Artificial Intelligence Lab (BTCOL 706)

Experiment No – 01

Aim: Study of PROLOG.

Theory:

1. Prolog stands for programming in logic. In the logic programming paradigm, prolog language is most widely available. Prolog is a declarative language, which means that a program consists of data based on the facts and rules (Logical relationship) rather than computing how to find a solution. A logical relationship describes the relationships which hold for the given application.
2. To obtain the solution, the user asks a question rather than running a program. When a user asks a question, then to determine the answer, the run time system searches through the database of facts and rules.
3. Prolog - PROgramming in LOGic, was first developed by Alain Colmerauer and Philippe Roussel in 1972 - A Logic Programming language based on the predicate logic.
4. In prolog, clauses are actually descriptive statements that specify what is true about the problem and because of that Prolog is also known as declarative language or rule-based language.

Basically, in prolog program we write the program statements in terms of facts and rules. The system reads in the program and stores it. Upon asking the questions (known as queries) the system gives the answer by searching through the possible solution(s).

Some basic features of Prolog include:

- Pattern-matching mechanism
- Backtracking strategy that searches for possible solutions
- Uniform data structures from which programs are built

Prolog is also widely used for AI programs especially experts systems. For Prolog programming SWI-Prolog provides comprehensive prolog environment and it is also free.

Applications of Prolog

The applications of prolog are as follows:

- Specification Language
- Robot Planning
- Natural language understanding
- Machine Learning
- Problem Solving
- Intelligent Database retrieval
- Expert System
- Automated Reasoning

In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it. However, in Prolog we are supposed to give clues as the solution method.

Prolog language basically has three different elements −

Facts − The fact is predicate that is true, for example, if we say, "Tom is the son of Jack", then this is a fact. Facts are those statements that state the objects or describe the relationship between objects. For an instance when we say john likes piano, we are showing the 'like' relationship between two objects 'john and piano' and in prolog this fact can be written as likes(john,piano). Facts are also known as "unconditional horn clauses" and they are always true.

Syntax

The syntax for facts is as follows −

relation(object1,object2...).

Examples:

logic_programming. // Read as : logic programming

music_student(john). // Read as : john is a music student

likes('John', car(bmw)) // Read as : john likes bmw car

gives(john, chocolate, jane). // Read as : john gives chocolate to jane

Rules − Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met.

For example, if we define a rule as −

grandfather(X, Y) :- father(X, Z), parent(Z, Y)

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.

Examples: If we want to say that john and jane are friends if john likes jane and jane likes john. Then in prolog this friends rule can be written as,

friends(john,jane) :- likes(john,jane), likes(jane,john).

Examples: Rules with variables.

likes(john, X) :- car(X). // Read as : john likes X if X is a car.

friends(X, Y) :- likes(X, Y), likes(Y, X). // Read as : X and Y are friends if X likes Y and Y likes X. OR Two people are friends if they like each other.

Questions − And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules. Prolog queries are the questions asked by the user to prolog interpreter about facts and rules stored in it's database. For the particular program, upon asking the query, Prolog interpreter trace through the facts and rules it has been given in the top-down manner and try to find a match for a given query. If it finds the matches it will report

yes/true and if it doesn't then it will report no/false. Query can be made up of multiple subgoals - for example ?- student(john, music), student(jane,music) - so here in this sentence we have query with two subgoals and subgoals must be separated by commas. But all subgoals must be satisfied otherwise whole query would result in failure. One more thing is - if you use any variables to define rules in prolog program, it's not necessary to use the same variable name when you ask queries.

Example : For below Prolog Program, we will ask two queries about the facts we have provided.

Program :

likes(alice,john). // Read as : alice likes john

likes(john,mary). // Read as : john likes mary

Queries : ?- likes(alice,john). // Read as :- Does alice like john?

true.

?- likes(alice,mary). // Read as :- Does alice like mary?

false.

In above example, text in red color is what user types and queries are terminated by the full stop at the end.

So, when we enter ?- likes(alice,john). interpreter will answer yes or true since it can find the match for the given query.

But when we ask ?- likes(alice,mary). then it would say no or false.

Prolog Syntax

The syntax of Prolog is as follows:

Symbols

Using the following truth-functional symbols, the Prolog expressions are comprised. These symbols have the same interpretation as in the predicate calculus.

English	Predicate Calculus	Prolog
If	-->	:-
Not	~	Not
Or	V	;
and	^	,

Variable

Variable is a string. The string can be a combination of lower case or upper case letters. The string can also contain underscore characters that begin with an underscore or an upper-case letter. Rules for forming names and predicate calculus are the same.

Handling input and output

The write() Predicate

To write the output we can use the write() predicate. This predicate takes the parameter as input, and writes the content into the console by default. write() can also write in files. Let us see some examples of write() function.

Example:

?- write(56).

56

yes

| ?- write('hello').

hello

yes

| ?- write('hello'),nl,write('world').

hello

world

The read() Predicate

The read() predicate is used to read from console. User can write something in the console, that can be taken as input and process it. The read() is generally used to read from console, but this can also be used to read from files. Now let us see one example to see how read() works.

Program

```
cube :-
   write('Write a number: '),
   read(Number),
   process(Number).
process(stop) :- !.
process(Number) :-
   C is Number * Number * Number,
   write('Cube of '),write(Number),write(': '),write(C),nl, cube.
```

Output

```
| ?- [read_write].
compiling D:/TP Prolog/Sample_Codes/read_write.pl for byte code...
D:/TP Prolog/Sample_Codes/read_write.pl compiled, 9 lines read - 1226 bytes written, 12 ms

(15 ms) yes
| ?- cube.
Write a number: 2.
Cube of 2: 8
Write a number: 10.
Cube of 10: 1000
Write a number: 12.
Cube of 12: 1728
Write a number: 8.
Cube of 8: 512
Write a number: stop
.
```

Program: Write a following program and take print and attached

1. Write a Prolog in Prolog calculate addition of two no.

2. Write a Prolog in Prolog to find Maximum of two no .
3. Write a Prolog in Prolog that take number N from the user and count from N to 10.
4. Write a Prolog in Prolog that take number N from the user and count from N to 1.
5. Write a Prolog in Prolog that take number N from the user calculate factorial of no.
6. Write a Prolog in Prolog that take number N from the user calculate square of no from N to 20 and display it .

Questions:

1. What is Prolog. Explain it .
2. Explain Loop and condition, decision making statement with example.
3. Explain List and its operation with example.
4. What is backtracking explain with example.

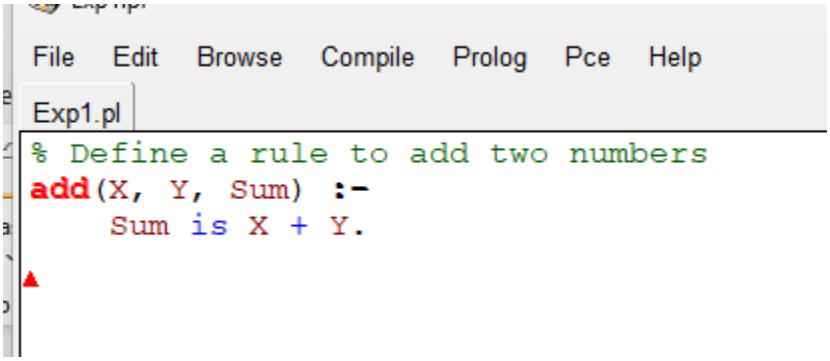
.

(Subject In-charge)

(Prof.S.B.Mehta)

---

1. Write a Prolog in Prolog calculate addition of two no:




```
?-
% c:/users/utkarsh/documents/prolog/exp1 compiled 0.00 sec, -1 clauses
?- add(5, 7, Result).
Result = 12.
```

2. Write a Prolog in Prolog to find Maximum of two no:

```
% Define a rule to find the maximum of two numbers
max(X, Y, Max) :-
    (X >= Y -> Max = X; Max = Y).
```

```
% c:/users/utkarsh/documents/prolog/exp1 compiled 0.00 sec, 0 clauses
?- max(10, 20, Result).
Result = 20.
```

3. Write a Prolog in Prolog that take number N from the user and count from N to 10:

```
% Define a rule to count from N to 10
count_to_ten(N) :-
    N =< 10,
    write(N), nl,
    N1 is N + 1,
    count_to_ten(N1).


count_to_ten(N) :-
    N > 10.
```

```
?- count_to_ten(7).
7
8
9
10
true
```

4. Write a Prolog in Prolog that take number N from the user and count from N to 1.

```
% Define a rule to count from N down to 1
count_down(N) :-
    N >= 1,
    write(N), nl,
    N1 is N - 1,
    count_down(N1).


count_down(N) :-
    N < 1.
```

```
% c:/users/utkarsh/documents/prolog/exp1 compiled 0.00 sec, 0 clauses
?-
|    count_down(5).
5
4
3
2
1
true
```

5. Write a Prolog in Prolog that take number N from the user calculate factorial of no.

```
% Define a rule to calculate the factorial of N
factorial(0, 1).                    % Base case: factorial of 0 is 1
factorial(N, Result) :-
    N > 0,                          % Ensure N is positive
    N1 is N - 1,                    % Decrement N by 1
    factorial(N1, TempResult),      % Recursively calculate the facto
    Result is N * TempResult.       % Multiply N by the factorial of N
```

```
% c:/users/utkarsh/documents/prolog/exp1 compiled 0.00 sec, 0 clauses
|    factorial(5, Result).
Result = 120
```

6. Write a Prolog in Prolog that take number N from the user calculate square of no from N to 20 and display it:

```
% Define a rule to calculate and display the square of numbers from N to 20
square_to_twenty(N) :-
    N =< 20,
    Square is N * N,
    write(N),
    write(' squared is '),
    write(Square), nl,
    N1 is N + 1,
    square_to_twenty(N1).

square_to_twenty(N) :-
    N > 20.
```

```
?- square_to_twenty(17)
17 squared is 289
18 squared is 324
19 squared is 361
20 squared is 400
true
```

Artificial Intelligence Lab (BTCOL 706)

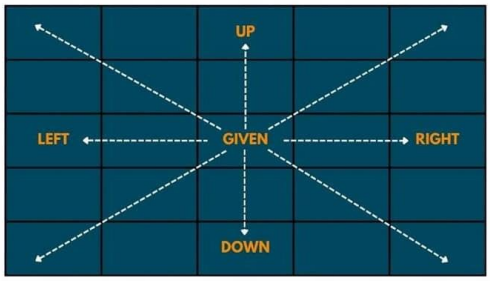
Experiment No – 02

Aim: Write a Program to solve N X N Queen Problem.

Theory:

N Queen problem demands us to place N queens on a N x N chessboard so that no queen can attack any other queen directly.

Problem Statement: We need to find out all the possible arrangements in which N queens can be seated in each row and each column so that all queens are safe. The queen moves in 8 directions and can directly attack in these 8 directions only.



The N-Queens problem is a classic combinatorial problem that requires placing N chess queens on an N×N chessboard in such a way that no two queens threaten each other. Here's a high-level algorithm for solving the N-Queens problem:

N X N Queen Problem Algorithms:

1. **Initialize the Chessboard:**

- Create an empty N×N chessboard to represent the placement of queens. This can be represented as a 2D array or a list of lists.

2. **Place Queens Recursively:**
   - Start with the first row (or any row you prefer).
   - For each cell in the row, try to place a queen and check if it's safe. If it's safe, mark the cell as occupied by a queen.
   - Move to the next row and repeat the process recursively.
   - If you reach a row where you can't place a queen without violating the rules (no two queens in the same row, column, or diagonal), backtrack to the previous row and explore other options.
3. **Base Case:**
   - When you've successfully placed N queens on the board without any conflicts, you've found a solution. Store or display this solution.
4. **Backtracking:**
   - If you encounter a situation where you cannot place a queen in any cell of the current row without conflicts, backtrack to the previous row and explore other options. This is done by undoing the placement of the last queen and trying the next available cell in the current row.
5. **Repeat:**
   - Continue this process, moving forward and backward between rows, until you've found all possible solutions or determined that no more solutions exist.
6. **Output Solutions:**
   - Keep track of all valid solutions found during the search. Once the search is complete, you can display or use these solutions as needed.

Here's a more detailed description of the steps within the recursive placement process:

- In each row, iterate through the columns to find a safe spot for a queen.
- To check if it's safe to place a queen in a cell, you need to verify that no other queens are in the same column, same row, or diagonals (both left and right diagonals).
- If a safe spot is found in the current row, place a queen, mark the cell as occupied, and proceed to the next row.

- If no safe spot is found in the current row, backtrack to the previous row, undo the placement of the queen, and explore other options.

This algorithm is often implemented using recursion and backtracking techniques. It systematically explores the solution space, trying different combinations of queen placements until all possible solutions have been found or it determines that no solution exists for the given N.

Program: Write a following program and take print with output and attached

1. Write a Program in Prolog or Python to solve N X N Queen Problem.

Questions:

1. Explain Step performed by Problem Solving Agent
2. Explain Constraint Satisfaction Problem in details with example.
3. Explain Constraint Propagation and local consistency with type.

.

(Subject In-charge)

(Prof.S.B.Mehta)

Write a Program in Prolog or Python to solve N X N Queen Problem

```
def solve_n_queens(n):
    board = [-1] * n
    solution = []
    if solve(board, 0, solution):
        return solution
    return None

def solve(board, row, solution):
    n = len(board)
    if row == n:
        # A solution is found; add it to the solution list
        solution.extend(board.copy())
        return True
    else:
        # Try placing a queen in each column of the current row
        for col in range(n):
            if is_safe(board, row, col):
                board[row] = col
                if solve(board, row + 1, solution):
                    return True
                # Reset the row position for backtracking
                board[row] = -1
    return False

def is_safe(board, row, col):
    # Check if it's safe to place a queen at (row, col)
    for i in range(row):
        if board[i] == col or \
```

```
            board[i] - i == col - row or \
            board[i] + i == col + row:
                return False
        return True


    def print_solution(solution, n):
        # Print the solution in a readable format
        if not solution:
            print("No solution found.")
        else:
            print("solution:")
            for i in range(n):
                row = ['.'] * n
                row[solution[i]] = 'Q'
                print(' '.join(row))


    # Set N value for the board size
    n = 4  # Change this value to solve for a different board size
    solution = solve_n_queens(n)
    print_solution(solution, n)
```

```
C:\Users\UTKARSH\anaconda3\python.exe C:\Users\UTKARSH\PycharmProjects\Area\main.py
solution:
. Q . .
. . . Q
Q . . .
. . Q .

Process finished with exit code 0
```