

Artificial Intelligence LAB
(CIE - 374P)

Faculty Name : Mr. Saurabh Rastogi

Name : Amit Singhal

Enrollment No. : 11614802722

Semester : 6

Group : AIML-II-B



Maharaja Agrasen Institute of Technology, PSP Area,
Sector – 22, Rohini, New Delhi – 110085



MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

VISION

“To attain global excellence through **education, innovation, research, and work ethics** with the commitment to **serve humanity.**”

MISSION

- M1.** To promote diversification by adopting advancement in science, technology, management, and allied discipline through continuous learning
- M2.** To foster **moral values** in students and equip them for developing sustainable solutions to serve both national and global needs in society and industry.
- M3.** To **digitize educational resources and process** for enhanced teaching and effective learning.
- M4.** To cultivate an **environment** supporting **incubation, product development, technology transfer, capacity building and entrepreneurship.**
- M5.** To encourage **faculty-student networking with alumni, industry, institutions,** and other **stakeholders** for collective engagement.



Department of Computer Science and Engineering

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

VISION

"To attain global excellence through education, innovation, research, and work ethics in the field of Computer Science and engineering with the commitment to serve humanity."

MISSION

- M1.** To lead in the advancement of computer science and engineering through internationally recognized research and education.
- M2.** To prepare students for full and ethical participation in a diverse society and encourage lifelong learning.
- M3.** To foster development of problem solving and communication skills as an integral component of the profession.
- M4.** To impart knowledge, skills and cultivate an environment supporting incubation, product development, technology transfer, capacity building and entrepreneurship in the field of computer science and engineering.
- M5.** To encourage faculty, student's networking with alumni, industry, institutions, and other stakeholders for collective engagement.

Rubrics for Lab Assessment:

Rubrics		10 Marks			POs and PSOs Covered	
		0 Marks	1 Marks	2 Marks	PO	PSO
R1	Is able to identify and define the objective of the given problem?	No	Partially	Completely	PO1, PO2	PSO1, PSO2
R2	Is proposed design/procedure/algorithm solves the problem?	No	Partially	Completely	PO1,PO2, PO3	PSO1, PSO2
R3	Has the understanding of the tool/programming language to implement the proposed solution?	No	Partially	Completely	PO1,PO3, PO5	PSO1, PSO2
R4	Are the result(s) verified using sufficient test data to support the conclusions?	No	Partially	Completely	PO2,PO4, PO5	PSO2
R5	Individuality of submission?	No	Partially	Completely	PO8, PO12	PSO1, PSO3

LAB INDEX

[illegible]

EXPERIMENT - 1

AIM :: Study of Prolog.

Theory ::

Prolog, short for "Programming in Logic," is a powerful declarative programming language that's particularly well-suited for tasks involving symbolic computation and artificial intelligence. Unlike imperative languages like C or Python, where you specify the exact steps to solve a problem, Prolog focuses on describing the relationships and constraints within a problem domain.

Key Concepts

Facts and Rules -

- **Facts:** These are basic statements about the problem domain. They are typically represented as simple relationships between objects.

```
parent(alice, bob).
```

```
parent(alice, carol).
```

```
parent(bob, david).
```

- **Rules:** These define more complex relationships between objects. They consist of a head (the conclusion) and a body (the conditions).

```
ancestor(X, Y) :- parent(X, Y).
```

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Unification

Prolog's core mechanism for matching patterns. It compares terms (constants, variables, or structures) and attempts to find a substitution that makes them identical.

Backtracking

Prolog's built-in search strategy. When a rule fails to match, Prolog backtracks to the previous choice point and tries an alternative.

Declarative Programming

In Prolog, you describe what you want to achieve rather than how to achieve it. The Prolog engine figures out the steps needed to find a solution.

Applications of Prolog -

Artificial Intelligence

- Expert systems
- Natural language processing
- Machine learning

Symbolic Computation

- Theorem proving
- Constraint satisfaction problems

Other

- Logic programming
- Database systems
- Software engineering

Advantages of Prolog -

- Expressive: Well-suited for representing complex relationships and knowledge
- Declarative: Focuses on what needs to be done, not how
- Backtracking: Provides a powerful search mechanism

Disadvantages of Prolog -

- Efficiency: Can be less efficient than imperative languages for certain tasks
- Debugging: Can be challenging due to its non-deterministic nature

Facts in Prolog

In Prolog, facts are fundamental building blocks that represent basic truths or relationships within the problem domain. They are declarative statements that provide the knowledge base for the Prolog system to reason with.

Structure of Facts -

Components

- **Predicate:** A relation or property that holds true for certain objects or entities. It's essentially the name of the fact.
- **Arguments:** Entities involved in the relationship. They can be:
 - Constants (specific values)
 - Variables (placeholders for values)
 - Complex terms

Syntax

`predicate(argument1, argument2, ..., argumentN).`

Example

Let's consider a simple family relationship:

`parent(alice, bob).`

In this example:

- `parent` is the predicate, representing the parent-child relationship
- `alice` and `bob` are the arguments, representing the parent and child, respectively
- This fact declares that "Alice is the parent of Bob"

Key Points

- Facts are always true
- They are the foundation upon which rules are built
- They are used to represent specific instances or relationships within the problem domain

Rules in Prolog

Rules define implicit relationships between objects. Facts become conditionally true - when one associated condition is true, then the predicate is also true.

Example Rules

- Lili is happy if she dances
- Tom is hungry if he is searching for food
- Jack and Bili are friends if both of them love to play cricket
- Will go to play if school is closed, and he is free

Syntax Elements

- `:-` (neck symbol): Pronounced as "If" or "is implied by"
 - Left-hand side (LHS): Called the Head
 - Right-hand side (RHS): Called the Body
- `,` (comma): Known as conjunction
- `;` (semicolon): Known as disjunction


```
rule_name(object1, object2, ...) :- fact/rule(object1,  
object2, ...)
```

Suppose a clause is like :

```
P :- Q;R.
```

This can also be written as

```
P :- Q.
```

```
P :- R.
```

If one clause is like :

```
P :- Q,R;S,T,U.
```

Is understood as

```
P :- (Q,R);(S,T,U).
```

Or can also be written as:

```
P :- Q,R.
```

```
P :- S,T,U.
```

Examples in Prolog Syntax

```
happy(lili) :- dances(lili).
```

```
hungry(tom) :- search_for_food(tom).
```

```
friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).
```

```
goToPlay(ryan) :- isClosed(school), free(ryan).
```

Queries in Prolog

Queries are questions about relationships between objects and object properties. They allow us to extract information from Prolog's knowledge base.

Example Queries

- Is tom a cat?
- Does Kunal love to eat pasta?
- Is Lili happy?
- Will Ryan go to play?

How Queries Work?

- Queries are checked against the Knowledge Base
- Returns affirmative if:
 - Query matches facts directly in Knowledge Base
 - Query can be implied by Knowledge Base rules
- Returns negative otherwise

Query 1 : ?- singer(sonu).

Output : Yes.

Explanation : As our knowledge base contains the above fact, so output was 'Yes', otherwise it would have been 'No'.

Query 2 : ?- odd_number(7).

Output : No.

Explanation : As our knowledge base does not contain the above fact, so output was 'No'.

Predicates in Prolog

A predicate is the fundamental building block for representing relationships and knowledge in Prolog.

Key Characteristics -

Represents Relationships

Examples:

`parent(X, Y)` *% X is the parent of Y*

`likes(Person, Food)` *% Person likes Food*

`color(Object, Color)` *% Object has the color Color*

Used in Facts and Rules

- **Facts:** Represent basic, unchanging truths

`parent(alice, bob).` *% Alice is the parent of Bob*

- **Rules:** Define complex relationships

`ancestor(X, Y) :- parent(X, Y).` *% X is an ancestor of Y if X is the parent of Y*

Arguments

- Can have zero or more arguments
- Represent entities in the relationship

`parent(alice, bob)` *% alice and bob are arguments*

`likes(john, pizza)` *% john and pizza are arguments*

Arity

- Number of arguments a predicate takes
- Examples:
 - `parent(X, Y)` has arity 2
 - `likes(Person, Food)` has arity 2

Clauses in Prolog

A clause is the basic building block of a Prolog program, representing a piece of knowledge or rule.

Types of Clauses -

Facts

- Simple statements asserting truth
- Only has head (no body)
- Example:

`parent(alice, bob).`

Rules

- Defines complex relationships
- Has head and body
- Separated by :- ("if")
- Example:

`ancestor(X, Y) :- parent(X, Y).`

Key Points

1. Building Blocks: Basic units of Prolog programs
2. Knowledge Representation: Used to represent domain knowledge
3. Inference Engine: Used for logical inference and query answering

Features of Prolog

1. Logic Programming Paradigm

- Declarative approach vs imperative
- Focuses on what to achieve, not how
- Structured knowledge representation

2. Core Mechanisms

- Unification: Pattern-matching
- Backtracking: Built-in search strategy

3. Key Features

- Symbolic Computation
- AI Applications

- Database Integration
- Programming Style Flexibility

4. Advantages

- Expressive power
- Declarative nature
- Robust search mechanisms

5. Disadvantages

- Efficiency concerns
- Debugging challenges

Symbolic Language Aspects

1. Knowledge Representation

Facts

`parent(alice, bob).` *% alice is parent of bob*

Rules

`ancestor(X, Y) :- parent(X, Y).`

`ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).`

Variables

- Start with uppercase or underscore
- Represent unknown values

2. Symbolic Computation

- Unification for pattern matching
- Inference engine for conclusions

3. Advantages

- Expressive representation
- Flexible adaptation
- Human-readable code

Symbols in Prolog

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	\rightarrow	<code>:-</code>
Not	\sim	<code>Not</code>
Or	\vee	<code>;</code>
and	\wedge	<code>,</code>

Data Types in Prolog

The fundamental data type in Prolog is the term. Terms come in several varieties:

Basic Terms -

Atoms

- Start with lowercase letter or quoted text
- Examples:
 - `x`
 - `red`
 - `'Taco'`
 - `'some atom'`
 - `'p(a)'`

Numbers

- Includes integers and floats
- Supports arbitrary-length integers in most implementations

Variables

- Begin with uppercase letter or underscore
- Can contain letters, numbers, underscores
- Act as placeholders for arbitrary terms

Compound Terms -

- Composed of:
 - Functor (an atom)
 - Arguments (other terms)
- Written as: `functor(arg1, arg2, ...)`
- Example: `person_friends(zelda, [tom, jim])`

Special Cases

Lists

- Ordered collections of terms
- Syntax: [term1, term2, ...]
- Examples:
 - [1, 2, 3, 4]
 - [red, green, blue]
 - [] (empty list)

Strings

- Character sequences in quotes
- Can represent:
 - Character code lists
 - Character atom lists
 - Single atom
- Example: "to be, or not to be"

Metaprogramming in Prolog

Key Concepts -

Introspection Capabilities

1. Examine program structure
2. Access predicate definitions
3. Retrieve clause information
4. Analyze term structure

Meta-predicates

- `call/1`: Execute goals
- `assert/1`: Add clauses
- `retract/1`: Remove clauses
- `clause/2`: Get clause definitions

Term Manipulation

- `=.. (univ)`: Term-list conversion
- `functor/3`: Extract functor and arity
- `arg/3`: Access term arguments

Example Meta-interpreter

% Base case: If the goal is a fact, succeed.

`solve(Goal) :-`

Goal.

% Recursive case: If the goal is a rule, solve its subgoals.

`solve(Goal) :-`

`clause(Goal, Body),`

`solve(Body).`

Applications

1. Program transformation
2. Generic programming
3. Debugging tools
4. Custom interpreters

Advantages

- Flexible code reuse
- High-level program manipulation
- Expressive transformations

Expert System Design in Prolog

Core Components -

Knowledge Base

- Facts (e.g., `color(apple, red).`)
- Rules (e.g., `ripe(Fruit) :- color(Fruit, red).`)

Inference Engine

- Built-in unification
- Backtracking mechanism

User Interface

- Command-line or GUI
- Input/output handling

Design Process

1. Knowledge Acquisition

- Gather expert domain knowledge
- Identify key relationships

2. Knowledge Representation

- Convert to Prolog facts/rules
- Design data structures

3. Inference Implementation

- Control strategy design
- Optimization

4. Interface Design

- User interaction
- Result presentation

5. Testing

- Scenario validation
- Accuracy verification

Example: Disease Diagnosis System

% Facts

has_symptom(patient1, cough).

has_symptom(patient1, runny_nose).

has_symptom(patient1, sore_throat).

% Rules

diagnose(patient, common_cold) :-

has_symptom(patient, cough),

has_symptom(patient, runny_nose),

has_symptom(patient, sore_throat).

% Query

?- diagnose(patient1, Disease).

Advantages

- Natural knowledge representation
- Built-in inference capabilities
- Flexible knowledge base
- Rapid prototyping

Limitations

- Knowledge acquisition challenges
- Complex explanation generation
- Uncertainty handling requirements

Installing Prolog on Windows

To install Prolog on Windows, follow these steps:

Step 1: Download SWI-Prolog

SWI-Prolog is one of the most popular implementations of Prolog.

1. Visit the official SWI-Prolog website: [SWI-Prolog Download](http://www.swi-prolog.org/download.html)
2. Click on the Windows version and download the .exe installer.

Step 2: Install SWI-Prolog

- Run the downloaded .exe file.
- Follow the installation wizard:
 - Click **Next** to proceed.
 - Choose the installation directory (default is fine).
 - Select **Add SWI-Prolog to the system PATH** (important for running Prolog from the command line).
 - Click **Install** and wait for the installation to complete.
- Click **Finish** once the installation is done.

Step 3: Verify Installation

1. Open **Command Prompt (cmd)** or **SWI-Prolog** from the Start menu.
2. Type:

prolog.

If Prolog starts successfully, the installation is complete.

Basic "Hello, World!" Program in Prolog

Code ::

```
| ?- write('Hello World!'), nl.
```

Output ::

```
| ?- write('Hello World!'), nl.  
Hello World!
```

EXPERIMENT - 2

AIM :: Write simple fact for the statements using PROLOG.

Ram likes mango.

Seema is a girl.






Bill likes Cindy.

Rose is red.

John owns gold.

Program ::

1. Create the desired file in the bin folder for GNU-Prolog.

 create_bat	08-07-2021 12:24	Application	291 KB
 exp1s	07-02-2024 10:17	Prolog File	1 KB
 fd2c	08-07-2021 12:23	Application	1,281 KB
 gplc	08-07-2021 12:23	Application	397 KB
 gprolog	08-07-2021 12:23	Application	1,705 KB

2. Write the facts according to the question.

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

3. Open the prolog console and use the 'consult' command to compile the file and use it as a knowledge base.

```
| ?- consult('exp1s.pl').  
compiling C:/GNU-Prolog/bin/exp1s.pl for byte code...  
C:/GNU-Prolog/bin/exp1s.pl compiled, 4 lines read - 716 bytes written, 4 ms
```

4. Write the appropriate queries in the console for verification. Prolog uses negation as failure i.e. not p is true if p cannot be derived

```
% c:/Users/maitl/Documents/Prolog/exp2.pl compiled 0.00 sec, 6 clauses
?- likes(ram, mango).
true.

?- likes(john, mango).
false.

?- girl(sambhav).
false.

?- girl(seema).
true.

?- likes(john, cid).
false.

?- likes(bill, cindy).
true.

?- red(cupcake).
false.

?- red(rose).
true.

?- owns(john, silver).
false.

?- owns(john, gold).
true.

?- ■
```

EXPERIMENT - 3

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

Program ::

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

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

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

```
below_freezing_c(C) :-  
    C <= 0.
```

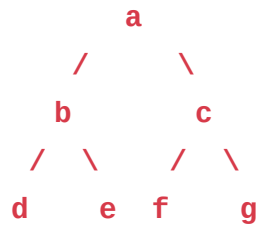
Output ::

```
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.9)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit https://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
% c:/Users/maatl/Documents/Prolog/exp3.pl compiled 0.00 sec, 2 clauses  
  
?- c_to_f(0, F).  
F = 32.  
  
?- f_to_c(32, C).  
C = 0.  
  
?- below_freezing_f(30).  
true.  
  
?- below_freezing_f(62).  
false.  
  
?- below_freezing_c(-5).  
true.  
  
?- below_freezing_c(10).  
false.
```

EXPERIMENT - 4

AIM :: Write a program to implement Breadth First Search Traversal (BFS).

Tree ::



Program ::

```
% Define the graph as facts (edges)
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).
edge(c, g).

% Check if a node exists in the graph (as a parent or child)
node_exists(Node) :-
    edge(Node, _); % Check if the node has an outgoing edge (parent)
    edge(_, Node). % Check if the node has an incoming edge (child)

% Breadth-First Search Traversal
bfs(Start, Goal) :-
    bfs_helper([Start], Goal, []).

bfs_helper([Goal | _], Goal, Path) :-
    reverse([Goal | Path], FinalPath),
    print_path(FinalPath).

bfs_helper([Node | Rest], Goal, Visited) :-
    findall(Neighbor, (edge(Node, Neighbor), \+ member(Neighbor, Visited)),
    Neighbors),
    append(Rest, Neighbors, NewQueue),
    bfs_helper(NewQueue, Goal, [Node | Visited]).

% Print the path from Start to Goal
print_path([]) :-
```

```

        writeln('No path found.').

print_path([Node]) :-
    write(Node), nl.

print_path([Node | Rest]) :-
    write(Node), write(' - '),
    print_path(Rest).

% Print the path from Start to Goal in a parent-child relationship
print_parent([]).

print_parent([Node]) :-
    write(Node), nl.

print_parent([Node | Rest]) :-
    write(Node), write(' - '),
    print_parent(Rest).

% Print the path from Start to Goal in a reverse relationship
print_reverse_path([]).

print_reverse_path([Node]) :-
    write(Node), nl.

print_reverse_path([Node | Rest]) :-
    print_reverse_path(Rest),
    write(Node), write(' - ').

```

Output ::

```

?- node_exists(a)
.
true ? y
|
| ?- bfs(a,d).
a - b - c - d
|
| ?- bfs(a,g).
a - b - c - d - e - f - g
|
| ?- bfs(a,f).
a - b - c - d - e - f
|
true ?
bfs(a,h).
no

```

EXPERIMENT - 5

AIM :: Write a program to implement Water Jug Problem.

Program ::

```
:- dynamic visited_state/2. % Declare visited_state as dynamic

% Define a predicate that takes user input and starts the process
fill(X, Y) :-
    retractall(visited_state(_, _)), % Clear visited states before starting
    assertz(visited_state(X, Y)),
    state(X, Y, [(X, Y)]). % Pass an empty list to collect steps

% Goal State: 4-Litre jug should have exactly 2 Litres
state(2, Y, Steps) :-
    format("Goal reached: (2, ~d)\n", [Y]),
    print_steps(Steps), % Print the steps
    !. % Stop further execution

% Fill the 4-Litre Jug
state(X, Y, Steps) :-
    X < 4,
    \+ visited_state(4, Y),
    assertz(visited_state(4, Y)),
    format("Fill 4-Litre Jug: (~d, ~d) --> (4, ~d)\n", [X, Y, Y]),
    state(4, Y, [(X, Y) | Steps]). % Record the current state in the steps

% Fill the 3-Litre Jug
state(X, Y, Steps) :-
    Y < 3,
    \+ visited_state(X, 3),
    assertz(visited_state(X, 3)),
    format("Fill 3-Litre Jug: (~d, ~d) --> (~d, 3)\n", [X, Y, X]),
    state(X, 3, [(X, Y) | Steps]). % Record the current state in the steps

% Empty the 4-Litre Jug
state(X, Y, Steps) :-
    X > 0,
    \+ visited_state(0, Y),
    assertz(visited_state(0, Y)),
    format("Empty 4-Litre Jug: (~d, ~d) --> (0, ~d)\n", [X, Y, Y]),
    state(0, Y, [(X, Y) | Steps]). % Record the current state in the steps
```

```

% Empty the 3-Litre Jug
state(X, Y, Steps) :-
    Y > 0,
    \+ visited_state(X, 0),
    assertz(visited_state(X, 0)),
    format("Empty 3-Litre Jug: (~d, ~d) --> (~d, 0)\n", [X, Y, X]),
    state(X, 0, [(X, Y) | Steps]). % Record the current state in the steps

% Pour from 3-Litre to 4-Litre (until full)
state(X, Y, Steps) :-
    X + Y >= 4,
    Y > 0,
    NEW_Y is Y - (4 - X),
    \+ visited_state(4, NEW_Y),
    assertz(visited_state(4, NEW_Y)),
    format("Pour from 3-Litre to 4-Litre: (~d, ~d) --> (4, ~d)\n", [X, Y,
NEW_Y]),
    state(4, NEW_Y, [(X, Y) | Steps]). % Record the current state in the
steps

% Pour from 4-Litre to 3-Litre (until full)
state(X, Y, Steps) :-
    X + Y >= 3,
    X > 0,
    NEW_X is X - (3 - Y),
    \+ visited_state(NEW_X, 3),
    assertz(visited_state(NEW_X, 3)),
    format("Pour from 4-Litre to 3-Litre: (~d, ~d) --> (~d, 3)\n", [X, Y,
NEW_X]),
    state(NEW_X, 3, [(X, Y) | Steps]). % Record the current state in the
steps

% Pour all from 3-Litre to 4-Litre
state(X, Y, Steps) :-
    X + Y < 4,
    Y > 0,
    NEW_X is X + Y,
    \+ visited_state(NEW_X, 0),
    assertz(visited_state(NEW_X, 0)),
    format("Pour all from 3-Litre to 4-Litre: (~d, ~d) --> (~d, 0)\n", [X,
Y, NEW_X]),

```



```

        state(NEW_X, 0, [(X, Y) | Steps]). % Record the current state in the
steps

% Pour all from 4-Litre to 3-Litre
state(X, Y, Steps) :-
    X + Y < 3,
    X > 0,
    NEW_Y is X + Y,
    \+ visited_state(0, NEW_Y),
    assertz(visited_state(0, NEW_Y)),
    format("Pour all from 4-Litre to 3-Litre: (~d, ~d) --> (0, ~d)\n", [X,
Y, NEW_Y]),
    state(0, NEW_Y, [(X, Y) | Steps]). % Record the current state in the
steps

% Predicate to print all steps in the solution path
print_steps([]).
print_steps([(X, Y) | Rest]) :-
    print_steps(Rest),
    format("State --> (~d, ~d)\n", [X, Y]).

```

Output ::

```

% c:/Users/Student/Desktop/waterjug.pl compiled 0.00 sec, 12 clauses
?- fill(0,1).
Fill 4-Litres Jug: ( 0,1 ) -- ( 4,1 )
Fill 3-Litres Jug: ( 4,1 ) -- ( 4,3 )
Empty 4-Litres Jug: ( 4,3 ) -- ( 0,3 )
Empty 3-Litres Jug: ( 0,3 ) -- ( 0,0 )
Fill 4-Litres Jug: ( 0,0 ) -- ( 4,0 )
Pour from 4-Litres to 3-Litres: ( 4,0 ) -- ( 1,3 )
Empty 3-Litres Jug: ( 1,3 ) -- ( 1,0 )
Pour all from 3-Litres to 4-Litres: ( 0,3 ) -- ( 3,0 )
Fill 3-Litres Jug: ( 3,0 ) -- ( 3,3 )
Pour from 3-Litres to 4-Litres: ( 3,3 ) -- ( 4,2 )
Empty 4-Litres Jug: ( 4,2 ) -- ( 0,2 )
Pour all from 3-Litres to 4-Litres: ( 0,2 ) -- ( 2,0 )
Goal reached: ( 2,0 )
State--> (0, 1)
State--> (0, 1)
State--> (4, 1)
State--> (4, 3)
State--> (0, 3)
State--> (3, 0)
State--> (3, 3)
State--> (4, 2)
State--> (0, 2)
true .

```

```

?- fill(0,2).
Fill 4-Litres Jug: ( 0,2 ) -- ( 4,2 )
Fill 3-Litres Jug: ( 4,2 ) -- ( 4,3 )
Empty 4-Litres Jug: ( 4,3 ) -- ( 0,3 )
Empty 3-Litres Jug: ( 0,3 ) -- ( 0,0 )
Fill 4-Litres Jug: ( 0,0 ) -- ( 4,0 )
Pour from 4-Litres to 3-Litres: ( 4,0 ) -- ( 1,3 )
Empty 3-Litres Jug: ( 1,3 ) -- ( 1,0 )
Pour all from 4-Litres to 3-Litres: ( 1,0 ) -- ( 0,1 )
Fill 4-Litres Jug: ( 0,1 ) -- ( 4,1 )
Pour from 4-Litres to 3-Litres: ( 4,1 ) -- ( 2,3 )
Goal reached: ( 2,3 )
State--> (0, 2)
State--> (0, 2)
State--> (4, 2)
State--> (4, 3)
State--> (0, 3)
State--> (0, 0)
State--> (4, 0)
State--> (1, 3)
State--> (1, 0)
State--> (0, 1)
State--> (4, 1)
true .

```

```

?- fill(0,5).
Fill 4-Litres Jug: ( 0,5 ) -- ( 4,5 )
Empty 3-Litres Jug: ( 4,5 ) -- ( 4,0 )
Fill 3-Litres Jug: ( 4,0 ) -- ( 4,3 )
Empty 4-Litres Jug: ( 4,3 ) -- ( 0,3 )
Empty 3-Litres Jug: ( 0,3 ) -- ( 0,0 )
Pour all from 3-Litres to 4-Litres: ( 0,3 ) -- ( 3,0 )
Fill 3-Litres Jug: ( 3,0 ) -- ( 3,3 )
Pour from 3-Litres to 4-Litres: ( 3,3 ) -- ( 4,2 )
Empty 4-Litres Jug: ( 4,2 ) -- ( 0,2 )
Pour all from 3-Litres to 4-Litres: ( 0,2 ) -- ( 2,0 )
Goal reached: ( 2,0 )
State--> (0, 5)
State--> (0, 5)
State--> (4, 5)
State--> (4, 0)
State--> (4, 3)
State--> (0, 3)
State--> (3, 0)
State--> (3, 3)
State--> (4, 2)
State--> (0, 2)
true ■

```

EXPERIMENT - 6

AIM :: Write a program to implement Tic-Tac-Toe game.

Program ::

```
% Define the initial empty board
initial_board([empty, empty, empty, empty, empty, empty, empty, empty, empty]).

% Start the game
start_game :-
    initial_board(Board),
    play_game(Board, x).

% Game loop
play_game(Board, Player) :-
    print_board(Board),
    ( winner(Board, x) -> write('Player X wins!'), nl
    ; winner(Board, o) -> write('Player O wins!'), nl
    ; full_board(Board) -> write('It\'s a draw!'), nl
    ; make_move(Board, Player, NewBoard),
      next_player(Player, NextPlayer),
      play_game(NewBoard, NextPlayer)
    ).

% Print the Tic-Tac-Toe board
print_board([A, B, C, D, E, F, G, H, I]) :-
    write(' '), display_symbol(A), write(' | '), display_symbol(B), write('
| '), display_symbol(C), nl,
    write('-----'), nl,
    write(' '), display_symbol(D), write(' | '), display_symbol(E), write('
| '), display_symbol(F), nl,
    write('-----'), nl,
    write(' '), display_symbol(G), write(' | '), display_symbol(H), write('
| '), display_symbol(I), nl, nl.

% Display 'X', 'O', or an empty position
display_symbol(empty) :- write(' ').
display_symbol(X) :- write(X).

% Check if a player has won
winner(Board, Player) :-
```

```

    ( row_win(Board, Player)
    ; col_win(Board, Player)
    ; diag_win(Board, Player)
    ).

% Check winning row conditions
row_win([A, B, C, _, _, _, _, _, _], Player) :- A = Player, B = Player, C =
Player.
row_win([_, _, _, D, E, F, _, _, _], Player) :- D = Player, E = Player, F =
Player.
row_win([_, _, _, _, _, _, G, H, I], Player) :- G = Player, H = Player, I =
Player.

% Check winning column conditions
col_win([A, _, _, D, _, _, G, _, _], Player) :- A = Player, D = Player, G =
Player.
col_win([_, B, _, _, E, _, _, H, _], Player) :- B = Player, E = Player, H =
Player.
col_win([_, _, C, _, _, F, _, _, I], Player) :- C = Player, F = Player, I =
Player.

% Check winning diagonal conditions
diag_win([A, _, _, _, E, _, _, _, I], Player) :- A = Player, E = Player, I =
Player.
diag_win([_, _, A, _, E, _, I, _, _], Player) :- A = Player, E = Player, I =
Player.

% Check if the board is full
full_board(Board) :-
    \+ member(empty, Board).

% Make a move for the current player
make_move(Board, Player, NewBoard) :-
    repeat,
    write('Enter a position (1-9): '), read(Pos),
    ( valid_move(Board, Pos) -> replace(Pos, Player, Board, NewBoard), !
    ; write('Invalid move! Try again.'), nl, fail
    ).

% Validate the move (position must be between 1-9 and must be empty)
valid_move(Board, Pos) :-
    integer(Pos), Pos >= 1, Pos <= 9, nth1(Pos, Board, empty).

```

```
% Replace a position on the board with the player's symbol
replace(Pos, Player, Board, NewBoard) :-
    nth1(Pos, Board, empty, TempBoard),
    nth1(Pos, NewBoard, Player, TempBoard).

% Switch to the next player
next_player(x, o).
next_player(o, x).
```

Output ::

```
% c:/Users/Student/Desktop/
?- start_game.
  |  | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): 1.
x |  | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 4.
x |  | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 2.
x | x | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 5.
x | x | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 3.
x | x | x
  |  | 
  |  | 
  |  | 
Player X wins!
true .
```

```
?- start_game.
  |  | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): 1.
x |  | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 2.
x | o | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 4.
x | o | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 6.
x | o | 
  |  | 
  |  | 
  |  | 
Enter a position (1-9): |: 7.
x | o | 
  |  | 
  |  | 
  |  | 
Player X wins!
true .
```

?- start_game.

Enter a position (1-9): 1.

x		

Enter a position (1-9): |: 2.

x		o	

Enter a position (1-9): |: 5.

x		o	

	x		

Enter a position (1-9): |: 3.

x		o		o

	x			

Enter a position (1-9): |: 9.

x		o		o

	x			

			x	

Player X wins!

true .

EXPERIMENT - 7

AIM :: Write a Prolog program to remove Punctuations from text.

Program ::

```
% Define punctuation characters by their ASCII codes
is_punctuation(Char) :-
    member(Char, [46, 44, 59, 58, 33, 63, 39, 34, 45, 40, 41, 91, 93, 123,
125]).

% Predicate to remove punctuations from a list of character codes
remove_punctuations([], []). % Base case: empty list returns empty list

remove_punctuations([H | T], Result) :-
    is_punctuation(H), % If the character is punctuation, skip it
    remove_punctuations(T, Result).

remove_punctuations([H | T], [H | Result]) :- % Otherwise, keep the
character
    \+ is_punctuation(H),
    remove_punctuations(T, Result).

% Convert input string to output without punctuations
clean_string(Input, Output) :-
    string_codes(Input, Codes), % Convert string to list of character
codes
    remove_punctuations(Codes, CleanCodes), % Remove punctuation marks
    string_codes(Output, CleanCodes). % Convert back to string

% Main program that takes user input and displays cleaned string
start :-
    write('Enter a sentence: '), nl,
    read_line_to_string(user_input, Input), % Read input from user
    clean_string(Input, Output),
    write('Original String: '), write(Input), nl,
    write('Cleaned String: '), write(Output), nl.
```

Output ::

```
.
% c:/Users/Desktop/sample/prolog/exp7.pl compiled 0.00 sec, 6 clauses
?- start.
Enter a sentence:
|: "Hello, World! I, am, Gautam."
Original String: "Hello, World! I, am, Gautam."
Cleaned String: Hello World I am Gautam
true .

?- start.
Enter a sentence:
|: this, was" game , look ! at you"!
Original String: this, was" game , look ! at you"!
Cleaned String: this was game look at you
true .

?- start.
Enter a sentence:
|: This, is, rock. That's I am taling!!!!"
Original String: This, is, rock. That's I am taling!!!!"
Cleaned String: This is rock Thats I am taling
true .

?- start.
Enter a sentence:
|: This's is, a, game!!!"
Original String: This's is, a, game!!!"
Cleaned String: Thiss is a game
true ■

?- start.
Enter a sentence:
|:
false.
.
```


EXPERIMENT - 8

AIM :: Write a Prolog program to sort a sentence

Program ::

```
% Main predicate to read input and sort the sentence
sort_sentence :-
    write('Enter a sentence: '), nl,
    read_line_to_string(user_input, Sentence), % Read sentence from user
input
    split_string(Sentence, " ", "", Words),    % Split the sentence into
words
    msort(Words, SortedWords),                  % Sort words alphabetically
    atomic_list_concat(SortedWords, ' ', SortedSentence), % Combine sorted
words back into a sentence
    write('Sorted sentence: '), writeln(SortedSentence).

% To run:
% ?- sort_sentence.
% Then input your sentence when prompted.
```

Output ::

```
% c:/Users/sample/prolog/exp8.pl compiled 0.00 sec, 1 clauses |
.
?- sort_sentence.
Enter a sentence: Today is monday.
Sorted sentence: Today is monday.
true.

?- sort_sentence.
Enter a sentence: It is Aman, a good boy.
Sorted sentence: Aman, It a boy. good is
true.

?- sort_sentence.
Enter a sentence: Can I be a good Doctor, with only this.
Sorted sentence: Can Doctor, I a be good only this. with
true.

?- sort_sentence.
Enter a sentence: I can be a teacher for you childrens and friend.
Sorted sentence: I a and be can childrens for friend. teacher you
true.

?-
```

EXPERIMENT - 9

AIM :: Write a program to implement Hangman game using Python.

Program ::

```
import random

# List of words to choose from
words = ['python', 'java', 'hangman', 'computer', 'programming',
'developer']

# Function to choose a random word from the list
def choose_word():
    return random.choice(words)

# Function to display the current state of the word
def display_word(word, guessed_letters):
    display = ""
    for letter in word:
        if letter in guessed_letters:
            display += letter
        else:
            display += "_"
    return display

# Main hangman game function
def hangman():
    word = choose_word()
    guessed_letters = []
    attempts = 6 # Number of incorrect attempts before the game is over
    print("Welcome to Hangman!")
    print("Try to guess the word, you have", attempts, "attempts.")

    while attempts > 0:
        print("\nWord: ", display_word(word, guessed_letters))
        print(f"Guessed letters: {'', ''.join(guessed_letters)}")
        print(f"Attempts left: {attempts}")

        guess = input("Enter a letter: ").lower()

        # Validate the input
        if len(guess) != 1 or not guess.isalpha():
```

```
        print("Please enter a single valid letter.")
        continue

    # Check if the letter has already been guessed
    if guess in guessed_letters:
        print("You've already guessed that letter!")
        continue

    # Add the guessed letter to the guessed_letters list
    guessed_letters.append(guess)

    # Check if the guessed letter is in the word
    if guess not in word:
        attempts -= 1
        print(f"Wrong guess! {guess} is not in the word.")

    # Check if the user has guessed the word
    if all(letter in guessed_letters for letter in word):
        print(f"Congratulations! You've guessed the word: {word}")
        break
else:
    print(f"Game Over! The word was: {word}")

# Run the hangman game
if __name__ == "__main__":
    hangman()
```

Output ::

```
Welcome to Hangman!
Try to guess the word, you have 6 attempts.

Word: _____
Guessed letters:
Attempts left: 6
Enter a letter: n

Word: __n__n
Guessed letters: n
Attempts left: 6
Enter a letter: d
Wrong guess! d is not in the word.

Word: __n__n
Guessed letters: n, d
Attempts left: 5
Enter a letter: h

Word: h_n__n
Guessed letters: n, d, h
Attempts left: 5
Enter a letter: a

Word: han__an
Guessed letters: n, d, h, a
Attempts left: 5
Enter a letter: g

Word: hang__an
Guessed letters: n, d, h, a, g
Attempts left: 5
Enter a letter: m
Congratulations! You've guessed the word: hangman
```

EXPERIMENT - 10

AIM :: Write a program to implement Hangman game using PROLOG.

Program ::

```
:- dynamic word/1.word(apple). % Define the word to guess

% Start the game
hangman :-
    word(Word),
    atom_chars(Word, WordList),
    length(WordList, N),
    play(WordList, N, [], 6). % Max 6 wrong attempts

% Play function
play(WordList, _, Guessed, 0) :-
    write('Game Over! You ran out of attempts. '), nl,
    write('The correct word was: '), write(WordList), nl, !.

play(WordList, _, Guessed, _) :-
    word_complete(WordList, Guessed),
    write('Congratulations! You guessed the word: '), write(WordList),
    nl, !.

play(WordList, N, Guessed, Attempts) :-
    display_progress(WordList, Guessed),
    write('Enter your guess: '),
    read(Char),
    ( member(Char, WordList) ->
        write('Correct Guess! '), nl,
        play(WordList, N, [Char|Guessed], Attempts)
    ;
        write('Wrong Guess! '), nl,
        NewAttempts is Attempts - 1,
        write('Remaining Attempts: '), write(NewAttempts), nl,
        play(WordList, N, Guessed, NewAttempts)
    ).

% Check if all letters are guessed
word_complete([], _).
word_complete([H|T], Guessed) :-
```

```

        member(H, Guessed),
        word_complete(T, Guessed).

% Display word progress
display_progress([], _):- nl.
display_progress([H|T], Guessed) :-
    ( member(H, Guessed) -> write(H) ; write('_') ),
    write(' '),
    display_progress(T, Guessed).

% Start game
?- hangman.

```

Output ::

```

Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.9)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software. Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

% c:/Users/maitl/Documents/Prolog/hangman.pl compiled 0.00 sec, 2 clauses

?- hangman.
_ _ _ _ _
Enter your guess: a.
Correct Guess!
a _ _ _ _
Enter your guess: e.
Correct Guess!
a _ _ _ e
Enter your guess: x.
Wrong Guess!
Remaining Attempts: 5
a _ _ _ e
Enter your guess: p.
Correct Guess!
a p p _ e
Enter your guess: l.
Correct Guess!
a p p l e
Congratulations! You guessed the word: [a, p, p, l, e].

```

EXPERIMENT - 11

AIM :: Write a program to remove stop words for a given passage from a text file using NLTK.

Program ::

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Function to remove stopwords from text
def remove_stopwords(text):
    stop_words = set(stopwords.words('english')) # Load NLTK's stop words
    words = word_tokenize(text) # Tokenize text into words
    filtered_words = [word for word in words if word.lower() not in
stop_words]
    return ' '.join(filtered_words) # Join words back into a sentence

# Read text from file
input_file = "input.txt"
with open(input_file, "r") as file:
    text = file.read()

# Process text to remove stopwords
filtered_text = remove_stopwords(text)

# Write the cleaned text to output file
output_file = "output.txt"
with open(output_file, "w") as file:
    file.write(filtered_text)

# Print result
print("Original Text:\n", text)
print("Filtered Text:\n", filtered_text)
```

Output ::

```
(myenv) Code$ python3 11_remove_stop_words.py
```

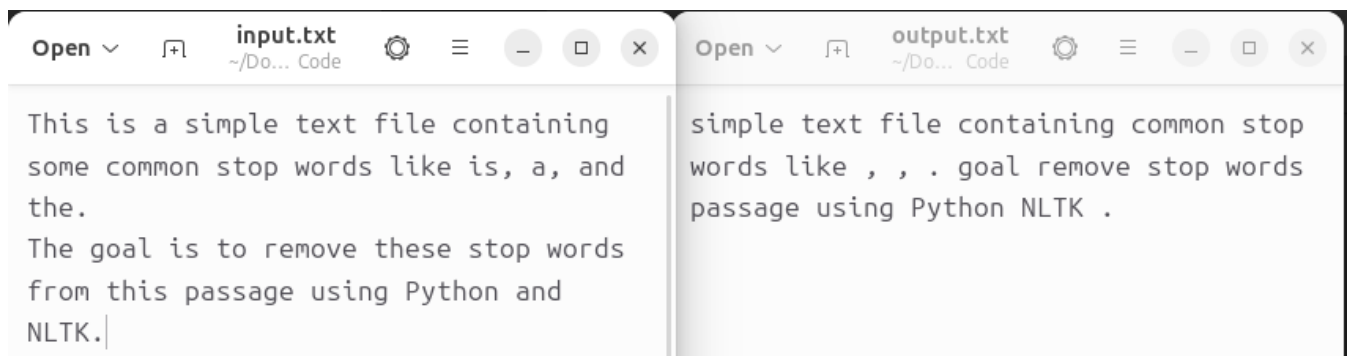
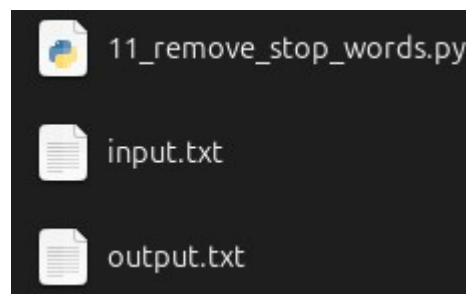
Original Text:

This is a simple text file containing some common stop words like is, a, and the.

The goal is to remove these stop words from this passage using Python and NLTK.

Filtered Text:

simple text file containing common stop words like , , . goal remove stop words passage using Python NLTK .



EXPERIMENT - 12

AIM :: Write a program to implement stemming for a given sentence using NLTK.

Program ::

```
import nltk
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Function to perform stemming
def stem_sentence(sentence):
    # Initialize the Porter Stemmer
    porter_stemmer = PorterStemmer()

    # Tokenize the sentence into words
    words = word_tokenize(sentence)

    # Stem each word in the sentence
    stemmed_words = [porter_stemmer.stem(word) for word in words]

    # Join the stemmed words back into a sentence
    return ' '.join(stemmed_words)

# Example usage
sentence = "The cat is jumping and dogs are running fast."
stemmed_sentence = stem_sentence(sentence)

# Print the original and stemmed sentences
print("Original Sentence: ", sentence)
print("Stemmed Sentence: ", stemmed_sentence)
```

Output ::

```
● (myenv) Code$ python3 12_stemming_for_sentence.py
Original Sentence: The cat is jumping and dogs are running fast.
Stemmed Sentence: the cat is jump and dog are run fast .
```

EXPERIMENT - 13

AIM :: Write a program to POS (part of speech) tagging for the give sentence using NLTK.

Program ::

```
import nltk
from nltk.tokenize import word_tokenize
from nltk import pos_tag

# Function to perform POS tagging
def pos_tagging(sentence):
    # Tokenize the sentence into words
    words = word_tokenize(sentence)

    # Perform POS tagging
    tagged_words = pos_tag(words)

    return tagged_words

# Example usage
sentence = "The cat is jumping and dogs are running fast."
tagged_sentence = pos_tagging(sentence)

# Print the original sentence and its POS tags
print("Original Sentence: ", sentence)
print("POS Tagged Sentence: ", tagged_sentence)
```

Explanation of POS Tags:

- **DT:** Determiner (e.g., "The")
- **NN:** Noun, singular (e.g., "cat")
- **VBZ:** Verb, 3rd person singular present (e.g., "is")
- **VBG:** Verb, gerund or present participle (e.g., "jumping", "running")
- **CC:** Coordinating conjunction (e.g., "and")
- **NNS:** Noun, plural (e.g., "dogs")
- **VBP:** Verb, non-3rd person singular present (e.g., "are")
- **RB:** Adverb (e.g., "fast")

Output ::

- (myenv) **Code**\$ python3 13_POS_tagging.py

Original Sentence: The cat is jumping and dogs are running fast.

POS Tagged Sentence: [('The', 'DT'), ('cat', 'NN'), ('is', 'VBZ'), ('jumping', 'VBG'), ('and', 'CC'), ('dogs', 'NNS'), ('are', 'VBP'), ('running', 'VBG'), ('fast', 'RB'), ('.', '.')]]

EXPERIMENT - 14

AIM :: Write a program to implement Lemmatization using NLTK.

Program ::

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Initialize the WordNet Lemmatizer
lemmatizer = WordNetLemmatizer()

# Function to perform Lemmatization
def lemmatize_sentence(sentence):
    # Tokenize the sentence into words
    words = word_tokenize(sentence)

    # Lemmatize each word
    lemmatized_words = [lemmatizer.lemmatize(word, pos='v') for word in words] # 'v' for verb, you can change it depending on the word
    return ' '.join(lemmatized_words)

# Example usage
sentence = "The cats are running and they are better than others."
lemmatized_sentence = lemmatize_sentence(sentence)

# Print the original and lemmatized sentences
print("Original Sentence: ", sentence)
print("Lemmatized Sentence: ", lemmatized_sentence)
```

Output ::

```
• (myenv) Code$ python3 14_lemmatization.py
Original Sentence: The cats are running and they are better than others.
Lemmatized Sentence: The cat be run and they be better than others .
```

EXPERIMENT - 15

AIM :: Write a program for Text Classification for the given sentence using NLTK.

Program ::

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.classify import NaiveBayesClassifier
from nltk.classify.util import accuracy

# Sample labeled data for training the classifier
training_data = [
    ("I love programming", "positive"),
    ("I hate bugs", "negative"),
    ("This is so great", "positive"),
    ("I am so angry at this error", "negative"),
    ("Coding is awesome", "positive"),
    ("This is terrible", "negative")
]

# Preprocess the data: tokenization and removing stopwords
stop_words = set(stopwords.words("english"))

def preprocess(text):
    tokens = word_tokenize(text.lower()) # Tokenize and convert to lowercase
    return {word: True for word in tokens if word not in stop_words and word.isalpha()}

# Feature extraction
train_set = [(preprocess(text), label) for text, label in training_data]

# Train the Naive Bayes classifier
classifier = NaiveBayesClassifier.train(train_set)

# Test the classifier with a new sentence
def classify_sentence(sentence):
    features = preprocess(sentence)
    return classifier.classify(features)
```

```
# Example usage
test_sentence = "I love coding but debugging is hard."
print(f"Sentence: {test_sentence}")
print(f"Classification: {classify_sentence(test_sentence)}")

# Print classifier accuracy on training data
print(f"Classifier accuracy: {accuracy(classifier, train_set) * 100}%")
```

Output ::

```
• (myenv) Code$ python3 15_text_classification.py
Sentence: I love coding but debugging is hard.
Classification: positive
Classifier accuracy: 100.0%
```