**Experiment - 1**

**Aim:** Study of PROLOG.

**Theory:**

Prolog is a logic programming language that is used to create artificial intelligence (AI). It is a free and open-source programming language that is available on many platforms.

Prolog is short for programming logic. It 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.

In Prolog, logic is expressed as relations (called as Facts and Rules). The programmer specifies a set of rules and facts about a problem domain, and then use those rules and facts to automatically infer solutions to problems.

**• Facts −** The fact is predicate that is true, for example, if we say, “Shyam is the son of Ram”, then this is a fact.

**• 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.

**• Questions −** And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Prolog has many built-in features for AI programming, such as backtracking, data structures, and pattern matching.

Prolog was created around 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses.

**Syntax:**

In Prolog, we create a list of facts which make up the system's knowledge base. We can ask questions to this knowledge base. If our question matches something in the knowledge base or can be deduced from it, we'll get a positive response. If not, we'll get a negative response. The knowledge base is like a database that we can ask questions to. In Prolog, facts are written in a specific format: there's a relationship expressed outside the parentheses, with entities listed inside separated by commas.   
Each fact or rule ends with a period(.).  
  
**Format :** relation(entity1, entity2, ....k'th entity).

**Example :**

Father\_of(Raju, Monu).

actor(Sonu).

even\_number(14).

**Explanation :**

Raju is the father of Monu.

Sonu is an actor.

14 is an even number.

**Key Features:**

1. Unification: The basic idea is, can the given terms be made to represent the same structure.

2. Backtracking: When a task fails, prolog traces backwards and tries to satisfy previous task.

3. Recursion: Recursion is the basis for any search in program.

**Some Applications of Prolog:**

Prolog is used in various domains. It plays a vital role in automation system. Following are some other important fields where Prolog is used −

• Intelligent Database Retrieval

• Natural Language Understanding

• Specification Language

• Machine Learning

• Robot Planning

• Automation System

• Problem Solving

**PROLOG Data Types:**

1. Char: Between a pair of single quotes, a character is encapsulated.

2. Integer: An integer between -32768 and 32767 that is a full number.

3. Real: A peculiar character that is either positive or negative, followed by numbers.

4. String: A collection of characters encased in a pair of double-quotes. Strings can have up to 255 characters in them.

5. Symbol: A combination of letters (A to Z or a to z), numerals (0 to 9) and the underscore(\_) character.

6. Variables: A variable is a symbol that can have multiple values assigned to it at different stages of the program’s execution.

7. Reserved terms: PROLOG features a few reserved words that should not be substituted for user-defined names.

8. Arithmetic Operators: The basic arithmetic operators in PROLOG are +, -, \*, and /.

9. Relational Operators: PROLOG utilises the relational operators,=, =>, >=, >=. A relational operator in PROLOG can be either goal or subgoal. The relational operator (=) resembles an assignment operator in appearance.

**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 | ^ | . |

**Experiment - 2**

**Aim:** Write simple fact for the statements using PROLOG

a. Ram likes mango.

b. Seema is girl.

c. Bill likes Cindy.

d. Rose is red.

e. John owns gold.

**Program:**

likes(ram, mango).

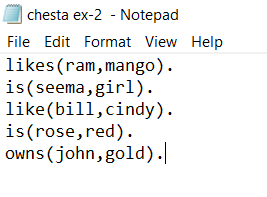
Is(seema,girl).

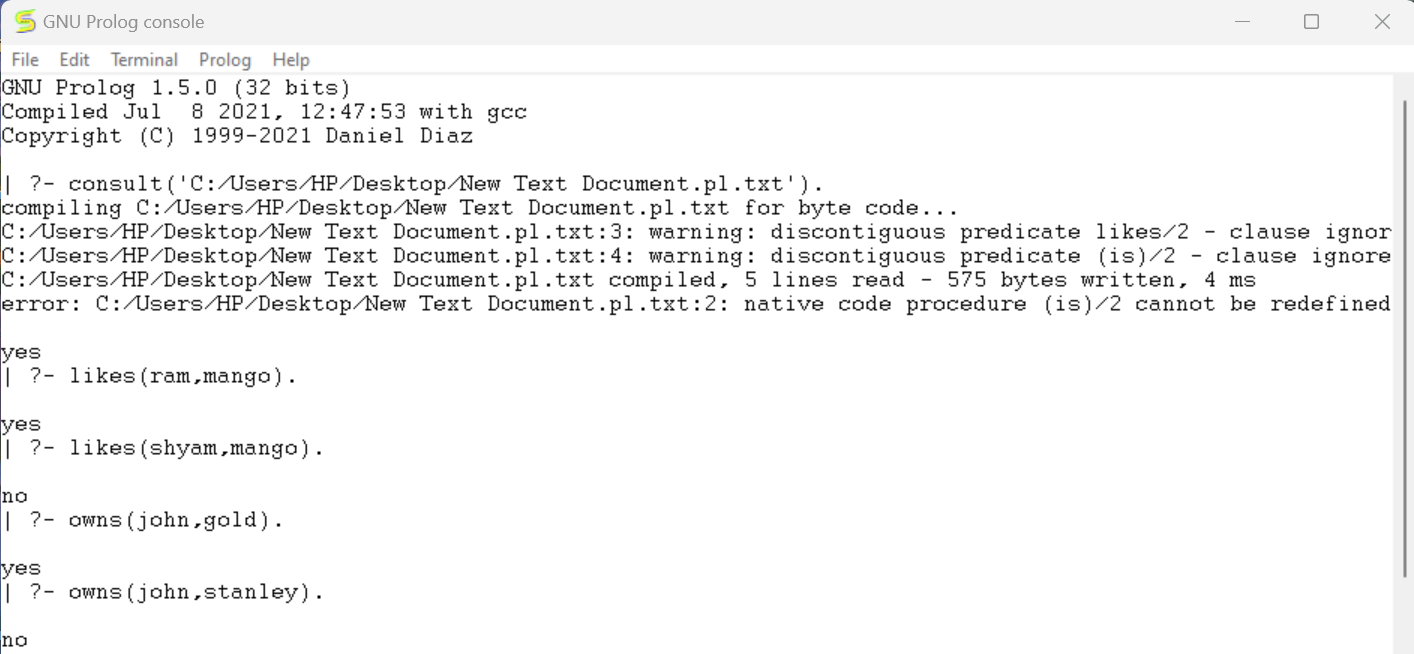
likes(bill, cindy).

Is(red,rose).

owns(john, gold).

**Output:**



****

**Experiment – 3 (a)**

**Aim:**

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

**Program:**

celsius\_to\_fah(celsius,fahrenheit):-

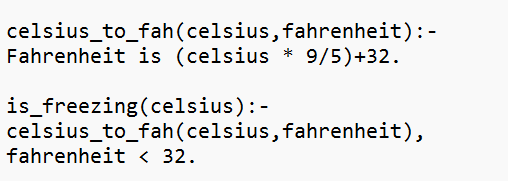
Fahrenheit is (celsius \* 9/5)+32.

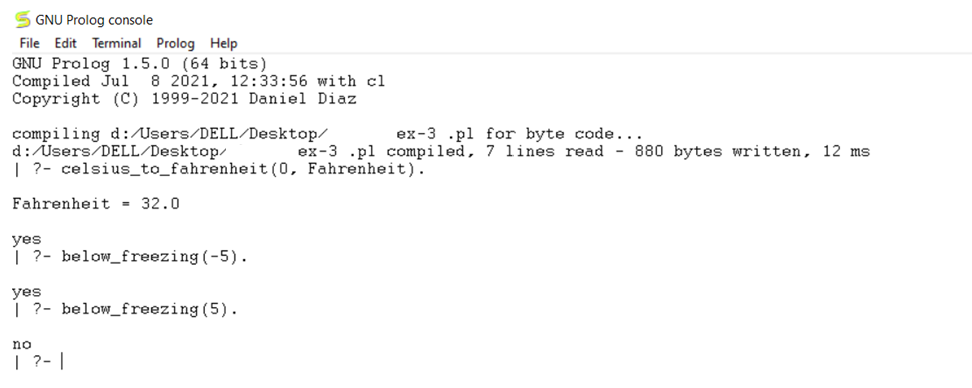
is\_freezing(celsius):-

celsius\_to\_fah(celsius,fahrenheit),

fahrenheit < 32.

**Output:**



****

**Experiment - 3(b)**

**Aim:** Write program to calculate the sum, subtraction and multiplication of 2 numbers using variables.

1: - **For sum,** 2: - **For subtraction,** 3: - **For multiplication.**

**Program:**sum(X, Y, S):-

S is X + Y,

write(S).

sub(X, Y, S):-

S is X - Y,

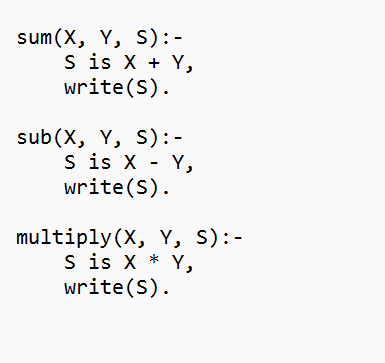
write(S).

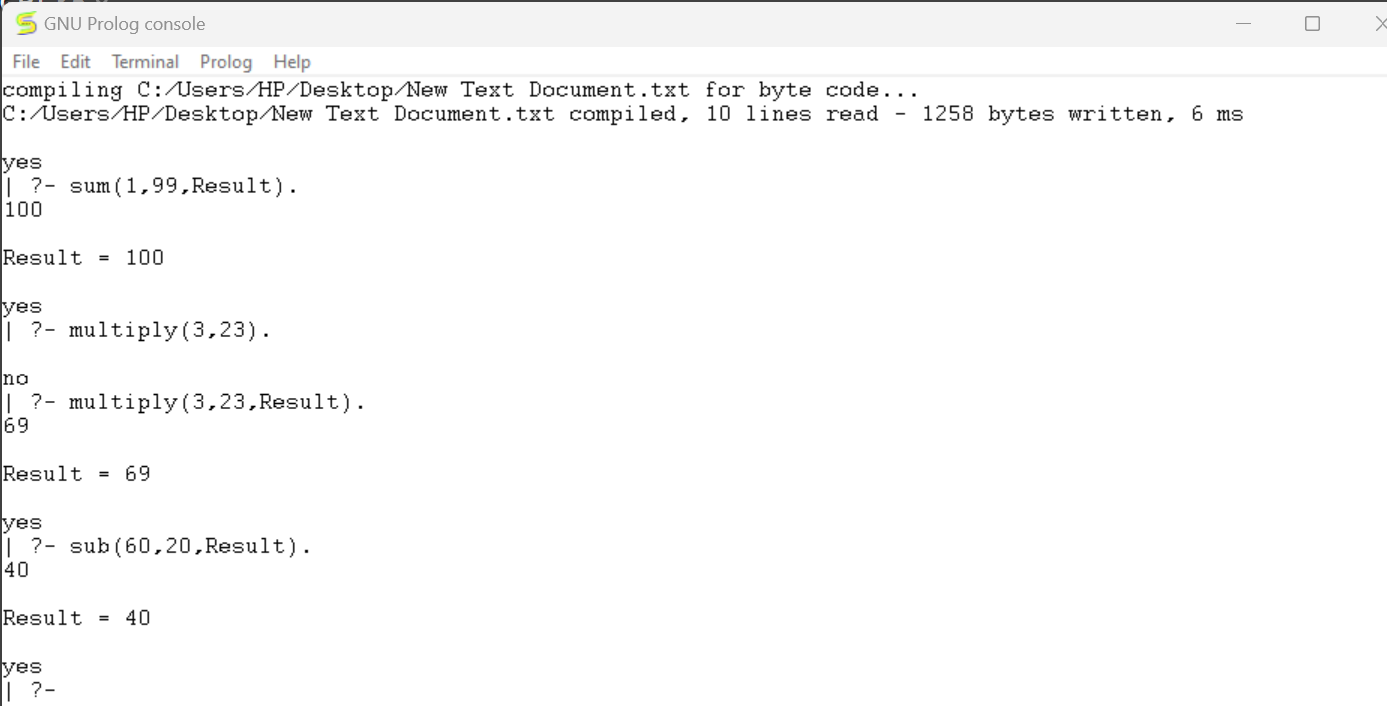
multiply(X, Y, S):-

S is X \* Y,

write(S).

**Output:**

****

****

**Experiment-4(a)**

**Aim:**

Write a program to implement Breadth First Search Traversal.

**Theory:**

Breadth-first search (BFS) is a graph traversal algorithm that explores the neighbor nodes at the current depth level before moving on to nodes at the next depth level. It starts from a designated node and explores all its neighboring nodes using queues.

**Algorithm:**

1. Initialize a queue and enqueue the starting node with an empty path.

2. While the queue is not empty:

a. Dequeue a node along with its path.

b. If the dequeued node is the goal node, print the path and continue.

c. Visit the dequeued node and enqueue all unvisited adjacent nodes with the updated path.

3. Repeat step 2 until the queue is empty or all nodes are visited.

**Program:**

edge (a, b).

edge (a, c).

edge (b, d).

edge (b, e).

edge (c, f).

edge (c, g).

goal (Node): - Node = d.

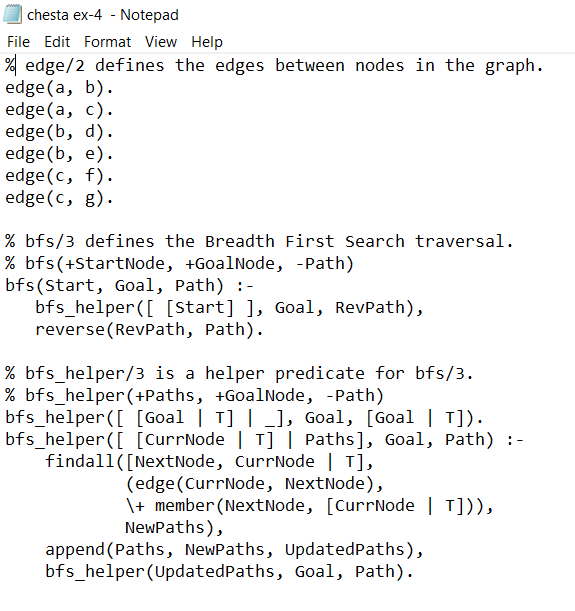
bfs (Start, Goal, Path): - bfs\_queue([Start], Goal, Path).

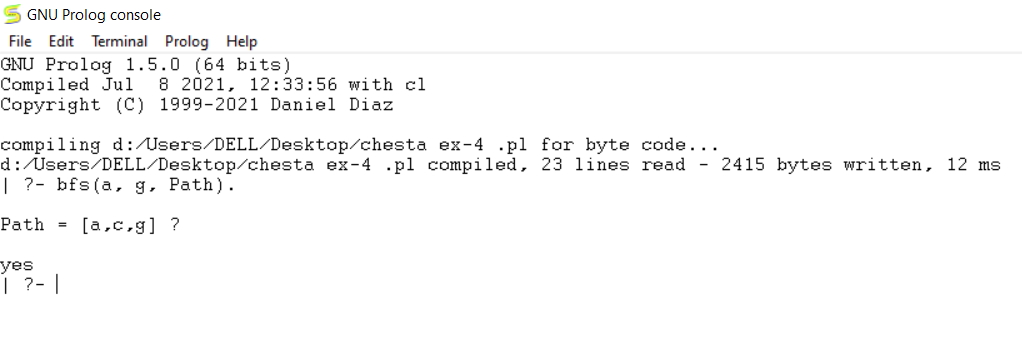
bfs\_queue([Node|\_], Node, [Node]).

bfs\_queue([Node|Rest], Goal, [Node|Path]): - findall (Next, edge (Node, Next), Neighbours),

append (Rest, Neighbours, NewQueue), bfs\_queue (NewQueue, Goal, Path).

**Output:**





**Experiment-4(b)**

**Aim:**

Write a program to implement Depth First Search Traversal.

**Theory:**

Depth-first search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It starts from a node and explores as deeply as possible along each branch before backtracking to explore other branches, using a stack.

**Algorithm**:

1. Initialize a stack and push the starting node with an empty path.

2. While the stack is not empty:

a. Pop a node along with its path from the stack.

b. If the popped node is the goal node, print the path and continue.

c. Visit the popped node and push all unvisited adjacent nodes with the updated path onto the stack.

3. Repeat step 2 until the stack is empty or all nodes are visited.

**Program:**

edge (x, y).

edge (x, z).

edge (y, w).

edge (y, v).

edge (z, u).

edge (z, t).

edge (w, s).

edge (v, r).

dfs (Node, Goal, Path): -

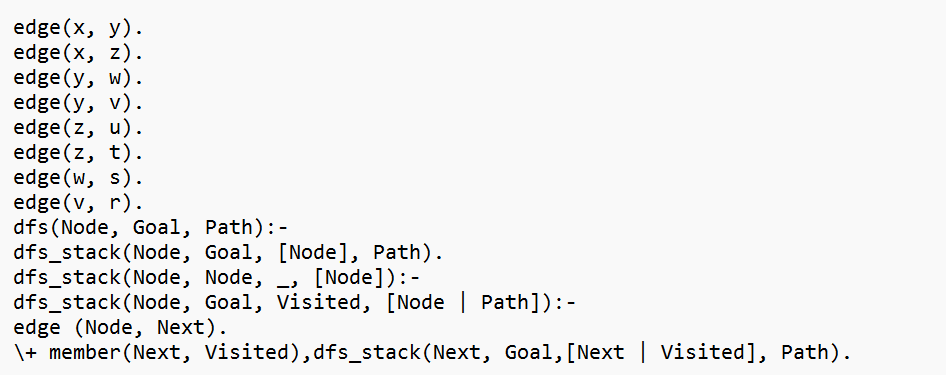
dfs\_stack (Node, Goal, [Node], Path).

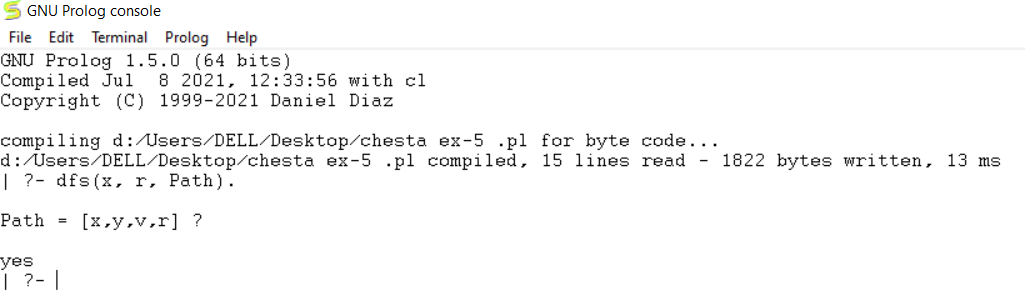
dfs\_stack (Node, Node, \_, [Node]): -!

dfs\_stack (Node, Goal, Visited, [Node | Path]): - edge (Node, Next),

\+ member (Next, Visited), dfs\_stack (Next, Goal, [Next | Visited], Path)

**Output:**

****



**Experiment - 5**

**Aim:**

Write a program to implement Water jug problem.

**Program:**

% 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].

% 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),

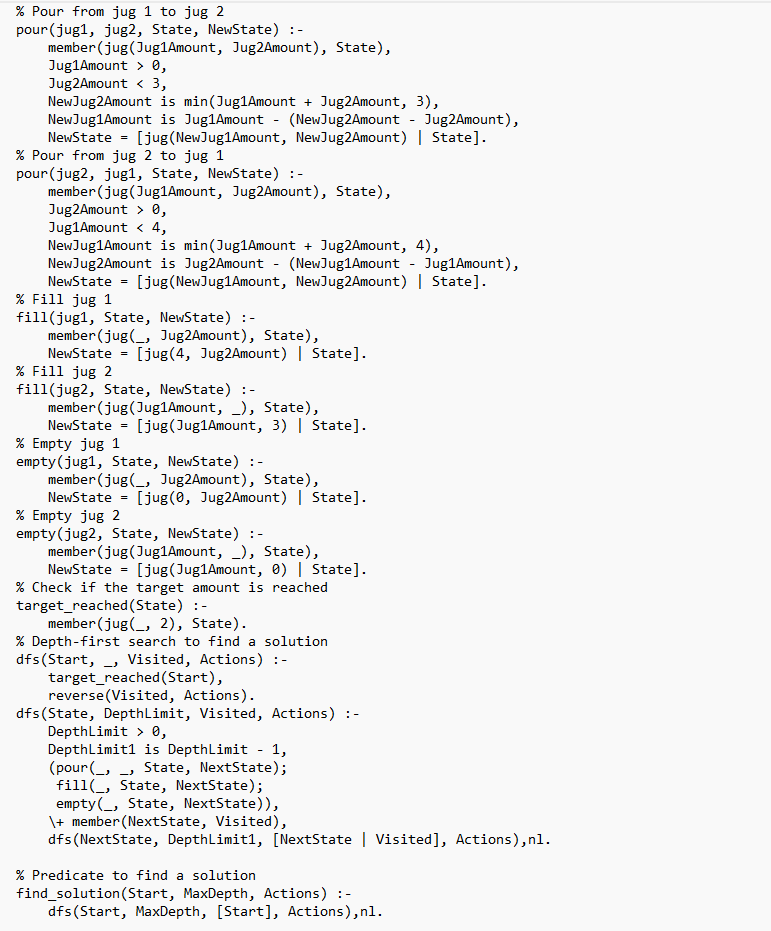
dfs(NextState, DepthLimit1, [NextState | Visited], Actions),nl.

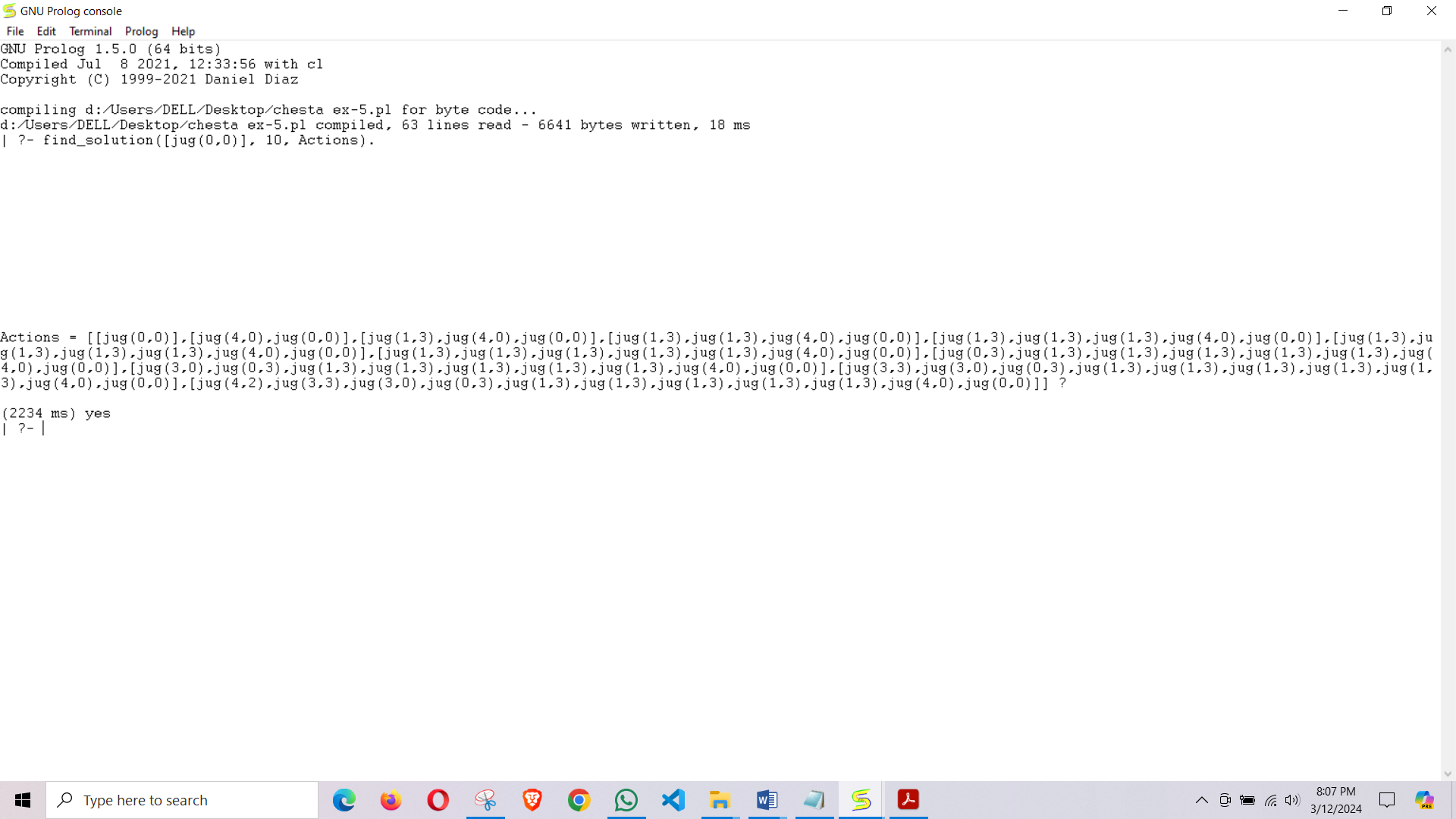
% Predicate to find a solution

find\_solution(Start, MaxDepth, Actions) :-

dfs(Start, MaxDepth, [Start], Actions),nl.

**Output:**

****

****

**EXPERIMENT – 6**

**Aim:**

Write a program to remove punctuations from the given string.

**Source Code:**

% Define a predicate to remove punctuations from a string

remove\_punctuations(Input, Output) :-

atom\_chars(Input, InputChars),

filter\_punctuations(InputChars, CleanChars),

atom\_chars(Output, CleanChars).

% Define a predicate to filter out punctuation characters

filter\_punctuations([], []).

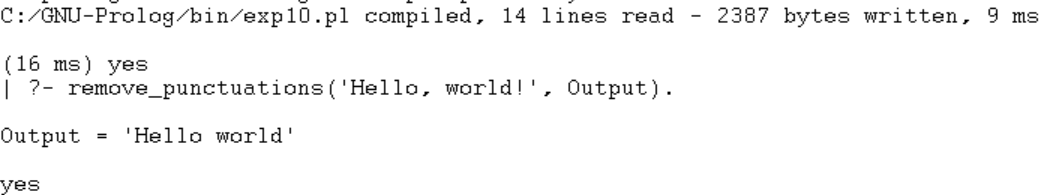
filter\_punctuations([Char|Rest], CleanChars) :-

(punctuation\_char(Char) -> filter\_punctuations(Rest, CleanChars) ; CleanChars = [Char|CleanRest], filter\_punctuations(Rest, CleanRest)).

% Define a predicate to check if a character is a punctuation character

punctuation\_char(Char) :-

member(Char, ['.', ',', ';', ':', '!', '?', '"', '\'', '-', '(', ')']).

**Output:**

**EXPERIMENT – 7**

**Aim:**

Write a program to sort the sentence in alphabetical order.

**Source Code:**

 % Base case: If the list is empty, it is already sorted

sort\_sentence([], []).

% Sort the sentence by finding the minimum word in the list

sort\_sentence(Sentence, [Min|Sorted]) :-

find\_min(Sentence, Min),

remove\_from\_list(Sentence, Min, Remaining),

sort\_sentence(Remaining, Sorted).

% Find the minimum word in the list

find\_min([Word], Word). % Base case: The minimum of one element is itself

find\_min([Word1,Word2|Rest], Min) :-

Word1 @=< Word2, % If Word1 is less than or equal to Word2

find\_min([Word1|Rest], Min).

find\_min([Word1,Word2|Rest], Min) :-

Word1 @> Word2, % If Word1 is greater than Word2

find\_min([Word2|Rest], Min).

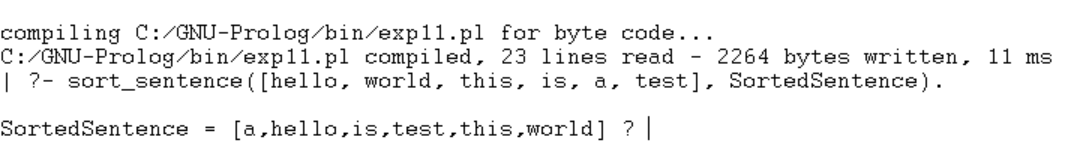
% Remove an element from the list

remove\_from\_list([], \_, []).

remove\_from\_list([X|Xs], X, Xs).

remove\_from\_list([Y|Ys], X, [Y|Zs]) :-

remove\_from\_list(Ys, X, Zs).

**Output:**