EXPERIMENT-1

Aim:

Introduction to PROLOG.

Theory:

Prolog is a declarative programming language based on formal logic, particularly first-order logic. Its name stands for "PROgramming in LOGic." In Prolog, you specify what you want to achieve rather than how to achieve it. This is done by defining a set of logical rules and relationships between objects, and then querying these rules to derive answers to specific queries.

Key features of Prolog include:

1. **Logic-based programming**: Prolog programs consist of facts and rules represented in terms of predicates and clauses. Predicates define relationships between objects, while clauses specify rules and conditions.
2. **Pattern matching**: Prolog uses unification to match patterns in queries with patterns in the knowledge base. This enables flexible querying and inference.
3. **Backtracking**: Prolog employs a depth-first search strategy with backtracking to explore different branches of computation when finding solutions to queries. This allows Prolog to handle non-deterministic computations.
4. **Built-in search mechanisms**: Prolog provides built-in mechanisms for searching through the solution space, including the use of cut (!) to control backtracking and the adoption of various search strategies.

Uses of Prolog:

1. **Artificial Intelligence**: Prolog is commonly used in artificial intelligence and expert systems due to its ability to represent and manipulate symbolic information effectively.
2. **Natural Language Processing**: Prolog's pattern matching capabilities make it suitable for natural language processing tasks such as parsing and semantic analysis.
3. **Database Systems**: Prolog can be used to query and manipulate databases through its logic-based approach, making it useful in database systems and knowledge representation.
4. **Problem Solving**: Prolog is often employed in solving combinatorial problems, constraint satisfaction problems, and other types of logic-based puzzles.

5. **Education**: Prolog is frequently taught in academic settings to illustrate concepts of logic programming and to introduce students to declarative programming paradigms.

Overall, Prolog provides a powerful framework for expressing and solving problems in a logical and declarative manner, making it a valuable tool in various domains of computer science and beyond.

EXPERIMENT-2

Aim:

Write Simple Fact for the Statements using Prolog

1. Ram Likes Mango.
2. Seema is a Girl.
3. Bill Likes Cindy.
4. Rose is Red.
5. John Owns Gold.

Theory:

Facts: Facts are statements that assert a relationship between objects. They are typically written as predicate terms.

Eg: parent(john, mary).

This fact asserts that "john" is a parent of "mary".

facts - Notepad

File Edit Format View Help

```
likes(ram,mango).
likes(bill,cindy).
girl(seema).
red(rose).
owns(john,gold).
```

GNU Prolog console

File Edit Terminal Prolog Help

```
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021, 12:22:53 with gcc
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('C:/GNU-Prolog/bin/facts.pl').
compiling C:/GNU-Prolog/bin/facts.pl for byte code...
C:/GNU-Prolog/bin/facts.pl compiled, 4 lines read - 716 bytes written, 11 ms

yes
| ?- likes(ram,mango).

yes
| ?- likes(bill,cindy).

yes
| ?- girl(seema).

yes
| ?- red(rose).

yes
| ?- owns(john,gold).
```

EXPERIMENT-3

Aim:

Write Predicates, One Converts Centigrade Temperature to Fahrenheit, the other Check If a Temperature is Below Freezing using PROLOG.

Theory:

1. **Converting Celsius to Fahrenheit**: The formula to convert Celsius to Fahrenheit is: $F=(9/5)C+32$ Where *F* is the temperature in Fahrenheit and *C* is the temperature in Celsius.
2. **Checking if a Temperature is Below Freezing**: In the Celsius scale, the freezing point of water is 0°C. So, any temperature below 0°C is considered freezing.

exp3 - Notepad

File Edit Format View Help

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

below_freezing(Temp):-c_to_f(Temp,F),F<32.
```

GNU Prolog console

File Edit Terminal Prolog Help

```
Compiled Jul  8 2021, 12:22:53 with gcc
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('C:/GNU-Prolog/bin/exp3.pl').
compiling C:/GNU-Prolog/bin/exp3.pl for byte code...
C:/GNU-Prolog/bin/exp3.pl compiled, 2 lines read - 796 bytes written, 10 ms

yes
| ?- c_to_f(90,F).

F = 194.0

yes
| ?- c_to_f(0,F).

F = 32.0

yes
| ?- below_freezing(32).

no
| ?- below_freezing(-1).

yes
| ?-
```

EXPERIMENT-4

Aim:

Write a Program to implement Breath First Search Traversal.

Theory:

1. **Representing the Graph**: The graph is represented using **edge/2** facts, where each fact indicates an edge between two nodes.
2. **Defining the Goal Condition**: The **goal/1** predicate defines the condition for the goal node. In your case, it checks if a given node is equal to the goal node.
3. **BFS Traversal**: The **bfs/3** predicate initiates the BFS traversal. It calls the **bfs_queue/3** predicate with an initial queue containing the start node.
4. **Base Case**: The base case of **bfs_queue/3** checks if the current node is the goal node. If it is, it returns the path containing only the goal node.
5. **Recursive Case**: In the recursive case of **bfs_queue/3**, it explores the neighbors of the current node. It finds all neighbors using **findall/3**, appends them to the queue, and continues the traversal recursively.

```
bfs - Notepad
File  Edit  Format  View  Help
% Define edges in the graph
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).
edge(c, g).

% Define the goal condition
goal(Node) :- Node = d. % Example goal node

% bfs(Start, Goal, Path) is true if Path is a path from Start to Goal
bfs(Start, Goal, Path) :-
    bfs_queue([Start], Goal, Path).

% Base case: Goal is the current node
bfs_queue([Node|_], Node, [Node]).

% Recursive case: Explore neighbors of the current node
bfs_queue([Node|Rest], Goal, [Node|Path]) :-
    findall(Next, edge(Node, Next), Neighbors), % Find all neighbors of the
current node
    append(Rest, Neighbors, NewQueue), % Add neighbors to the end of the queue
    bfs_queue(NewQueue, Goal, Path).
Ln 19, Col 52    100%    Windows (CRLF)    UTF-8
```

```
GNU Prolog console
File  Edit  Terminal  Prolog  Help
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021, 12:22:53 with gcc
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('C:/GNU-Prolog/bin/bfs.pl').
compiling C:/GNU-Prolog/bin/bfs.pl for byte code...
C:/GNU-Prolog/bin/bfs.pl compiled, 22 lines read - 1831 bytes written, 12 ms

(16 ms) yes
| ?- bfs(a,g,Path).

Path = [a,b,c,d,e,f,g] ?

yes
| ?- bfs(a,c,Path).

Path = [a,b,c] ? |
```

EXPERIMENT-5

Aim:

Write a Program to Implement Water Jug Problem.

Theory:

The water jug problem, also known as the water pouring problem, is a classic puzzle that involves using two or more jugs of different capacities to measure out a desired amount of water. The problem typically involves determining a sequence of actions (pouring, filling, and emptying) that will result in a particular amount of water being present in one of the jugs.

1. **Actions**: The actions available in the water jug problem typically include:
    - **Pouring**: Transferring water from one jug to another.
    - **Filling**: Filling a jug to its maximum capacity.
    - **Emptying**: Emptying the contents of a jug.
2. **Constraints**:
    - The water jugs cannot hold more water than their respective capacities.
    - Water cannot be split or combined (i.e., there is no spillage or overflow during pouring).
    - Only the actions of pouring, filling, and emptying are allowed.
3. **Search Algorithm**: Solving the water jug problem often involves using a search algorithm, such as depth-first search (DFS) or breadth-first search (BFS), to explore the space of possible states (configurations of water in the jugs) and find a sequence of actions that leads to the desired outcome.
4. **State Representation**: The state of the problem is typically represented as a tuple or list containing the current amount of water in each jug. For example, a state **(2, 0)** represents that the first jug contains 2 units of water, and the second jug is empty.
5. **Goal State**: The goal state is the state in which one of the jugs contains the desired amount of water, as specified by the problem statement.

waterjug - Notepad

File Edit Format View Help

```prolog
% Action rules for pouring water between jugs

% Pour from jug 1 to jug 2
pour(jug1, jug2, State, NewState) :-
    member(jug(Jug1Amount, Jug2Amount), State),
    Jug1Amount > 0,
    Jug2Amount < 3,
    NewJug2Amount is min(Jug1Amount + Jug2Amount, 3),
    NewJug1Amount is Jug1Amount - (NewJug2Amount - Jug2Amount),
    NewState = [jug(NewJug1Amount, NewJug2Amount) | State].

% Pour from jug 2 to jug 1
pour(jug2, jug1, State, NewState) :-
    member(jug(Jug1Amount, Jug2Amount), State),
    Jug2Amount > 0,
    Jug1Amount < 4,
    NewJug1Amount is min(Jug1Amount + Jug2Amount, 4),
    NewJug2Amount is Jug2Amount - (NewJug1Amount - Jug1Amount),
    NewState = [jug(NewJug1Amount, NewJug2Amount) | State].

% Fill jug 1
fill(jug1, State, NewState) :-
    member(jug(_, Jug2Amount), State),
    NewState = [jug(4, Jug2Amount) | State].

% Fill jug 2
fill(jug2, State, NewState) :-
    member(jug(Jug1Amount, _), State),
    NewState = [jug(Jug1Amount, 3) | State].

% Empty jug 1
empty(jug1, State, NewState) :-
    member(jug(_, Jug2Amount), State),
    NewState = [jug(0, Jug2Amount) | State].


% Empty jug 2
empty(jug2, State, NewState) :-
    member(jug(Jug1Amount, _), State),
    NewState = [jug(Jug1Amount, 0) | State].

% Check if the target amount is reached
target_reached(State) :-
    member(jug(_, 2), State).

% Depth-first search to find a solution
dfs(Start, _, Visited, Actions) :-
    target_reached(Start),
    reverse(Visited, Actions).

dfs(State, DepthLimit, Visited, Actions) :-
    DepthLimit > 0,
    DepthLimit1 is DepthLimit - 1,
    (pour(_, _, State, NextState);
     fill(_, State, NextState);
    empty(_, State, NextState)),
    \+ member(NextState, Visited),
    dfs(NextState, DepthLimit1, [NextState | Visited], Actions).

% Predicate to find a solution
find_solution(Start, MaxDepth, Actions) :-
    dfs(Start, MaxDepth, [Start], Actions),nl.
```

GNU Prolog console

File Edit Terminal Prolog Help

```
NU Prolog 1.5.0 (64 bits)
ompiled Jul  8 2021, 12:22:53 with gcc
opyright (C) 1999-2021 Daniel Diaz

 ?- consult('C:/GNU-Prolog/bin/waterjug.pl').
ompiling C:/GNU-Prolog/bin/waterjug.pl for byte code...
:/GNU-Prolog/bin/waterjug.pl compiled, 60 lines read - 6615 bytes written, 10 ms

es
 ?- find_solution([jug(0,0)], 10, Actions).


ctions = [[jug(0,0)],[jug(4,0),jug(0,0)],[jug(1,3),jug(4,0),jug(0,0)],[jug(1,3),jug(1,3),jug(4,0),jug(
,0)],[jug(1,3),jug(1,3),jug(1,3),jug(4,0),jug(0,0)],[jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(4,0),jug(
,0)],[jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(4,0),jug(0,0)],[jug(0,3),jug(1,3),jug(1,3),jug(
,3),jug(1,3),jug(1,3),jug(4,0),jug(0,0)],[jug(3,0),jug(0,3),jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(1,
),jug(4,0),jug(0,0)],[jug(3,3),jug(3,0),jug(0,3),jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(4,0)
jug(0,0)],[jug(4,2),jug(3,3),jug(3,0),jug(0,3),jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(1,3),jug(4,0),j
g(0,0)]] ?
```