeaf(Tree, X, NewTree) which is true if NewTree is the Tree producedby adding the item X in a leaf node. Tree and NewTree are binary search trees. The empty treeis represented by the atom nil.

```
% Base case: Adding X to an empty tree creates a new leaf node
addLeaf(nil, X, tree(X, nil, nil)).

% If X is greater than Root, insert into the right subtree
addLeaf(tree(Root, Left, Right), X, tree(Root, Left, NewRight)) :-
   gt(X, Root),
   addLeaf(Right, X, NewRight).

% If X is less than or equal to Root, insert into the left subtree
addLeaf(tree(Root, Left, Right), X, tree(Root, NewLeft, Right)) :-
   \+ gt(X, Root),
   addLeaf(Left, X, NewLeft).

% Predicate gt(A, B) is true if A is greater than B
  gt(A, B) :- A > B.
```

Example Queries

```
?- addLeaf(nil, 5, Tree).
Tree = tree(5, nil, nil).  % Insert into an empty tree

?- addLeaf(tree(10, tree(5, nil, nil), tree(15, nil, nil)), 12, NewTree).
NewTree = tree(10, tree(5, nil, nil), tree(15, tree(12, nil, nil), nil)).

?- addLeaf(tree(10, tree(5, nil, nil), tree(15, nil, nil)), 3, NewTree).
NewTree = tree(10, tree(5, tree(3, nil, nil), nil), tree(15, nil, nil)).
```

10. **Write a Prolog predicate, countLists(Alist, Ne, Nl), using accumulators, that is true when Nl is the number of items that are listed at the top level of Alist and Ne is the number of empty lists. Suggestion: First try to count the lists, or empty lists, then modify by adding the other counter.**

```
% Base case: When the list is empty, both counts remain unchanged
countLists([], Ne, Nl) :- countLists([], 0, 0, Ne, Nl).

% Recursive case: Process each element while keeping track of counts
countLists([H|T], AccE, AccL, Ne, Nl) :-
  is_list(H),        % Check if H is a list
  ( H = [] ->        % Check if it's an empty list
     NewAccE is AccE + 1  % Increment empty list counter
  ;
     NewAccE is AccE  % Keep empty list counter unchanged
  ),
  NewAccL is AccL + 1, % Increment list counter
  countLists(T, NewAccE, NewAccL, Ne, Nl).

% Recursive case: If the element is not a list, continue processing
countLists([_|T], AccE, AccL, Ne, Nl) :-
  countLists(T, AccE, AccL, Ne, Nl).

% Base case for the accumulator version
countLists([], Ne, Nl, Ne, Nl).
```

11. Define a predicate memCount(AList,Blist,Count) that is true if Alist occurs Count times within Blist. Define without using an accumulator. Use "not" as defined in utilities.pro, to make similar cases are unique, or else you may get more than one count as an answer.

```
Examples:
memCount(a,[b,a],N).
N = 1 ;
no
memCount(a,[b,[a,a,[a],c],a],N).
N = 4 ;
no
memCount([a],[b,[a,a,[a],c],a],N).
N = 1 ;
No
% Base case: An empty list contains nothing.
memCount(_, [], 0).

% If the head of BList matches AList, increase count and check the tail.
memCount(AList, [AList|Tail], Count) :-
   memCount(AList, Tail, Count1),
   Count is Count1 + 1.

% If the head is a list, recursively check inside it and also in the tail.
memCount(AList, [Head|Tail], Count) :-
   is_list(Head),
   memCount(AList, Head, Count1),
   memCount(AList, Tail, Count2),
   Count is Count1 + Count2.

% If the head does not match AList, continue checking in the tail.
memCount(AList, [_|Tail], Count) :-
   memCount(AList, Tail, Count).
```

Example Queries:

```
?- memCount(a, [b, a], N).
N = 1.

?- memCount(a, [b, [a, a, [a], c], a], N).
N = 4.
?- memCount([a], [b, [a, a, [a], c], a], N).
N = 1.
```