**LAB FILE**

**ARTIFICIAL INTELLIGENCE LAB**

**(AI‐302P)**

Student Name: Lakshay Sharma

Roll No: 02396402721

Semester: 6th

Group: 6-CSE-AIML-II-C

Faculty name: Mr. Saurabh Rastogi



Department Of Computer Science & Engineering

Maharaja Agrasen Institute of Technology, PSP area, Sector – 22, Rohini,

New Delhi – 110085

(Affiliated to Guru Gobind Singh Indraprastha University, New Delhi)

**2024**



## MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

**VISION**

To nurture young minds in a learning environment of high academic value and imbibe spiritual and ethical values with technological and management competence.

## MISSION

The Institute shall endeavor to incorporate the following basic missions in the teaching methodology:

**Engineering Hardware – Software Symbiosis**

Practical exercises in all Engineering and Management disciplines shall be carried out by Hardware equipment as well as the related software enabling deeper understanding of basic concepts and

encouraging inquisitive nature.

**Life – Long Learning**

The Institute strives to match technological advancements and encourage students to keep updating their knowledge for enhancing their skills and inculcating their habit of continuous learning.

**Liberalization and Globalization**

The Institute endeavors to enhance technical and management skills of students so that they are intellectually capable and competent professionals with Industrial Aptitude to face the challenges of globalization.

**Diversification**

The Engineering, Technology and Management disciplines have diverse fields of studies with different attributes. The aim is to create a synergy of the above attributes by encouraging analytical thinking.

**Digitization of Learning Processes**

The Institute provides seamless opportunities for innovative learning in all Engineering and Management disciplines through digitization of learning processes using analysis, synthesis, simulation, graphics,

tutorials and related tools to create a platform for multi-disciplinary approach.

**Entrepreneurship**

The Institute strives to develop potential Engineers and Managers by enhancing their skills and research capabilities so that they become successful entrepreneurs and responsible citizens.



## MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

**COMPUTER SCIENCE & ENGINEERING DEPARTMENT**

# VISION

“To be centre of excellence in education, research and technology transfer in the field of computer engineering and promote entrepreneurship and ethical values.”

# MISSION

“To foster an open, multidisciplinary and highly collaborative research environment to produce world-class engineers capable of providing innovative solutions to real life problems and fulfil societal needs.”

Department of Computer Science and Engineering Rubrics for Lab Assessment

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Rubrics** | | **0** | **1** | **2** | **3** |
| **Missing** | **Inadequate** | **Needs Improvement** | **Adequate** |
| R1 | Is able to identify the problem to be solved and define the objectives of the experiment. | No mention is made of the problem to be solved. | An attempt is made to identify the problem to be solved but it is described in a confusing manner, objectives are not relevant, objectives contain technical/ conceptual errors or objectives are not measurable. | The problem to be solved is described but there are minor omissions or vague details. Objectives are conceptually correct and measurable but may be incomplete in scope or have linguistic errors. | The problem to be solved is clearly stated. Objectives are complete, specific, concise, and measurable. They are written using correct technical terminology and are free from linguistic errors. |
| R2 | Is able to design a reliable experiment that solves the problem. | The experiment does not solve the problem. | The experiment attempts to solve the problem but due to the nature of the design the data will not lead to a reliable solution. | The experiment attempts to solve the problem but due to the nature of the design there is a moderate chance the data will not lead to a reliable  solution. | The experiment solves the problem and has a high likelihood of producing data that will lead to a reliable solution. |
| R3 | Is able to communicate the details of an experimental procedure clearly and completely. | Diagrams are missing and/or experimental procedure is missing or extremely vague. | Diagrams are present but unclear and/or experimental procedure is present but important details are missing. | Diagrams and/or experimental procedure are present but with minor omissions or vague details. | Diagrams and/or experimental procedure are clear and complete. |
| R4 | Is able to record and represent data in a meaningful way. | Data are either absent or incomprehensible. | Some important data are absent or incomprehensible. | All important data are present, but recorded in a way that requires some effort to comprehend. | All important data are present, organized and recorded clearly. |
| R5 | Is able to make a judgment about the results of the experiment. | No discussion is presented about the results of the experiment . | A judgment is made about the results, but it is not reasonable or coherent. | An acceptable judgment is made about the result, but the reasoning is flawed or incomplete. | An acceptable judgment is made about the result, with clear reasoning. The effects of assumptions and experimental uncertainties are considered. |

# PRACTICAL RECORD

**PAPER CODE : AI‐302P**

Name of Student : Lakshay Sharma

University roll : 02396402721

Semester : 6th (3rd year)

Group : 6-CSE-AIML-II-C

## PRACTICAL DETAILS

Experiments according to the lab syllabus prescribed by GGSIPU

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Exp. no** | **Experiment Name** | **Date of performance** | **Date of checking** | **R1 (3)** | **R2 (3)** | **R3 (3)** | **R4 (3)** | **R5 (3)** | **Total Marks (15)** | **Signature** |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

Experiments according to the lab syllabus prescribed by GGSIPU

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Exp. no** | **Experiment Name** | **Date of Performance** | **Date of checking** | **R1 (3)** | **R2 (3)** | **R3 (3)** | **R4 (3)** | **R5 (3)** | **Total Marks (15)** | **Signature** |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

# EXPERIMENT – 1

Date: \_\_\_\_\_\_\_\_

### AIM: Study of PROLOG

### THEORY:

**Introduction:**

Prolog is a logic programming language that is primarily used for artificial intelligence applications. It is based on first-order predicate calculus and uses a restricted version of the clausal form known as Horn clause form. In Prolog, logic is expressed as relations (called facts and rules). A fact is a statement that is true, while a rule is a conditional statement. A query is the end goal that is derived from the AI's analysis of facts and rules.

To use Prolog, you need to define facts and rules that represent the knowledge you want to work with. For example, you might define a fact that represents a person's name and age:

person(john, 30).

This fact can be interpreted as "John is 30 years old."

You can also define rules that represent relationships between facts. For example, you might define a rule that represents the relationship between a person's name and their age:

age(X, Y) :- person(X, Y).

This rule can be interpreted as "The age of a person is equal to the age specified in the person fact."

Once you have defined your facts and rules, you can query the knowledge base to find information. For example, you might query the knowledge base to find the age of a person:

?- age(john, Y).

This query will return the age of John, which is 30.

Prolog features include unification, backtracking, and recursion. Unification is the process of matching variables in a query with variables or constants in the knowledge base. Backtracking is the process of undoing previous steps in a computation to explore alternative paths. Recursion is the process of calling a function or procedure within itself.

Prolog is used in various applications, including artificial intelligence, natural language processing, and knowledge representation. It is also used for pattern matching over natural language parse trees and in the development of knowledge-based systems.

**Symbolic Language:**

A symbolic language, such as Prolog, is a type of programming language that emphasizes symbolic manipulation and reasoning over explicit step-by-step instructions. In Prolog, logic is expressed in terms of relations, facts, and rules. The program in Prolog consists of a set of facts and rules that define relationships and logic within a domain. When running a Prolog program, a query is posed to the program, and the Prolog engine uses logical inference to find solutions based on the defined relations.

Prolog's symbolic nature allows it to excel in tasks involving knowledge representation, natural language processing, expert systems, and rule-based systems. Unlike traditional imperative programming languages like Java or C++, where you specify how to solve a problem step by step, in Prolog, you define what the problem is through relationships and rules, letting the language's inference engine handle the execution.

Prolog uses a "predicate logic" syntax to represent facts, rules, and queries. Facts represent knowledge about the domain, rules define relationships and logic, and queries are used to ask questions or retrieve information from the knowledge base. The language's backtracking feature is crucial as it allows Prolog to explore alternative solutions when a query fails.

Overall, Prolog's symbolic nature and logic programming paradigm make it well-suited for tasks that involve symbolic or non-numeric computation. It automates the process of solving problems based on logical relationships defined within the program.

**Facts, Rules and Queries:**

In Prolog, the fundamental components are facts, rules, and queries, which form the building blocks of logic programming.

**Facts:**

Facts in Prolog are statements that describe relationships between objects or state the existence of objects. They are used to represent knowledge about the domain. Facts are also known as "unconditional horn clauses" and are always true.

In Prolog, facts are defined using a specific syntax. The syntax for defining a fact consists of three parts: the relationship name, the property name, and the object.

1. Relationship Name: The relationship name appears as the first term in the fact. It represents the relationship between the property and the object. For example, in the fact `likes(john, chocolate)`, the relationship name is "likes".

2. Property Name: The property name is the second term in the fact. It represents the property of the object. In the example `likes(john, chocolate)`, the property name is "john".

3. Object: The object is the third term in the fact. It represents the value of the property. In the example `likes(john, chocolate)`, the object is "chocolate".

Here's a breakdown of the syntax for defining a fact:

relationship\_name(property\_name, object).

For example:

likes(john, chocolate).

This fact states that John likes chocolate.

The names of properties start with lowercase letters. The relationship name appears as the first term, and the object is comma-separated arguments within parentheses. This syntax allows for clear and concise representation of relationships between objects and their properties.

**Rules:**

In Prolog, **Rules** are essential components that establish logical relationships between facts.

Rules in Prolog define logical implications or deductions based on existing facts. They consist of a head (conclusion) and a body (conditions) that must be satisfied for the rule to be applicable.

Example:

happy(lili) :- dances(lili).

This rule states that if Lili dances, then she is happy.

Here, happy(lili) is the conclusion (head), and dances(lili) is the condition (body).

Usage:

Rules in Prolog enable the formulation of more complex logical deductions beyond simple facts. By defining rules, programmers can infer new information based on the existing knowledge base.

Rules allow for conditional reasoning, where certain conditions need to be met for a conclusion to hold true.

In Prolog, when a query is posed, the system not only matches facts but also evaluates rules to derive answers. The logical inference mechanism in Prolog processes these rules to make deductions and provide responses to queries based on the defined relationships.

Rules are powerful tools in Prolog programming as they facilitate the creation of sophisticated logical structures and enable the system to draw conclusions from the available information. They enhance the expressiveness and reasoning capabilities of Prolog programs, making them suitable for complex problem-solving tasks in various domains.

By leveraging rules alongside facts and queries, programmers can build robust and intelligent systems that can reason logically and derive new insights from the knowledge base effectively.

**Queries:**

In Prolog, Queries play a vital role in extracting information from the knowledge base by posing questions about relationships between objects and their properties. Let's explore queries in Prolog in more detail:

Queries in Prolog are essentially questions that seek to uncover information stored within the knowledge base. They are used to inquire about relationships between objects and properties defined by facts and rules.

Examples:

1: Is Tom a cat?

2: Does Kunal love to eat pasta?

Functionality:

Prolog's logic programming language processes queries by matching them against the defined relations within the knowledge base. The Prolog engine uses logical inference to find answers to these queries based on the established facts and rules.

In Prolog, when a query is posed, the system searches for matching facts or rules that can provide an answer. The process involves unifying variables with values and applying logical reasoning to derive the response. Queries allow users to interact with the knowledge base dynamically, extracting specific information based on their inquiries.

Prolog's ability to process queries efficiently and derive answers based on logical relationships makes it a powerful tool for knowledge representation and reasoning tasks in various domains

**Predicate, Argument and Clauses:**

**Predicates:**

Predicates in Prolog represent relationships or functions. They consist of a functor (predicate name) and arguments that define the properties or attributes of objects.

Predicates can have different arities (number of arguments) that determine how many values need to be passed for the predicate to be evaluated.

**Arguments:**

Arguments are the values passed to predicates in Prolog clauses.

They represent the entities or objects involved in the relationships defined by predicates.

In Prolog, arguments are enclosed in parentheses and separated by commas within predicates.

**Clauses:**

Clauses are the building blocks of Prolog programs. They consist of a head (predicate being defined) and a body (conditions that must be satisfied for the clause to be true).

Example:

happy(lili) :- dances(lili).

Here, happy(lili) is the head, and dances(lili) is the body.

Usage:

Clauses define relationships between objects and their properties, allowing for logical deductions based on specified conditions.

In Prolog programs, clauses define logical implications or rules that govern the relationships between objects. By specifying predicates with appropriate arguments, programmers can model complex relationships and define logical deductions based on these relationships.

Understanding how clauses, arguments, and predicates work together is crucial for writing efficient Prolog programs that can perform logical reasoning and inference tasks effectively. These concepts form the foundation of knowledge representation and reasoning in Prolog programming.

**Advantages and Dis advantages of Prolog:**

*Advantages of Prolog*

- Built-in List Handling: Prolog provides built-in list handling capabilities, making it efficient for representing sequences, trees, and other data structures.

- Ease of Programming: Writing and reading programs that build complex structures is straightforward in Prolog.

- Declarative Programming: Prolog's declarative nature focuses on "what" needs to be achieved rather than the specific steps to achieve it.

- Logic Programming: Prolog's logic programming paradigm allows for elegant and concise representation of complex relationships and rules, making it well-suited for certain problem domains.

- Knowledge-Driven Solutions: Prolog excels in scenarios where a knowledge base forms a significant part of the solution, such as expert systems and natural language processing.

- Backtracking: Prolog's backtracking mechanism enables exploring multiple solutions to a problem, making it ideal for tasks with multiple possible outcomes.

- Natural Language Interpretation: Prolog is well-suited for natural language interpretation tasks due to its ability to represent logical rules effectively.

- Unification Engine: Implementing a unification engine in Prolog can provide a clean and general solution for certain types of problems, enhancing program reliability.

- Ambiguity Resolution: Prolog is effective in handling ambiguity, especially in tasks like natural language processing, where multiple interpretations need to be considered.

*Disadvantages of Prolog*

- Limited Performance: Prolog may not be optimal for performance-critical applications due to its execution model.

- Steep Learning Curve: Understanding Prolog's syntax and logical paradigm can be challenging for newcomers.

- Not Suitable for All Applications: While ideal for symbolic reasoning, Prolog may not be suitable for tasks requiring extensive data manipulation or real-time processing.

- Limited Libraries: Prolog has a smaller ecosystem of libraries compared to mainstream languages, potentially limiting pre-existing solutions.

- Lack of Concurrency Support: Traditional Prolog lacks robust support for concurrency and parallel processing.

- Debugging Challenges: Debugging complex Prolog programs can be more challenging due to the language's declarative nature and backtracking mechanism.

- Limited Community Support: The smaller user community may make finding resources and support more difficult.

- Inefficient Data Structures: Built-in data structures in Prolog may not be as efficient as those in languages designed for data manipulation.

- Performance Bottlenecks: Backtracking in Prolog can lead to performance issues in programs with complex rule sets or deep search trees.

- Portability Issues: Despite an ISO standard, portability challenges may arise between different Prolog implementations.

**Expert Systems in Prolog:**

Expert systems, which emulate the decision-making ability of human experts, find a natural fit in Prolog due to its logic programming paradigm. Here is an in-depth explanation of expert systems in Prolog:

1. **Logic Programming Paradigm**

- Logic Programming: Prolog's logic programming paradigm aligns well with the rule-based nature of expert systems, allowing for the representation of knowledge through logical rules and relationships.

1. **Knowledge Representation**

- Knowledge Base: In Prolog, expert systems are built on a knowledge base that consists of rules and facts representing domain-specific knowledge.

- Inference Engine: Prolog's inference engine processes these rules to derive conclusions based on the given input.

1. **Backward Chaining**

- Backward Chaining: Prolog's built-in backward chaining inference engine is commonly used to implement expert systems.

- Rule-Based Reasoning: Expert systems in Prolog use rules for knowledge representation, where each rule has a goal and sub-goals that are proven or disproven during inference.

1. **User Interaction**

- User Interface: Expert systems in Prolog often involve user interactions to gather information or provide recommendations based on the system's knowledge base.

- Ask Predicate: The `ask` predicate is used to interact with users, prompting them for input to infer conclusions.

1. **Knowledge Acquisition**

- Knowledge Base Development: Developing a knowledge base involves capturing the expertise of human experts in the form of rules and facts.

- Rule-Based System: Expert systems in Prolog operate on a collection of rules that encapsulate domain-specific knowledge, forming the system's knowledge base.

1. **Practical Applications**

- Domain-Specific Solutions: Expert systems in Prolog are commonly used in domains where rule-based reasoning and logical inference are crucial, such as medical diagnosis, fault detection, and decision support systems.

**Meta-program:**

Meta-programming in Prolog is a powerful technique that allows the manipulation of programs themselves. Prolog's unique data structures, which are used to represent both programs and data, make it well-suited for writing meta-programs. Here is an in-depth look at meta-programming in Prolog:

Basics of Meta-Programming:

- Programming Technique: Meta-programming enables the manipulation of programs within Prolog.

- Data Structures: Prolog uses terms as its basic data structure, allowing for the representation of programs and data in a unified manner.

Predicates for Meta-Programming:

1. =.. Predicate: The =.. predicate is used to decompose a term into a functor and its arguments, enabling the construction of Prolog goals dynamically.
2. call Predicate: The call predicate is crucial for executing arbitrary Prolog goals constructed using ‘=..’ .

Meta-Interpreters:

- Meta-Interpreters: In Prolog, meta-interpreters are used to interpret and manipulate Prolog programs within Prolog itself.

- Guarded Prolog: Meta-interpreters can extend the behavior of Prolog to handle guards, allowing for more sophisticated rule-based systems.

Use Cases:

- Proof Explainers: Meta-interpreters are valuable for explaining proofs and tracing program execution.

- Changing Search Strategies: They can be used to modify proof search strategies, such as switching between breadth-first and depth-first search.

Meta-programming in Prolog opens up a realm of possibilities for dynamic program manipulation, advanced rule-based systems, and enhanced program understanding. By leveraging meta-programming techniques, developers can build sophisticated applications that manipulate and reason about programs at runtime.

# EXPERIMENT – 2

Date: \_\_\_\_\_\_\_\_

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

a. Ram likes mango.

b. Seema is a girl.

c. Bill likes Cindy.

d. Rose is red.

e. John owns gold.

## CLAUSES:

likes(ram, mango).

girl(seema).

likes(bill, cindy).

red(rose).

owns(john, gold).

## QUERIES:

1. | ?- likes(ram, What).

What = mango

Yes

1. | ?- likes(balram, What).

No

1. | ?- girl(Who).

Who = seema

yes

1. | ?- likes(Who, cindy).

Who = bill

Yes

1. | ?- red(What).

What = rose

Yes

1. | ?- green(What).

uncaught exception: error(existence\_error(procedure,green/1),top\_level/0)

1. | ?- owns(Who, What).

What = gold

Who = jhon

yes

# **EXPERIMENT – 3**

Date: \_\_\_\_\_\_\_\_\_

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

## RULES:

Arithmetic Rules:

* Convert Celsius to Fahrenheit: F is C \* 9 / 5 + 32
* Determine if temperature is freezing: F <= 32

**Rules:-**

c\_to\_f(C, F) :-

F is C \* 9 / 5 + 32.

freezing(F) :-

F =< 32.

## QUERIES: -

1. | ?- c\_to\_f(100,F).

F = 212.0

Yes

1. | ?- freezing(5).

Yes

1. | ?- freezing(65).

no

# **EXPERIMENT – 4**

Date: \_\_\_\_\_\_\_\_\_

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

**PREDICATES:**

/\* Define the graph with nodes and their neighbors \*/

connected(1, 7). connected(1, 8). connected(1, 3).

connected(7, 4). connected(7, 20). connected(7, 17).

connected(8, 6).

connected(3, 9). connected(3, 12).

connected(4, 42). connected(20, 28).

connected(17, 10). connected(9, 19).

neighbors(Node, Neighbors) :-

findall(Neighbor, connected(Node, Neighbor), Neighbors).

/\* Breadth First Search Traversal \*/

bfs(Graph, Start, End) :-

bfs(Graph, [Start], [Start], End).

bfs(\_, \_, VisitedNodes, End) :-

last(VisitedNodes, LastNode),

LastNode = End,

write('End node reached: '), write(LastNode), nl.

bfs(Graph, [CurrentNode|Rest], VisitedNodes, End) :-

neighbors(CurrentNode, Neighbors),

subtract(Neighbors, VisitedNodes, NewNeighbors),

append(Rest, NewNeighbors, Queue),

append(VisitedNodes, NewNeighbors, NewVisited),

bfs(Graph, Queue , NewVisited , End).

**QUERIES:**

| ?- bfs(connected,3,5).

no

| ?- bfs(connected,1,5).

no

| ?- bfs(connected,1,8).

End node reached: 98

true ?

yes

| ?- bfs(connected,3,12).

End node reached: 12

true ?

yes

| ?- bfs(connected,1,12).

End node reached: 12

true ?

# **EXPERIMENT – 5**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

water\_jug(X, Y):-X>4, Y<3, write('4L water jug overflowed.'), nl.

water\_jug(X, Y):-X<4, Y>3, write('3L water jug is overflowed.'), nl.

water\_jug(X, Y):-X>4, Y>3, write('Both water jugs overflowed.'), nl.

water\_jug(X, Y):-

(X=:=0, Y=:=0,

nl, write('4L:0 & 3L:3 (Action: Fill 3L jug.)'),

YY is 3,

water\_jug(X, YY));

(X=:=0, Y=:=0,

nl, write('4L:4 & 3L:0 (Action: Fill 4L jug.)'),

XX is 4,

water\_jug(XX,Y));

(X=:=2, Y=:=0,

nl, write('4L:2 & 3L:0 (Action: Goal State Reached....)'));

(X=:=4, Y=:=0,

nl, write('4L:1 & 3L:3 (Action: Pour water from 4L to 3L jug.)'),

XX is X-3,

YY is 3,

water\_jug(XX, YY));

(X=:=0, Y=:=3,

nl, write('4L:3 & 3:0 (Action: Pour water from 3L jug to 4L jug.)'),

XX is 3,

YY is 0,

water\_jug(XX,YY));

(X=:=1, Y=:=3,

nl, write('4L:1 & 3L:0 (Action: Empty 3L jug.)'),

YY is 0,

water\_jug(X, YY));

(X=:=3, Y=:=0,

nl,write('4L:3 & 3L:3 (Action: Fill 3L jug.)'),

YY is 3,

water\_jug(X, YY));

(X=:=3, Y=:=3,

nl, write('4L:4 & 3L:2 (Action: Pour water from 3L jug to 4L jug until 4L jug is full.)'),

XX is X+1,

YY is Y-1,

water\_jug(XX,YY));

(X=:=1, Y=:=0,

nl,write('4L:0 & 3L:1 (Action: Pour water from 4L to 3L jug.)'),

XX is Y,

YY is X,

water\_jug(XX,YY));

(X=:=0, Y=:=1,

nl, write('4L:4 & 3L:1 (Action: Fill 4L jug.)'),

XX is 4,

water\_jug(XX, Y));

(X=:=4, Y=:=1,

nl, write('4L:2 & 3L:3 (Action: Pour water from 4L jug to 3L jug until 3L jug is full.)'),

XX is X-2,

YY is Y+2,

water\_jug(XX,YY));

(X=:=2, Y=:=3,

nl, write('4L:2 & 3L:0 (Action: Empty 3L jug.)'),

YY is 0,

water\_jug(X, YY));

(X=:=4, Y=:=2,

nl, write('4L:0 & 3L:2 (Action: Empty 4L jug.)'),

XX is 0,

water\_jug(XX, Y));

(X=:=0, Y=:=2,

nl, write('4L:2 & 3L:0 (Action: Pour water from 3L jug to 4L jug.)'),

XX is Y,

YY is X,

water\_jug(XX, YY)).

**QUERIES:**

| ?- water\_jug(0,0).

4L:0 & 3L:3 (Action: Fill 3L jug.)

4L:3 & 3:0 (Action: Pour water from 3L jug to 4L jug.)

4L:3 & 3L:3 (Action: Fill 3L jug.)

4L:4 & 3L:2 (Action: Pour water from 3L jug to 4L jug until 4L jug is full.)

4L:0 & 3L:2 (Action: Empty 4L jug.)

4L:2 & 3L:0 (Action: Pour water from 3L jug to 4L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

4L:4 & 3L:0 (Action: Fill 4L jug.)

4L:1 & 3L:3 (Action: Pour water from 4L to 3L jug.)

4L:1 & 3L:0 (Action: Empty 3L jug.)

4L:0 & 3L:1 (Action: Pour water from 4L to 3L jug.)

4L:4 & 3L:1 (Action: Fill 4L jug.)

4L:2 & 3L:3 (Action: Pour water from 4L jug to 3L jug until 3L jug is full.)

4L:2 & 3L:0 (Action: Empty 3L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ?

(16 ms) yes

| ?- water\_jug(0,1).

4L:4 & 3L:1 (Action: Fill 4L jug.)

4L:2 & 3L:3 (Action: Pour water from 4L jug to 3L jug until 3L jug is full.)

4L:2 & 3L:0 (Action: Empty 3L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(0,2).

4L:2 & 3L:0 (Action: Pour water from 3L jug to 4L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(0,3).

4L:3 & 3:0 (Action: Pour water from 3L jug to 4L jug.)

4L:3 & 3L:3 (Action: Fill 3L jug.)

4L:4 & 3L:2 (Action: Pour water from 3L jug to 4L jug until 4L jug is full.)

4L:0 & 3L:2 (Action: Empty 4L jug.)

4L:2 & 3L:0 (Action: Pour water from 3L jug to 4L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

(16 ms) no

| ?- water\_jug(1,0).

4L:0 & 3L:1 (Action: Pour water from 4L to 3L jug.)

4L:4 & 3L:1 (Action: Fill 4L jug.)

4L:2 & 3L:3 (Action: Pour water from 4L jug to 3L jug until 3L jug is full.)

4L:2 & 3L:0 (Action: Empty 3L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(1,1).

no

| ?- water\_jug(1,2).

no

| ?- water\_jug(1,3).

4L:1 & 3L:0 (Action: Empty 3L jug.)

4L:0 & 3L:1 (Action: Pour water from 4L to 3L jug.)

4L:4 & 3L:1 (Action: Fill 4L jug.)

4L:2 & 3L:3 (Action: Pour water from 4L jug to 3L jug until 3L jug is full.)

4L:2 & 3L:0 (Action: Empty 3L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

(15 ms) no

| ?- water\_jug(2,0).

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(2,1).

no

| ?- water\_jug(2,2).

no

| ?- water\_jug(2,3).

4L:2 & 3L:0 (Action: Empty 3L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(3,0).

4L:3 & 3L:3 (Action: Fill 3L jug.)

4L:4 & 3L:2 (Action: Pour water from 3L jug to 4L jug until 4L jug is full.)

4L:0 & 3L:2 (Action: Empty 4L jug.)

4L:2 & 3L:0 (Action: Pour water from 3L jug to 4L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

(16 ms) no

| ?- water\_jug(3,1).

no

| ?- water\_jug(3,2).

no

| ?- water\_jug(3,3).

4L:4 & 3L:2 (Action: Pour water from 3L jug to 4L jug until 4L jug is full.)

4L:0 & 3L:2 (Action: Empty 4L jug.)

4L:2 & 3L:0 (Action: Pour water from 3L jug to 4L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(4,0).

4L:1 & 3L:3 (Action: Pour water from 4L to 3L jug.)

4L:1 & 3L:0 (Action: Empty 3L jug.)

4L:0 & 3L:1 (Action: Pour water from 4L to 3L jug.)

4L:4 & 3L:1 (Action: Fill 4L jug.)

4L:2 & 3L:3 (Action: Pour water from 4L jug to 3L jug until 3L jug is full.)

4L:2 & 3L:0 (Action: Empty 3L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(4,1).

4L:2 & 3L:3 (Action: Pour water from 4L jug to 3L jug until 3L jug is full.)

4L:2 & 3L:0 (Action: Empty 3L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

(16 ms) no

| ?- water\_jug(4,2).

4L:0 & 3L:2 (Action: Empty 4L jug.)

4L:2 & 3L:0 (Action: Pour water from 3L jug to 4L jug.)

4L:2 & 3L:0 (Action: Goal State Reached....)

true ? ;

no

| ?- water\_jug(4,3).

No

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

## AIM: Write a program to implement Tic‐Tac‐Toe game

**SOURCE CODE:**

% Win conditions

win(Board, Player) :- rowwin(Board, Player).

win(Board, Player) :- colwin(Board, Player).

win(Board, Player) :- diagwin(Board, Player).

rowwin([Player,Player,Player,\_,\_,\_,\_,\_,\_], Player).

rowwin([\_,\_,\_,Player,Player,Player,\_,\_,\_], Player).

rowwin([\_,\_,\_,\_,\_,\_,Player,Player,Player], Player).

colwin([Player,\_,\_,Player,\_,\_,Player,\_,\_], Player).

colwin([\_,Player,\_,\_,Player,\_,\_,Player,\_], Player).

colwin([\_,\_,Player,\_,\_,Player,\_,\_,Player], Player).

diagwin([Player,\_,\_,\_,Player,\_,\_,\_,Player], Player).

diagwin([\_,\_,Player,\_,Player,\_,Player,\_,\_], Player).

% Helping predicate for alternating play in a "self" game:

other(x,o).

other(o,x).

game(Board, Player) :-

win(Board, Player),

!,

write([player, Player, wins]).

game(Board, Player) :-

other(Player,Otherplayer),

move(Board,Player,Newboard),

!,

displayer(Newboard),

game(Newboard,Otherplayer).

move([-,B,C,D,E,F,G,H,I], Player, [Player,B,C,D,E,F,G,H,I]).

move([A,-,C,D,E,F,G,H,I], Player, [A,Player,C,D,E,F,G,H,I]).

move([A,B,-,D,E,F,G,H,I], Player, [A,B,Player,D,E,F,G,H,I]).

move([A,B,C,-,E,F,G,H,I], Player, [A,B,C,Player,E,F,G,H,I]).

move([A,B,C,D,-,F,G,H,I], Player, [A,B,C,D,Player,F,G,H,I]).

move([A,B,C,D,E,-,G,H,I], Player, [A,B,C,D,E,Player,G,H,I]).

move([A,B,C,D,E,F,-,H,I], Player, [A,B,C,D,E,F,Player,H,I]).

move([A,B,C,D,E,F,G,-,I], Player, [A,B,C,D,E,F,G,Player,I]).

move([A,B,C,D,E,F,G,H,-], Player, [A,B,C,D,E,F,G,H,Player]).

displayer([A,B,C,D,E,F,G,H,I]) :-

write([A,B,C]),nl,

write([D,E,F]),nl,

write([G,H,I]),nl,nl.

% Predicates to support playing a game with the user:

x\_can\_win\_in\_one(Board) :- move(Board, x, Newboard), win(Newboard, x).

% The predicate orespond generates the computer's (playing o) response

% from the current Board.

orespond(Board,Newboard) :-

move(Board, o, Newboard),

win(Newboard, o),

!.

orespond(Board,Newboard) :-

move(Board, o, Newboard),

\+ x\_can\_win\_in\_one(Newboard).

orespond(Board,Newboard) :-

move(Board, o, Newboard).

orespond(Board,Newboard) :-

\+ member(b,Board),

!,

write('Cats game!'), nl,

Newboard = Board.

% The following translates from an integer description

% of x's move to a board transformation.

xmove([-,B,C,D,E,F,G,H,I], 1, [x,B,C,D,E,F,G,H,I]).

xmove([A,-,C,D,E,F,G,H,I], 2, [A,x,C,D,E,F,G,H,I]).

xmove([A,B,-,D,E,F,G,H,I], 3, [A,B,x,D,E,F,G,H,I]).

xmove([A,B,C,-,E,F,G,H,I], 4, [A,B,C,x,E,F,G,H,I]).

xmove([A,B,C,D,-,F,G,H,I], 5, [A,B,C,D,x,F,G,H,I]).

xmove([A,B,C,D,E,-,G,H,I], 6, [A,B,C,D,E,x,G,H,I]).

xmove([A,B,C,D,E,F,-,H,I], 7, [A,B,C,D,E,F,x,H,I]).

xmove([A,B,C,D,E,F,G,-,I], 8, [A,B,C,D,E,F,G,x,I]).

xmove([A,B,C,D,E,F,G,H,-], 9, [A,B,C,D,E,F,G,H,x]).

xmove(Board, \_, Board) :- write('Illegal move.'), nl.

% The 0-place predicate playo starts a game with the user.

playo :- explain, playfrom([-,-,-,-,-,-,-,-,-]).

explain :-

write('You play X by entering integer positions followed by a period.'),

nl,

displayer([1,2,3,4,5,6,7,8,9]).

playfrom(Board) :-

win(Board, x),

write('You win!').

playfrom(Board) :-

win(Board, o),

write('I win!').

playfrom(Board) :-

read(N),

xmove(Board, N, Newboard),

displayer(Newboard),

orespond(Newboard, Newnewboard),

displayer(Newnewboard),

playfrom(Newnewboard).

**QUERIES:**

| ?- playo.

You play X by entering integer positions followed by a period.

[1,2,3]

[4,5,6]

[7,8,9]

5.

[-,-,-]

[-,x,-]

[-,-,-]

[o,-,-]

[-,x,-]

[-,-,-]

3.

[o,-,x]

[-,x,-]

[-,-,-]

[o,-,x]

[-,x,-]

[o,-,-]

9.

[o,-,x]

[-,x,-]

[o,-,x]

[o,-,x]

[o,x,-]

[o,-,x]

I win!

true ?

yes

| ?- playo.

You play X by entering integer positions followed by a period.

[1,2,3]

[4,5,6]

[7,8,9]

1.

[x,-,-]

[-,-,-]

[-,-,-]

[x,o,-]

[-,-,-]

[-,-,-]

5.

[x,o,-]

[-,x,-]

[-,-,-]

[x,o,-]

[-,x,-]

[-,-,o]

7.

[x,o,-]

[-,x,-]

[x,-,o]

[x,o,o]

[-,x,-]

[x,-,o]

4.

[x,o,o]

[x,x,-]

[x,-,o]

[x,o,o]

[x,x,o]

[x,-,o]

You win!

true ?

(16 ms) yes

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

## AIM: Write a program to remove punctuations from the given string.

**SOURCE CODE:**

remove\_punctuation(InputString, CleanedString) :-

atom\_chars(InputString, InputChars), % Convert input string to a list of characters

\+ no\_punctuation(InputChars), % Check if there is no punctuation

remove\_punctuation\_chars(InputChars, CleanedChars), % Remove punctuation characters

atom\_chars(CleanedString, CleanedChars). % Convert cleaned characters back to a string

% Define a predicate to check if there is no punctuation in a list of characters

no\_punctuation(InputChars) :-

\+ (member(Char, InputChars), punct(Char)). % Check if any character is a punctuation mark

% Define a predicate to remove punctuation characters from a list of characters

remove\_punctuation\_chars([], []). % Base case: empty list

remove\_punctuation\_chars([Char|RestChars], CleanedChars) :-

% Check if the character is a punctuation mark

punct(Char),

!, % Cut to prevent backtracking

remove\_punctuation\_chars(RestChars, CleanedChars). % Continue with the rest of the list

remove\_punctuation\_chars([Char|RestChars], [Char|CleanedRest]) :-

% If the character is not a punctuation mark, keep it in the cleaned list

remove\_punctuation\_chars(RestChars, CleanedRest).

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

punct(Char) :-

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

**QUERIES:**

| ?- remove\_punctuation('Hello, World!', Cleaned).

Cleaned = 'Hello World'

yes

| ?- remove\_punctuation('Hi, This is Lakshay Sharma. Ni:ce to; Meet-you!', Cleaned).

Cleaned = 'Hi This is Lakshay Sharma Nice to Meet-you'

yes

| ?- remove\_punctuation('Nice to meet you', Cleaned).

no

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

## AIM: Write a program to sort the sentence in alphabetical order.

**SOURCE CODE:**

sort\_sentence([], []).

% Base case: Empty list remains empty after sorting

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

find\_min(Sentence, Min),

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

sort\_sentence(Remaining, Sorted).

% Base case: Minimum of a single word list is the word itself

find\_min([Word], Word).

% Recursive case: Compare first two words

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).

% Base case: Empty list after removing element

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

% Element found at head, return remaining list

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

% Recursive case: Element not found at head, continue searching

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

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

**QUERIES:**

| ?- sort\_sentence([hello,i,am,lakshay,sharma], SortedSentence).

SortedSentence = [am,hello,i,lakshay,sharma] ?

yes

| ?- sort\_sentence([this,is,not,a,drill], SortedSentence).

SortedSentence = [a,drill,is,not,this] ?

yes

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

import random

def choose\_word():

words = ["apple", "banana", "orange", "grape", "strawberry", "pineapple"]

return random.choice(words)

def display\_word(word, guessed\_letters):

displayed\_word = ""

for letter in word:

if letter in guessed\_letters:

displayed\_word += letter

else:

displayed\_word += "\_ "

return displayed\_word

def hangman():

word = choose\_word()

guessed\_letters = []

attempts = 6

print("Welcome to Hangman!")

print("Guess the word:", display\_word(word, guessed\_letters))

while True:

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

if guess in guessed\_letters:

print("You've already guessed that letter!")

elif guess in word:

guessed\_letters.append(guess)

print("Correct guess!")

else:

attempts -= 1

print("Wrong guess! You have", attempts, "attempts left.")

if attempts == 0:

print("Game over! The word was:", word)

break

print("Word:", display\_word(word, guessed\_letters))

if "\_ " not in display\_word(word, guessed\_letters):

print("Congratulations! You've guessed the word:", word)

break

hangman()

**QUERIES:**

Welcome to Hangman!

Guess the word: \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

Guess a letter: r

Correct guess!

Word: \_ \_ r\_ \_ \_ \_ rr\_

Guess a letter: s

Correct guess!

Word: s\_ r\_ \_ \_ \_ rr\_

Guess a letter: t

Correct guess!

Word: str\_ \_ \_ \_ rr\_

Guess a letter: x

Wrong guess! You have 5 attempts left.

Word: str\_ \_ \_ \_ rr\_

Guess a letter: r

You've already guessed that letter!

Word: str\_ \_ \_ \_ rr\_

Guess a letter: e

Correct guess!

Word: str\_ \_ \_ err\_

Guess a letter: a

Correct guess!

Word: stra\_ \_ err\_

Guess a letter: w

Correct guess!

Word: straw\_ err\_

Guess a letter: b

Correct guess!

Word: strawberr\_

Guess a letter: y

Correct guess!

Word: strawberry

Congratulations! You've guessed the word: strawberry

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

% Hangman game implementation

% Words database

word(astro). word(sugar). word(casting).

word(job). word(fig). word(life).

word(maths).

% Main predicate to start the game

hangman :-

write('Welcome to Hangman!'), nl,

write('Guess the word by entering one letter at a time.'), nl,

random\_word(Word),

atom\_chars(Word, WordList),

length(WordList, WordLength),

play(WordList, WordLength, []).

% Select a random word from the database

random\_word(Word) :-

findall(W, word(W), Words),

length(Words, Len),

random(0, Len, Index),

nth0(Index, Words, Word).

% Play the game

play(WordList, WordLength, Guesses) :-

display\_word(WordList, Guesses), nl,

write('Enter your guess: '),

read\_char(Guess),

process\_guess(Guess, WordList, WordLength, Guesses, NewGuesses),

( is\_game\_over(WordList, NewGuesses) ->

write('Congratulations! You guessed the word: '),

atom\_chars(Word, WordList),

write(Word), nl

; play(WordList, WordLength, NewGuesses)

).

% Display the current state of the word with blanks and guessed letters

display\_word([], \_).

display\_word([H|T], Guesses) :-

( member(H, Guesses) ->

write(H)

; write('\_')

),

write(' '),

display\_word(T, Guesses).

% Read a single character from the user

read\_char(Char) :-

get\_char(TempChar),

get\_char(\_), % Consume newline character

Char = TempChar.

% Process the user's guess

process\_guess(Guess, WordList, WordLength, Guesses, NewGuesses) :-

( member(Guess, Guesses) ->

write('You already guessed that letter!'), nl,

NewGuesses = Guesses

; ( member(Guess, WordList) ->

write('Correct guess!'), nl,

NewGuesses = [Guess|Guesses]

; write('Incorrect guess!'), nl,

NewGuesses = [Guess|Guesses]

)

).

% Check if the game is over

is\_game\_over(WordList, Guesses) :-

subset(WordList, Guesses).

% Helper predicate to check if a list is a subset of another list

subset([], \_).

subset([H|T], List) :-

member(H, List),

subset(T, List).

**QUERIES:**

Welcome to Hangman!

Guess the word by entering one letter at a time.

\_ \_ \_ \_ \_

Enter your guess: a

Correct guess!

a \_ \_ \_ \_

Enter your guess: b

Incorrect guess!

a \_ \_ \_ \_

Enter your guess: s

Correct guess!

a s \_ \_ \_

Enter your guess: o

Correct guess!

a s \_ \_ o

Enter your guess: r

Correct guess!

a s \_ r o

Enter your guess: t

Correct guess!

Congratulations! You guessed the word: astro

(110 ms) yes

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

example = """This is a nice paragraph but unfortunately it is very hard to

write it by my own so kindly assume this short para as a passage."""

stop\_words = set(stopwords.words('english'))

word\_tokens = word\_tokenize(example)

filtered\_sentence = [w for w in word\_tokens if not w.lower() in stop\_words]

print(word\_tokens)

print(filtered\_sentence)

**QUERIES:**

['This', 'is', 'a', 'nice', 'paragraph', 'but', 'unfortunately', 'it', 'is',

'very', 'hard', 'to', 'write', 'it', 'by', 'my', 'own', 'so', 'kindly', 'assume',

'this', 'short', 'para', 'as', 'a', 'passage', '.']

['nice', 'paragraph', 'unfortunately', 'hard', 'write', 'kindly', 'assume',

'short', 'para', 'passage', '.']

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

## AIM: Write a program to sort the sentence in alphabetical order.

**SOURCE CODE:**

sort\_sentence([], []).

% Base case: Empty list remains empty after sorting

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

find\_min(Sentence, Min),

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

sort\_sentence(Remaining, Sorted).

% Base case: Minimum of a single word list is the word itself

find\_min([Word], Word).

% Recursive case: Compare first two words

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).

% Base case: Empty list after removing element

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

% Element found at head, return remaining list

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

% Recursive case: Element not found at head, continue searching

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

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

**QUERIES:**

| ?- sort\_sentence([hello,i,am,lakshay,sharma], SortedSentence).

SortedSentence = [am,hello,i,lakshay,sharma] ?

yes

| ?- sort\_sentence([this,is,not,a,drill], SortedSentence).

SortedSentence = [a,drill,is,not,this] ?

yes

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

import nltk

from nltk.stem import PorterStemmer

from nltk.tokenize import word\_tokenize

ps = PorterStemmer()

sentence = "This is an urgent text to check teh stemming process for chainging

testing hiring"

words = word\_tokenize(sentence)

for w in words:

print(w, " : ", ps.stem(w))

**QUERIES:**

This : thi

is : is

an : an

urgent : urgent

text : text

to : to

check : check

teh : teh

stemming : stem

process : process

for : for

chainging : chaing

testing : test

hiring : hire

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

import nltk

from nltk.tokenize import sent\_tokenize

txt = "This is message from Lakshay to Earth, please send more troops to moon"

tokenized = sent\_tokenize(txt)

for i in tokenized:

wordsList = nltk.word\_tokenize(i)

print(nltk.pos\_tag(wordsList))

**QUERIES:**

[('This', 'DT'), ('is', 'VBZ'), ('message', 'NN'), ('from', 'IN'), ('Lakshay', 'NNP'), ('to', 'TO'), ('Earth', 'NNP'), (',', ','), ('please', 'VB'), ('send', 'VB'), ('more', 'JJR'), ('troops', 'NNS'), ('to', 'TO'), ('moon', 'VB')]

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

import nltk

nltk.download('wordnet')

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

text = "This is the file of Lakshay Sharma. There are checkpoints in the digital file."

for i in text.split():

print(i,": ", lemmatizer.lemmatize(i))

**QUERIES:**

This : This

is : is

the : the

file : file

of : of

Lakshay : Lakshay

Sharma. : Sharma.

There : There

are : are

checkpoints : checkpoint

in : in

the : the

digital : digital

file. : file.

# **EXPERIMENT – \_**

Date: \_\_\_\_\_\_\_\_\_

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

**SOURCE CODE:**

import nltk, random

training\_data = [

("The energetic dog played fetch in the park", "animal"),

("Roses are often given as a symbol of love", "flower"),

("Python is a versatile programming language", "programming"),

("Heavy snowfall blanketed the city overnight", "weather"),

("Apples are a popular choice for making pies", "fruit"),

("Tennis requires agility and precision", "sport"),

("The wise owl hooted softly in the moonlight", "animal"),

("Tulips come in a variety of vibrant colors", "flower"),

("C++ is often used for system programming", "programming"),

("Gusty winds swept across the open plains", "weather"),

("Oranges are rich in vitamin C", "fruit"),

("Soccer is the most popular sport worldwide", "sport"),

("The majestic lion roared loudly in the jungle", "animal"),

("Daisies are known for their simple beauty", "flower"),

("JavaScript is commonly used for web development", "programming"),

("Torrential rain caused flooding in low-lying areas", "weather"),

("Grapes are often used to make wine", "fruit"),

("Golf requires precision and concentration", "sport"),

]

def extract\_features(text):

return {word: True for word in nltk.word\_tokenize(text)}

featuresets = [(extract\_features(text), label) for (text, label) in training\_data]

classifier = nltk.NaiveBayesClassifier.train(featuresets)

sample = "Sunflowers are known for their bright yellow petals"

features = extract\_features(sample)

predicted\_label = classifier.classify(features)

print("Test Sentence:", sample,"\nPredicted Label:", predicted\_label)

**QUERIES:**

Test Sentence: Sunflowers are known for their bright yellow petals

Predicted Label: flower