Artificial Intelligence Lab using prolog Programming

(KCS 751A)

Department of Computer Science & Engineering
United College of Engineering & Research, Allahabad
Dr. A.P.J. Abdul Kalam Technical University

A-31, UPSIDC, Industrial Area, Naini Prayagraj
Website: www.united.ac.in

Name	--
Roll No.	---
Branch	------------------------------**Semester**------------------------
Session	**ODD 2022 -2023**

INDEX				
S.NO.	PROGRAM	DATE	GRADE	SIGN.
1.	Study of Prolog.			
2.	Write simple facts for the Statement using PROLOG.			
3.	Write predicates One converts centigrade temperature to Fahrenheit, the other checks if the temperature is below freezing.			
4.	Write a program to implement factorial and Fibonacci of a given number.			
5.	Write a program to display Head/Tail in given list.			
6.	Write a program to implement Union and Intersection of a given number.			
7.	Write a program to solve Tower of Hanoi problem.			
8.	Write to solve Monkey Banana Problem.			
9.	Write to solve 4-Queen Problem.			
10.	Write a program to solve traveling salesman problem.			

EXPERIMENT NO. 1

OBJECTIVE: Study of Prolog.

PROLOG-PROGRAMMING IN LOGIC

PROLOG stands for Programming, In Logic — an idea that emerged in the early 1970’s to use logic as programming language. The early developers of this idea included Robert Kowaiski at Edinburgh (on the theoretical side), Marrten van Emden at Edinburgh (experimental demonstration) and Alian Colmerauer at Marseilles (implementation).
David D.H. Warren’s efficient implementation at Edinburgh in the mid -1970’s greatly contributed to the popularity of PROLOG. PROLOG is a programming language centred on a small set of basic mechanisms, Including pattern matching, tree based data structuring and automatic backtracking. This Small set constitutes a surprisingly powerful and flexible programming framework. PROLOG is especially well suited for problems that involve objects- in particular, structured objects- and relations between them.

SYMBOLIC LANGUAGE
PROLOG is a programming language for symbolic, non-numeric computation. It is especially well suited for solving problems that involve objects and relations between objects.
For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y. and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language for ArtJIcia1 LanguageA1,) and non-numerical programming.
There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code, when the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.

FACTS, RULES AND QUERIES
Programming in PROIOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can be posed about the objects and valid conclusions will be determined and returned by the program Responses to user queries are determined through a form of inference control known as resolution.

FOR EXAIPLE:
a) **FACTS:**
Some facts about family relationships could be written as:
sister(sue,bill)
parent(ann.sam)
male(jo)
female(riya)

B)**Rules:** To represent the general rule for grandfather, we write: grand f.gher(X2) parent(X,Y) parent(Y,Z) male(X)

c) **QUERIES:** Given a database of facts and rules such as that above, we may make queries by typing after a query a symbol’?’ statements such as: ?-parent(X,sam) Xann ?grandfather(X,Y) X=jo, Y=sam

META PROGRAMMING: A meta-program is a program that takes other programs as data. Interpreters and compilers are examples of mela-programs. Meta-interpreter is a particular kind of meta-program: an interpreter for a language written in that language. So a PROLOG interpreter is an interpreter for PROLOG, itself written in PROLOG. Due to its symbol- manipulation capabilities, PROLOG is a powerful language for meta-programming. Therefore, it is often used as an implementation language for other languages. PROLOG is particularly suitable as a language for rapid prototyping where we are interested in implementing new ideas quickly. New ideas are rapidly implemented and experimented with.

OUTCOME: Students will get the basic idea of how to program in prolog and its working environment.

<u>EXPERIMENT NO. 2</u>

OBJECTIVE: 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.

Program:

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.
```

OUTCOME: Student will understand how to write simple facts using prolog.

EXPERIMENT NO. 3

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

Program:

Production rules:

Arithmetic:
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.
```

Output:
Queries:

```
?- c_to_f(100,X).
X = 212
Yes
?- freezing(15)
.Yes
?- freezing(45).
No
```

OUTCOME: Student will understand how to write a program using the rules.

EXPERIMENT NO. 4

OBJECTIVE: WAP to implement factorial, fibonacci of a given number.

Program:

Factorial:

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

Output:
Goal:

```
?- factorial(4,X).
X=24
```

Fibonacci:

```
fib(0, 0).
fib(X, Y) :- X > 0, fib(X, Y, _).
fib(1, 1, 0).
fib(X, Y1, Y2) :-
X > 1,
X1 is X - 1,
fib(X1, Y2, Y3),
Y1 is Y2 + Y3.
```

Output:
Goal:

```
?-fib(10,X).
X=55
```

OUTCOME: Student will understand the implementation of Fibonacci and factorial series using prolog.

EXPERIMENT NO. 5

Objective: WAP to display Head/Tail in a given list.

Program:

[H/T]=[1,2,3,4]
Head H=1
Tail T=2,3,4,5

%Multiple Head

[H1,H2/T]=[1,2,3,4,5]
H1=1,H2=2 and T=[3,4,5]

Head and Tail of the lists :
head : the 1st element of the list.
tail : all elements of the list except the 1st one
Syntax : [H | T]

For example :
(1) In list [1,2,3,4] head is 1 and tail is [2,3,4]
(2) In list [a] head is a and tail is [].
(3) In list [likes(john, mary), X , 1, 2] head is likes(john, mary) and tail is [X, 1, 2].
(4) In list [A, [p, n, c], 4] head is A and tail is [[p, n, c],4]
(5) In list [[1, 2], a, b] head is [1, 2] and tail is [a, b]
Note :

In above examples tails are always a list.
[H | T] = [1, 2, 3, 4], H is 1 and T is [2, 3, 4] = [1 | [2, 3, 4]]
Valid:
[1 | [2, 3]]
[[1,2] | [3, 4]]
Invalid:
[1, 2 | 3]

EXPERIMENT NO. 6

Objective: Write a program to implement Union and Intersection of a given number.

Program:

Head and Tail of the lists :
head : the 1st element of the list.
tail : all elements of the list except the 1st one
Syntax : [H | T]

Union Operation:

Program

```
list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).
list_union([X|Y],Z,W) :- list_member(X,Z),list_union(Y,Z,W).
list_union([X|Y],Z,[X|W]) :- \+ list_member(X,Z), list_union(Y,Z,W).
list_union([],Z,Z).
```

Output:

```
?- list_union([a,b,c,d,e],[a,e,i,o,u],L3).
L3 = [b,c,d,a,e,i,o,u] ?
(16 ms) yes
| ?- list_union([a,b,c,d,e],[1,2],L3).
L3 = [a,b,c,d,e,1,2]
yes
```

Intersection Operation:

```
list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).
list_intersect([X|Y],Z,[X|W]) :- list_member(X,Z), list_intersect(Y,Z,W).
list_intersect([X|Y],Z,W) :- \+ list_member(X,Z), list_intersect(Y,Z,W).
list_intersect([],Z,[]).
```

Output:

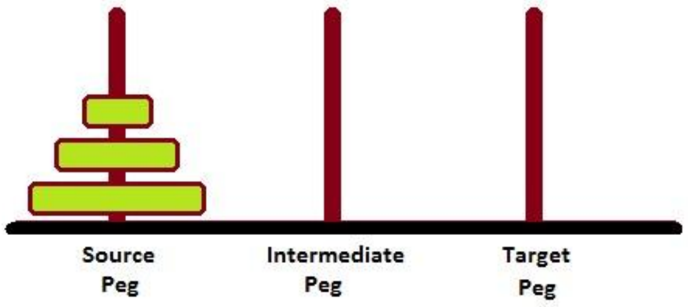
```
?- list_intersect([a,b,c,d,e],[a,e,i,o,u],L3).
L3 = [a,e] ?
yes

| ?- list_intersect([a,b,c,d,e],[],L3).
L3 = []
yes
```

EXPERIMENT NO. 7

Objective: Write a program to solve Tower of Hanoi problem.

Towers of Hanoi Problem is a famous puzzle to move N disks from the source peg/tower to the target peg/tower using the intermediate peg as an auxiliary holding peg. There are two conditions that are to be followed while solving this problem –

A larger disk cannot be placed on a smaller disk.
Only one disk can be moved at a time.
The following diagram depicts the starting setup for N=3 disks.



Hanoi Problem

To solve this, we have to write one procedure move(N, Source, Target, auxiliary). Here N number of disks will have to be shifted from Source peg to Target peg keeping Auxiliary peg as intermediate.

For example – move(3, source, target, auxiliary).

Move top disk from source to target
Move top disk from source to auxiliary
Move top disk from target to auxiliary
Move top disk from source to target
Move top disk from auxiliary to source
Move top disk from auxiliary to target
Move top disk from source to target

Program

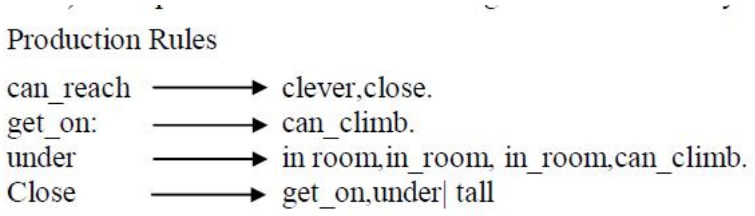
```
move(1,X,Y,_) :-
  write('Move top disk from '), write(X), write(' to '), write(Y), nl.
move(N,X,Y,Z) :-
  N>1,
  M is N-1,
  move(M,X,Z,Y),
  move(1,X,Y,_),
  move(M,Z,Y,X).
```

Output: move (4,source,target,auxiliary).

<u>**EXPERIMENT NO. 8**</u>

Objective: Write to solve Monkey Banana Problem.

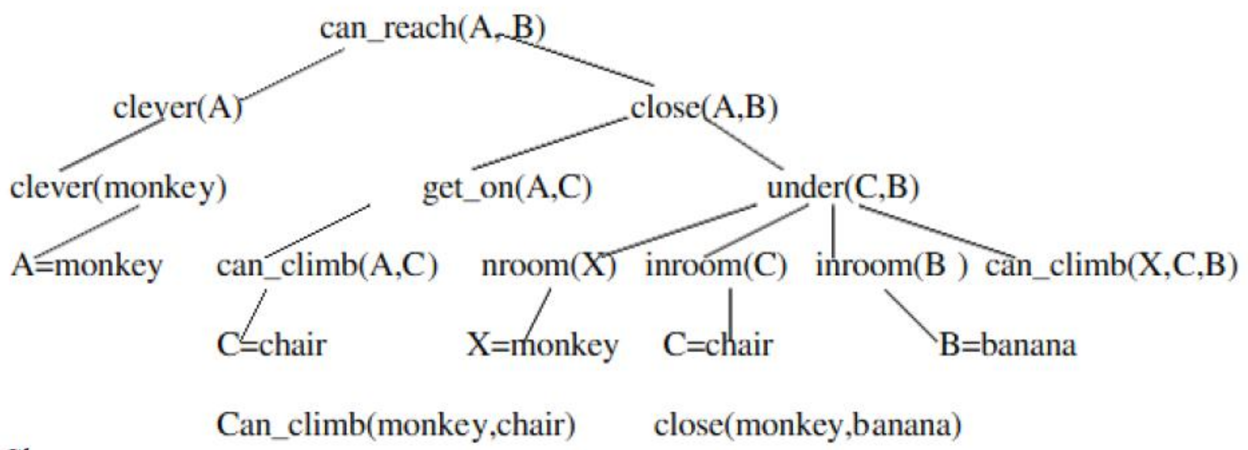
Imagine a room containing a monkey, chair and some bananas. That have been hanged from the centre of ceiling. If the monkey is clever enough he can reach the bananas by placing the chair directly below the bananas and climb on the chair .The problem is to prove the monkey can reach the bananas.The monkey wants it, but cannot jump high enough from the floor. At the window of the room there is a box that the monkey can use. The monkey can perform the following actions:-
1) Walk on the floor.
2) Climb the box.
3) Push the box around (if it is beside the box).
4) Grasp the banana if it is standing on the box directly under the banana.



Production Rules

can_reach $\longrightarrow$ clever,close.
get_on: $\longrightarrow$ can_climb.
under $\longrightarrow$ in room,in_room, in_room,can_climb.
Close $\longrightarrow$ get_on,under| tall

Parse Tree



Clauses:
in_room(bananas).
in_room(chair).
in_room(monkey).
clever(monkey).
can_climb(monkey, chair).
tall(chair).
can_move(monkey, chair, bananas).

can_reach(X, Y):-
clever(X),close(X, Y).
get_on(X,Y):- can_climb(X,Y).
under(Y,Z):-

```
in_room(X),in_room(Y),in_room(Z),can_climb(X,Y,Z).
close(X,Z):-get_on(X,Y),
under(Y,Z);
tall(Y).
```

Output:
Queries:

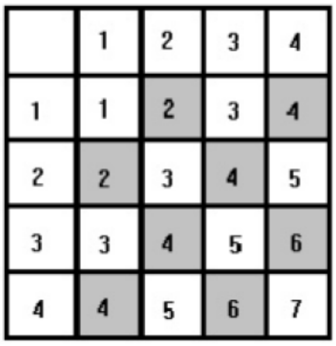
```
?- can_reach(A, B).
A = monkey.
B = banana.
?- can_reach(monkey, banana).Yes.
```

OUTCOME: Student will understand how to solve monkey banana problem using rules in prolog.

EXPERIMENT NO. 9

OBJECTIVE: Write a program to solve 4-Queen problem.

Program:
In the 4 Queens problem the object is to place 4 queens on a chessboard 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.



	1	2	3	4
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6
4	4	5	6	7

The n Queens Chessboard.

```
domains
queen = q(integer, integer)
queens = queen*
freelist = integer*
board = board(queens, freelist, freelist, freelist, freelist)
predicates
nondeterm placeN(integer, board, board)
nondeterm place_a_queen(integer, board, board)
nondeterm nqueens(integer)
nondeterm makelist(integer, freelist)
nondeterm findandremove(integer, freelist, freelist)
nextrow(integer, freelist, freelist)
clauses
nqueens(N):-
makelist(N,L),
Diagonal=N*2-1,
makelist(Diagonal,LL),
placeN(N,board([],L,L,LL,LL),Final),
write(Final).
placeN(_,board(D,[],[],D1,D2),board(D,[],[],D1,D2)):-!.

placeN(N,Board1,Result):-
place_a_queen(N,Board1,Board2),
placeN(N,Board2,Result).
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=N+C-R,findandremove(D1,Diag1,NewD1),
D2=R+C-1,findandremove(D2,Diag2,NewD2).
findandremove(X,[X|Rest],Rest).
findandremove(X,[Y|Rest],[Y|Tail]):-
findandremove(X,Rest,Tail).
makelist(1,[1]).
makelist(N,[N|Rest]) :-
N1=N-1,makelist(N1,Rest).
nextrow(Row,[Row|Rest],Rest).
```

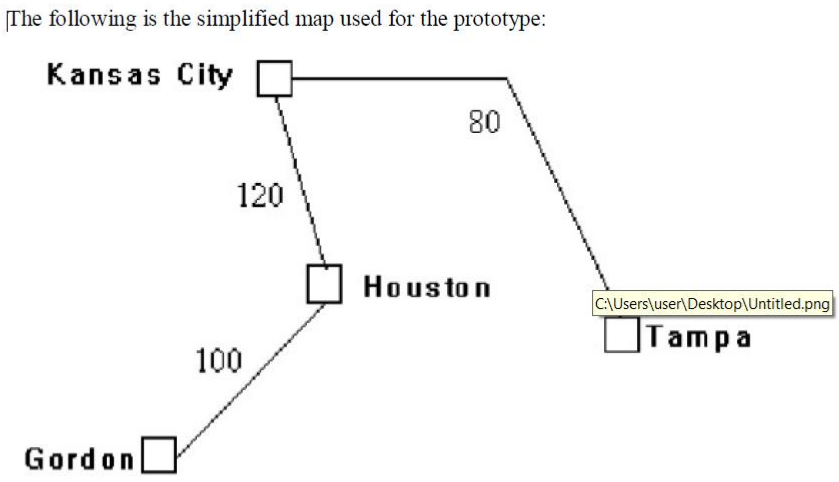
Output:
Goal:

```
?-nqueens(4),nl.
board([q(1,2),q(2,4),q(3,1),q(4,3),[],[],[7,4,1],[7,4,1])
yes
```

OUTCOME: Student will implement 4-Queen problem using prolog.

EXPERIMENT NO. 10

OBJECTIVE: Write a program to solve traveling salesman problem.


The following is the simplified map used for the prototype:
Kansas City
80
120
Houston
C:\Users\user\Desktop\Untitled.png
Tampa
100
Gordon

Program:

Production Rules:-

route(Town1,Town2,Distance) road(Town1,Town2,Distance).
route(Town1,Town2,Distance) road(Town1,X,Dist1),route(X,Town2,Dist2),Distance=Dist1+Dist2,

```
domains
town = symbol
distance = integer
predicates
nondeterm road(town,town,distance)
nondeterm route(town,town,distance)
```

clauses

```
road("tampa","houston",200).
road("gordon","tampa",300).
road("houston","gordon",100).
road("houston","kansas_city",120).
road("gordon","kansas_city",130).
route(Town1,Town2,Distance):-
road(Town1,Town2,Distance).
route(Town1,Town2,Distance):-
road(Town1,X,Dist1),
route(X,Town2,Dist2),
Distance=Dist1+Dist2,!.
```

Output:

Goal:

```
route("tampa", "kansas_city", X),
write("Distance from Tampa to Kansas City is ",X),nl.
Distance from Tampa to Kansas City is 320
X=320
```

OUTCOME: Student will implement travelling salesmen problem using prolog.