DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

LAB MANUAL

**Artificial Intelligence Lab (AI- 302P)**

**III Year, VI Sem.**

## Lab Incharge HoD

Department of Computer Science and Engineering

### Delhi Technical Campus, Greater Noida

|  |  |
| --- | --- |
| **Sr. No.** | **Lists** |
| 1 | Vision Mission (Institute) |
| 2 | Vision Mission (Department) |
| 3 | PO, PSO, PEO |
| 4 | University Syllabus |
| 6 | List of Experiment (Include Value Added, if any) |
| 7 | CO, Mapping of CO with PO |
| 8 | Internal evaluation Method |
| 9 | Hardware and Software Requirements |
| 10 | Experiments |
| 11 | Viva-Voice Questions |

**Vision of Institute**

To be a leading institution of higher education in India that provides an intellectually stimulating environment for academics, leadership, research excellence and social responsibility.

**Mission of Institute**

|  |  |
| --- | --- |
| **IM1:** | To provide quality state of the art for establishing remarkable standards in  Professional education. |
| **IM2:** | To collaborate with industries to achieve academic excellence, research  And entrepreneurship. |
| **IM3:** | To inculcate moral, ethical and social values in the students to make them socially responsible. |

**Vision of Department**

To be a premier department in producing efficient technocrats possessing leadership quality, research ability and a sense of social and ethical responsibility.

**Mission of Department**

**DM1:** To provide a conducive environment for better academics and research.

**DM2:** To impart leadership quality in students and the ability to work effectively in a team as per industry requirements.

**DM3:** To facilitate well-established laboratories for learning essential programming and communication skills.

**DM4:** To promote the outreach programme for producing socially responsible citizens for holistic development.

## Program Outcome (PO)

**PO1 Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

**PO2 Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

**PO3 Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**PO4 Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**PO5 Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

**PO6 The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

**PO7 Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**PO8 Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**PO9 Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**PO10 Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**PO11 Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**PO12 Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## Program Specific Outcomes (PSO)

**PSO1:** Apply basic programming techniques using hardware-software integration, web development and networking to provide solutions for the problems in the computational domain.

**PSO2:** Exhibit the attitude to work with emerging technologies in computer science engineering and open-source platforms that can solve the contemporary issues of the industries and society.

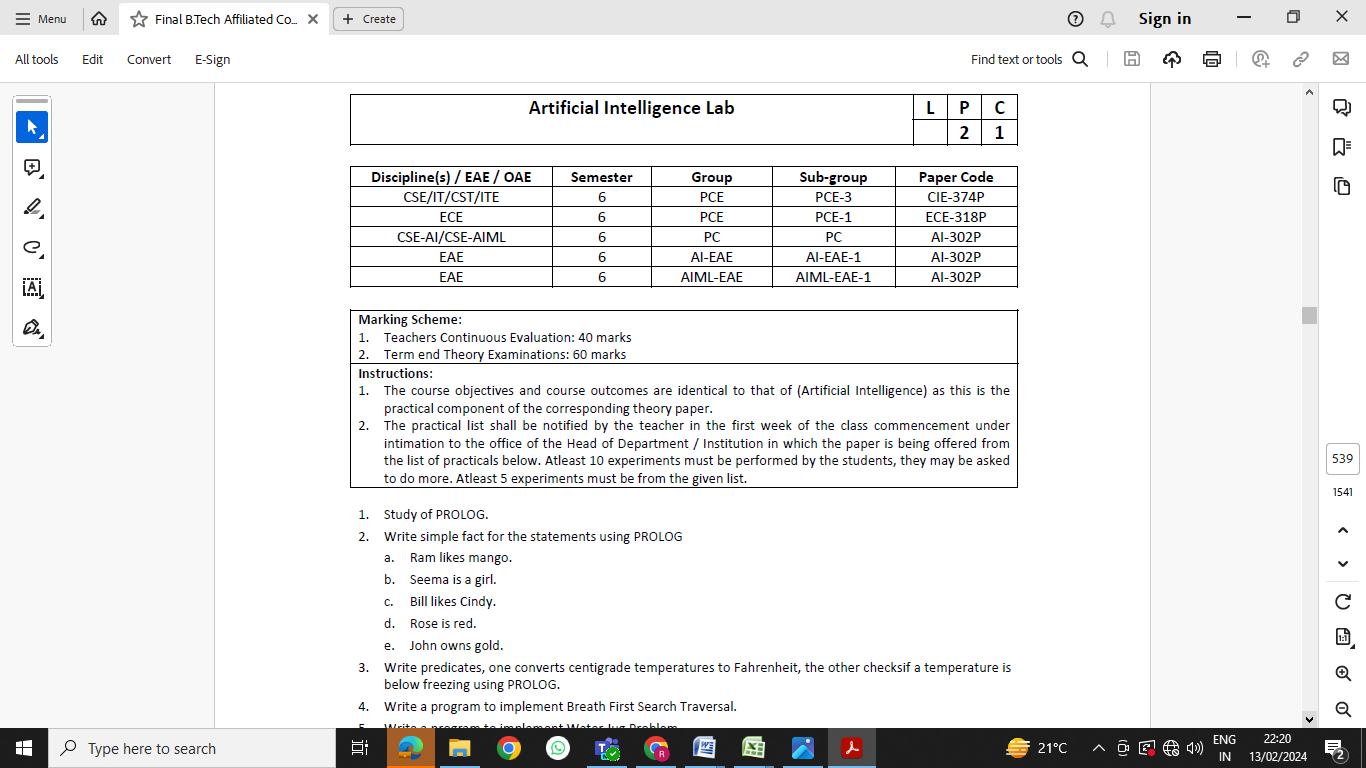
## Program Educational Objectives

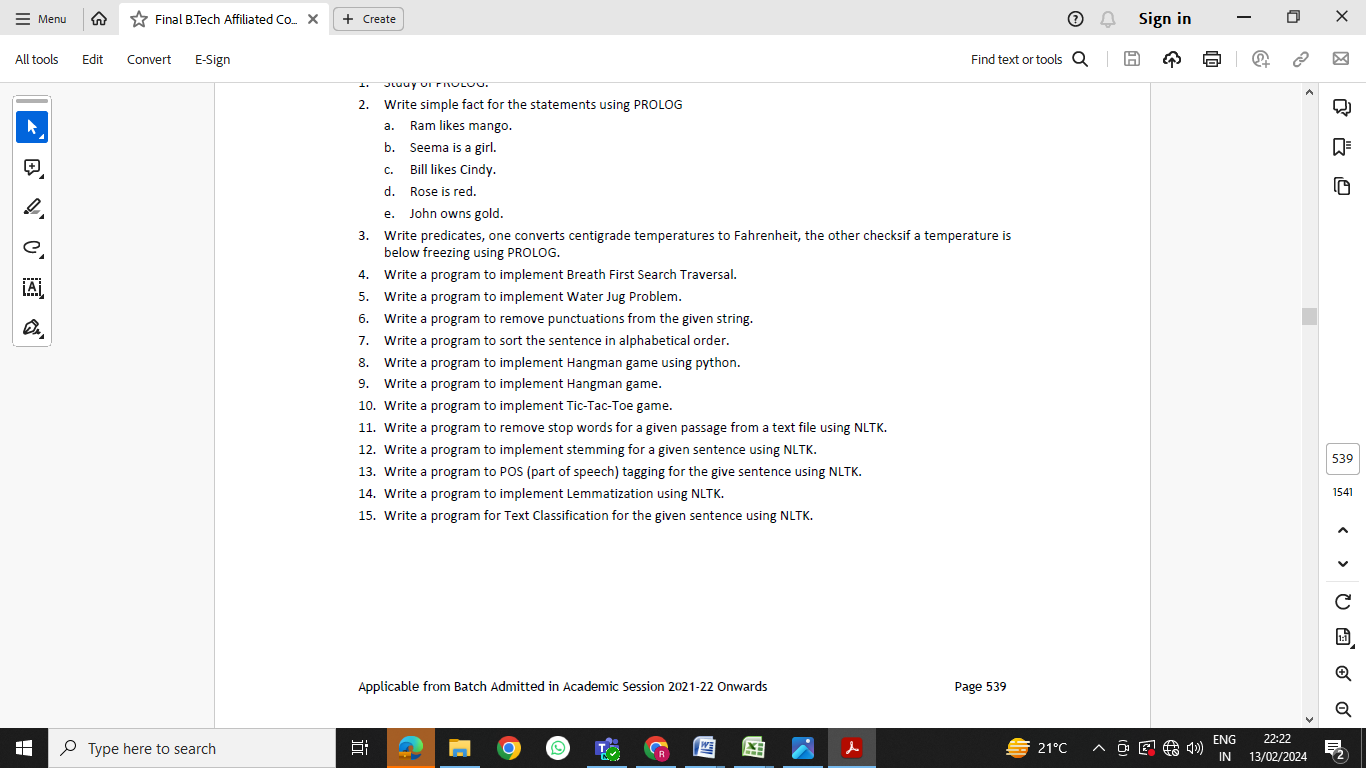
**PEO1** Excel in Computer Science and Engineering by leveraging cutting- edge tools and techniques to produce innovative solutions for complex problems.

**PEO2** Apply mathematical, scientific, and engineering principles to create software projects that meet customers’ business objectives while adhering to professional standards of quality and reliability.

**PEO3** Demonstrate effective communication skills to convey technical information to diverse audiences, collaborate with peers, and become successful leaders.

**PEO4** Analyze real-world problems in society and give ingenious ideas that positively impact the world.





# LIST OF EXPERIMENTS

|  |  |
| --- | --- |
| 1. Study of PROLOG. | **CO1** |
| 1. Write simple fact for the statements using PROLOG   Ram likes mango.  Seema is a girl.  Bill likes Cindy.  Rose is red.  John owns gold. | **CO1** |
| 1. Write predicates, one converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing using PROLOG. | **CO1** |
| 1. Write a program to implement factorial, Fibonacci of a given number. Or Write a program to implement Breath First Search Traversal. | **CO2** |
| 1. Write a program to implement Water Jug Problem. | **CO2** |
| 1. Write a program to remove punctuations from the given string. | **CO2** |
| 1. Write a program to sort the sentence in alphabetical order. | **CO2** |
| 1. Write a program to implement Hangman game using python. | **CO3** |
| 1. Write a program to implement Hangman game. | **CO3** |
| 1. Write a program to implement Tic-Tac-Toe game. | **CO3** |
| 1. Write a program to remove stop words for a given passage from a text file using NLTK. | **CO4** |
| 1. Write a program to implement stemming for a given sentence using NLTK. | **CO4** |
| 1. Write a program to POS (part of speech) tagging for the give sentence using NLTK. | **CO4** |
| 1. Write a program to implement Lemmatization using NLTK. | **CO5** |
| 1. Write a program for Text Classification for the given sentence using NLTK. | **CO5** |

## COURSE OUTCOMES

**CO1** Demonstrate the features of the Prolog Programming language, including basic syntax, selection and search strategies of PROLOG.

**CO2** Ability to apply and analyze various problem solving techniques in Artificial intelligence.

**CO3**  Demonstrate the understanding of different algorithms and the techniques, also analyze the results.

**CO4** Ability to implement concepts of natural language processing using NLTK and interpret the results.

**CO5** Understand the concept of Backtracking to find alternative solutions when searching queries.

## MAPPING OF CO WITH PO

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** | **PSO1** | **PSO2** |
| **CO1** | 3 | - | - | - | - | - | - | - | 2 | - | - | 2 | 3 | - |
| **CO2** | 2 | 2 | - | - | 3 | - | - | - | 2 | - | - | 2 | 2 | - |
| **CO3** | 2 | 2 | 3 | - | 2 | - | - | - | 2 | - | - | 2 | 2 | - |
| **CO4** | 2 | - | 3 | - | 3 | - | - | - | 2 | - | - | 2 | 3 | - |
| **CO5** | 3 | 2 | - | - | - | - | - | - | 2 | - | - | 3 | 2 | - |
| **Avg.** | 2.4 | 2 | 3 | - | 2.6 | - | - | - | 2 | - | - | 2.2 | 2.4 | - |

## HARDWARE AND SOFTWARE REQUIREMENTS

* Hardware requirements:
  + Intel CORETM i5
  + 19“ color monitor
  + Mouse
  + Keyboard
  + 256GB SSD
  + 8GB RAM
* Software Requirement:

SWI-Prolog 9.3.1

**Experiment No. 1**

**Aim: Study of PROLOG.**

**Solution:**

PROLOG, which stands for "Programming in Logic," is a high-level programming language designed for symbolic reasoning and manipulation. It is particularly well-suited for tasks related to artificial intelligence, logic programming, and rule-based systems. PROLOG is declarative, meaning that you specify the desired outcome rather than the step-by-step process to achieve it.

**History:** PROLOG was developed in the early 1970s by Alain Colmerauer and a team of researchers in Marseille, France. The language gained popularity in the field of artificial intelligence and knowledge representation due to its natural support for formal logic and rule-based reasoning. Since its inception, PROLOG has been used in a variety of applications, including expert systems, natural language processing, and database querying.

**Usage:** PROLOG finds applications in various domains, such as:

1. **Artificial Intelligence (AI):** PROLOG is widely used in AI research and applications for tasks such as expert systems, automated reasoning, and knowledge representation.
2. **Natural Language Processing (NLP):** Its logical and pattern-matching capabilities make PROLOG suitable for processing and understanding natural language.
3. **Database Systems:** PROLOG is used for querying and manipulating knowledge in databases through its powerful pattern-matching features.
4. **Symbolic Mathematics:** PROLOG is employed in symbolic mathematics and theorem proving, where logical inference is crucial.

**Features:** Some key features of PROLOG include:

1. **Declarative Syntax:** PROLOG programs describe relationships and rules rather than specifying a sequence of steps to achieve a goal.
2. **Logical Inference:** PROLOG is based on formal logic, enabling users to express and query relationships using rules of deduction and inference.
3. **Pattern Matching:** The language excels at pattern matching, allowing for powerful and flexible data manipulation.
4. **Backtracking:** PROLOG's unique backtracking mechanism allows the system to explore alternative solutions when searching for answers to queries.

**Basics of Programming in PROLOG:**

* **Facts and Rules:** Programs consist of facts (statements about relationships) and rules (logical conditions or implications).
* **Predicates:** Predicates define relationships between terms. They are the building blocks of PROLOG programs.
* **Variables:** Variables are placeholders for unknown values. They allow for more general and flexible rule definitions.
* **Queries:** Users interact with the system by posing queries, asking questions, or seeking solutions to problems.
* **Recursion:** PROLOG supports recursion, allowing functions or rules to call themselves, which is a fundamental concept for solving complex problems.

In summary, PROLOG's unique combination of logic-based programming, pattern matching, and inference mechanisms makes it a powerful language for a wide range of applications in the field of artificial intelligence and symbolic reasoning.

**Experiment No. 2**

**Aim:** Write simple fact for the following:

1. Ram likes mango.
2. Seema is a girl.
3. Bill likes Cindy.
4. Rose is red.
5. John owns gold.

**Solution:**In Prolog, the format for expressing facts is generally **predicate(term1, term2, ..., termN)**, where **predicate** is the name of the relation, and **term1, term2, ..., termN** are the terms involved in the relation. Each term can be a variable, an atom, or a compound term.

**CODE:**

Create a file ‘facts.pl’ and write the following facts in it:

**likes (ram, mango).**

**is\_a(seema, girl).**

**likes(bill, cindy).**

**is\_color(rose, red).**

**owns(john, gold).**

Now open SWI-Prolog and consult ‘facts.pl’ and type in your queries.

**OUTPUT:**

**A screenshot of a computer code

Description automatically generated**

##### Viva Questions:

1. First create a source file for the genealogical logic base application. Start by adding a few members of your family tree. It is important to be accurate, since we will be exploring family relationships. Your own knowledge of who your relatives are will verify the correctness of your Prolog programs.
2. Enter a two-argument predicate that records the parent-child relationship. One argument represents the parent, and the other the child. It doesn't matter in which order you enter the arguments, as long as you are consistent. Often Prolog programmers adopt the convention that parent(A,B) is interpreted "A is the parent of B".
3. Create a source file for the customer order entry program. We will begin it with three record types (predicates). The first is customer/3 where the three arguments are

arg1

arg2 arg3

Customer name City

Credit rating (aaa, bbb, etc)

1. Next add clauses that define the items that are for sale. It should also have three arguments

arg1

Item identification number

arg2 arg3

Item name

The reorder point for inventory (when at or below this level, reorder)

1. Next add an inventory record for each item. It has two arguments.

arg1 arg2

Item identification number (same as in the item record) Amount in stock

**Experiment No. 3**

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

**Solution:** In Prolog, the format for expressing facts is generally **predicate(term1, term2, ..., termN)**, where **predicate** is the name of the relation, and **term1, term2, ..., termN** are the terms involved in the relation. Each term can be a variable, an atom, or a compound term.

**Program:**

Create a file ‘temperature.pl’ and write the following predicates in it:

**% Predicate to convert Centigrade to Fahrenheit  
centigrade\_to\_fahrenheit(C, F) :-  
 F is (C \* 9/5) + 32.**

**% Predicate to check if a temperature is below freezing  
below\_freezing(Temperature) :-  
 Temperature < 0.**

Now open SWI-Prolog and consult ‘temperature.pl’ and type in your queries.

**OUTPUT:   
A screenshot of a computer

Description automatically generated**

##### Viva Questions:

1. Write a predicate valid\_order/3 that checks whether a customer order is valid. The arguments should be customer, item, and quantity. The predicate should succeed only if the customer is a valid customer with a good credit rating, the item is in stock, and the quantity ordered is less than the quantity in stock.
2. Write a reorder/1 predicate which checks inventory levels in the inventory record against the reorder quantity in the item record. It should write a message indicating whether or not it's time to reorder.

**Experiment No. 4**

**Aim:**Write a program to implement factorial and Fibonacci of a given number.

.

**Solution:**

Factorial:

factorial (0,1).

factorial (N,F) :-

N>0,

N1 is N-1,

factorial (N1,F1), F is N \* F1.

Output:

Goal:

?- factorial(4,X).

X=24

Fibonacci:

fib (0, 0).

fib (X, Y) :- X > 0, fib (X, Y, \_).

fib (1, 1, 0).

fib (X, Y1, Y2) :-

X > 1,

X1 is X - 1,

fib (X1, Y2, Y3),

Y1 is Y2 + Y3.

Output:

Goal:

?-fib (10,X).

X=55

**Viva Questions:**

1. Describe the difference between iterative and recursive approaches for computing Fibonacci numbers.
2. Can you discuss the time complexity of your factorial and Fibonacci implementations?
3. Can you discuss the time complexity of your factorial and Fibonacci implementations?

**Experiment 5**

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

**Solution:** The Water Jug Problem is a classic problem in artificial intelligence and problem-solving that involves determining a sequence of steps to measure a specific amount of water using two jugs of different capacities. The problem statement typically includes the following elements:

1. **Jug Capacities:** There are two jugs with different capacities. Let's denote them as Jug1 and Jug2.
2. **Initial State:** Both jugs start empty (0 units of water).
3. **Goal State:** The objective is to measure a certain amount of water, which is the goal. The goal state specifies the desired amount of water to be present in one of the jugs.
4. **Operations:** The only allowed operations involve pouring water between the jugs. There are two types of pouring operations:
   * Pour water from one jug to another (e.g., pour water from Jug1 to Jug2, or vice versa).
   * Empty a jug (pour water out of a jug).
5. **Constraints:** The pouring operations must respect the capacities of the jugs. For example, pouring water into a jug cannot exceed its capacity, and emptying a jug cannot go below 0 units.

The challenge is to find a sequence of pouring operations that, starting from the initial state, leads to the goal state, satisfying the conditions and constraints of the problem.

The Water Jug Problem is a classic example used in teaching and learning about problem-solving strategies, algorithm design, and logic programming. It is often used to illustrate concepts such as state space exploration, breadth-first search, and depth-first search.

**CODE:**

Create a file ‘water.pl’ and write the following predicates in it:

**member(X,[X|\_]).**

**member(X,[Y|Z]):-member(X,Z).**

**move(X,Y,\_):-X=:=2,Y=:=0,write('done'),!.**

**move(X,Y,Z):-X<4,\+member((4,Y),Z),write("fill 4 jug"),nl,move(4,Y,[(4,Y)|Z]).**

**move(X,Y,Z):-Y<3,\+member((X,3),Z),write("fill 3 jug"),nl,move(X,3,[(X,3)|z]).**

**move(X,Y,Z):-X>0,\+member((0,Y),Z),write("pour 4 jug"),nl,move(0,Y,[(0,Y)|Z]).**

**move(X,Y,Z):-Y>0,\+member((X,0),Z),write("pour 3 jug"),nl,move(X,0,[(X,0)|Z]).**

**move(X,Y,Z):-P is X+Y,P>=4,Y>0,K is 4-X,M is Y-K,\+member((4,M),Z),write("pour from 3jug to 4jug"),nl,move(4,M,[(4,M)|Z]).**

**move(X,Y,Z):-P is X+Y,P>=3,X>0,K is 3-Y,M is X-K,\+member((M,3),Z),write("pour from 4jug to 3jug"),nl,move(M,3,[(M,3)|Z]).**

**move(X,Y,Z):-K is X+Y,K<4,Y>0,\+member((K,0),Z),write("pour from 3jug to 4jug"),nl,move(K,0,[(K,0)|Z]).**

**move(X,Y,Z):-K is X+Y,K<3,X>0,\+member((0,K),Z),write("pour from 4jug to 3jug"),nl,move(0,K,[(0,K)|Z]).**

Now open SWI-Prolog and consult ‘water.pl’ and type in your queries.

**OUTPUT:   
A screenshot of a computer

Description automatically generated**

**Viva Question:**

1-How do you represent the state of the water jugs in your program?

2-Discuss any specific strategies or heuristics you used to optimize the solution.

3-Can you adapt your program to solve variations of the water jug problem, such as the infinite supply variant?

4-Discuss any practical applications or scenarios where the water jug problem or its variations might be relevant.

**Experiment 6**

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

**Solution:** In Prolog, lists are a fundamental data structure, and they are constructed using a head and tail system. A list can be viewed as a sequence of elements where the first element is called the "head," and the remaining elements form the "tail," which is itself a list.

Here's how the head and tail system works:

* **Head:** The first element of a list.
* **Tail:** The rest of the list, which is itself a list.

In Prolog syntax, a list is represented as **[Head | Tail]**, where **Head** is the first element, and **Tail** is the rest of the list. The **Tail** can be an empty list (**[]**), a single element, or another list.

**CODE:**

Create a file ‘punc.pl’ and write the following predicates in it:

**remove\_punctuations(Sentence, Output) :-**

**atom\_codes(Sentence, Codes),**

**remove\_punctuations\_codes(Codes, OpCodes),**

**atom\_codes(Output, OpCodes).**

**remove\_punctuations\_codes([], []).**

**remove\_punctuations\_codes([Code | Codes], [Code | OpCodes]) :-**

**not(is\_punctuation(Code)),**

**remove\_punctuations\_codes(Codes, OpCodes).**

**remove\_punctuations\_codes([\_| Codes], OpCodes) :-**

**remove\_punctuations\_codes(Codes, OpCodes).**

**is\_punctuation(Code) :-**

**char\_code(Char, Code),**

**memberchk(Char,['.', ',', ';', ':', '?', '!', '(', ')', '"', "'"]).**

Now open SWI-Prolog and consult ‘punc.pl’ and type in your queries.

**OUTPUT: A screenshot of a computer error

Description automatically generated  
Viva Question:**

1-Can you discuss any specific data structures or methods you used to efficiently remove punctuations?

2-How does your program handle edge cases, such as empty strings or strings containing only punctuation characters?

3-What happens if the input string contains special characters or non-ASCII characters?

4-How does the size of the input string impact the runtime of your program?

**Experiment 7**

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

**Solution:** The **sort/2** predicate in Prolog is used to sort a list in ascending order, removing duplicate elements, and arranging the remaining elements in ascending order.  
**Usage:** **sort(+List, -Sorted)  
How it Works:sort/2** compares terms using the standard order of terms in Prolog.

* Duplicate elements are removed.
* The remaining elements are arranged in ascending order.

**atomic\_list\_concat/3 Predicate:**

The **atomic\_list\_concat/3** predicate in Prolog concatenates a list of atomic elements into a single atomic element, separated by a specified separator.

**Usage:** **atomic\_list\_concat(+List, +Separator, -Atomic)**

**How it Works:**

* **atomic\_list\_concat/3** takes a list of atomic elements and a separator.
* It concatenates the elements in the list using the separator.
* The result is a single atomic element.

In Prolog, the sort/2 predicate typically works according to the standard order of terms, which is based on ASCII codes. Prolog uses the lexicographic order of terms, and for atoms (including characters), this order is determined by their ASCII codes.

1. Atoms and Characters: Prolog atoms, including individual characters, are ordered based on their ASCII codes. The ASCII code for a character determines its position in the lexicographic order.
2. Strings: In Prolog, strings are often represented as lists of characters or atoms. When you use sort/2 on a list of atoms or characters, it considers their ASCII codes for sorting.
3. Numbers: For numeric terms, Prolog uses the standard order of numbers. It compares numbers based on their magnitude.

**CODE:**

Create a file ‘sorting.pl’ and write the following predicates in it:

**split\_sentence(Sentence, Words) :-**

**atomic\_list\_concat(Words, ' ', Sentence).**

**sort\_words(Words, SortedWords) :-**

**sort(Words, SortedWords).**

**concatenate\_words(Words, Sentence) :-**

**atomic\_list\_concat(Words, ' ', Sentence).**

**sort\_sentence(UnsortedSentence, SortedSentence) :-**

**split\_sentence(UnsortedSentence, Words),**

**sort\_words(Words, SortedWords),**

**concatenate\_words(SortedWords, SortedSentence).**

Now open SWI-Prolog and consult ‘sorting.pl’ and type in your queries.

**OUTPUT: A screenshot of a computer

Description automatically generated**

**Viva Questions:**

1-Describe the algorithm you used to sort a sentence in alphabetical order.

2-ow do you handle punctuation marks, capitalization, and special characters during sorting?

3-Explain any considerations you took into account when designing your sorting algorithm.

4-Discuss any potential applications or scenarios where sorting sentences in alphabetical order might be useful.

**Experiment No. 8**

**Aim:** Write a program to implement Hangman game using python.

**Solution:** Hangman is a well-known game where one player thinks of a word and the other player tries to guess it by suggesting letters with a fixed number of guesses. One player (in this case the program) presents a sequence of dashes representing each letter of the word, along with a character as a starting point. The other player (or user) guesses the characters present in the word, one by one. If the letter they guess is in the target word, the program fills the dash with the character. Otherwise, the player loses one chance. In a total of seven chances, if the user guesses the word correctly, they win the game. If not, they lose.

Following is a Python script of the classic game “Hangman”. A row of dashes represents the word to guess. If the player guesses a letter in the word, the script writes it in all its correct positions.  The player has 10 turns to guess the word. You can easily customize the game by changing the variables. Make sure that you understand what each line does. All the python statements have been explained using python comments for this reason.

**Program:**

#importing the time module

import time

#welcoming the user

name = input("What is your name? ")

print ("Hello, " + name, "Time to play hangman!")

#wait for 1 second

time.sleep(1) print ("Start guessing...") time.sleep(0.5)

#here we set the secret. You can select any word to play with.

word = ("secret")

#creates an variable with an empty value

guesses = ''

#determine the number of turns

turns = 10

# Create a while loop

#check if the turns are more than zero

while turns > 0:

# make a counter that starts with zero

failed = 0

# for every character in secret word

for char in word:

# see if the character is in the players guess

if char in guesses:

# print then out the character

print (char,end=""),

else:

# if not found, print a dash

print ("\_",end=""),

# and increase the failed counter with one

failed += 1

# if failed is equal to zero

# print You Won

if failed == 0:

print ("You won")

# exit the script break

# ask the user go guess a character

guess = input("guess a character:")

# set the players guess to guesses

guesses += guess

# if the guess is not found in the secret word

if guess not in word:

# turns counter decreases with 1 (now 9)

turns -= 1

# print wrong

print ("Wrong")

# how many turns are left

print ("You have", + turns, 'more guesses' )

# if the turns are equal to zero

if turns == 0:

# print "You Lose"

print ("You Lose" )

Output:

What is your name? Aditya

Hello, Aditya Time to play hangman! Start guessing... \_\_\_\_\_\_guess a character:s

s\_\_\_\_\_guess a character:e

se\_\_e\_guess a character:c

sec\_e\_guess a character:r

secre\_guess a character:e

secre\_guess a character:t

secret You won

**ViVa Question :**

1-Explain how you determine whether a guessed letter is correct or incorrect?

2-How do you handle cases where the player inputs a non-alphabetic character or multiple characters?

3-How would you extend your Hangman game program to include features such as hints, multiplayer mode, or a graphical user interface (GUI

4-Explain how you would implement difficulty levels in the game, such as easy, medium, and hard?

**Experiment No. 9**

**Aim:** Write a program to implement Hangman game.

**Solution:**Hangman is a well-known game where one player thinks of a word and the other player tries to guess it by suggesting letters with a fixed number of guesses. One player (in this case the program) presents a sequence of dashes representing each letter of the word, along with a character as a starting point. The other player (or user) guesses the characters present in the word, one by one. If the letter they guess is in the target word, the program fills the dash with the character. Otherwise, the player loses one chance. In a total of seven chances, if the user guesses the word correctly, they win the game. If not, they lose.

**Program:**

% This top-level predicate runs the game. It prints a

% welcome message, picks a phrase, and calls getGuess.

% Ans = Answer

% AnsList = AnswerList

hangman(CountFailed):-

getPhrase(Ans),

!,

write('Welcome to hangman.'),

nl,

name(Ans,AnsList),

makeBlanks(AnsList, BlankList),

getGuess(AnsList,BlankList, CountFailed).

% Randomly returns a phrase from the list of possibilities.

getPhrase(Ans):-

phrases(L),

length(L, X),

R is random(X),

N is R+1,

getNth(L, N, Ans).

% Possible phrases to guess.

phrases(['a\_picture\_is\_worth\_a\_thousand\_words','one\_for\_the\_money','dead\_or\_alive','computer\_science']).

% Asks the user for a letter guess. Starts by writing the

% current "display phrase" with blanks, then asks for a guess and

% calls process on the guess.

getGuess(AnsList, BlankList, CountFailed):-

name(BlankName, BlankList),

write(BlankName),

nl,

write('Enter your guess, followed by a period and return.'),

nl,

read(Guess),

!,

name(Guess, [GuessName]),

processGuess(AnsList,BlankList,GuessName, CountFailed).

% Process guess takes a list of codes representing the answer, a list of codes representing the current

% "display phrase" with blanks in it, and the code of the letter that was just guessed. If the guess

% was right, call substitute to put the letter in the display phrase and check for a win. Otherwise, just

% get another guess from the user.

processGuess(AnsList,BlankList,GuessName, CountFailed):-

member(GuessName,AnsList),

!,

write('Correct!'),

nl,

substitute(AnsList, BlankList, GuessName, NewBlanks),

checkWin(AnsList,NewBlanks, CountFailed).

processGuess(AnsList, BlankList, \_, CountFailed) :-

( CountFailed == 5

-> format('Sorry, game over. You didn\'t guess (~s)~n', [AnsList])

; write('Nope!'),

nl,

CountFailed1 is CountFailed + 1,

getGuess(AnsList, BlankList, CountFailed1)

).

% Check to see if the phrase is guessed. If so, write 'You win' and if not, go back and get another guess.

checkWin(AnsList, BlankList, CountFailed):-

name(Ans, AnsList),

name(BlankName, BlankList),

BlankName = Ans,

!,

write('You win!').

checkWin(AnsList, BlankList, CountFailed):-

getGuess(AnsList, BlankList, CountFailed).

% getNth(L,N,E) should be true when E is the Nth element of the list L. N will always

% be at least 1.

getNth([H|T],1,H).

getNth([H|T],N,E):-

N1 is N-1,

getNth(T,N1,E1),

E=E1.

% makeBlanks(AnsList, BlankList) should take an answer phrase, which is a list

% of character codes that represent the answer phrase, and return a list

% where all codes but the '\_' turn into the code for '\*'. The underscores

% need to remain to show where the words start and end. Please note that

% both input and output lists for this predicate are lists of character codes.

% You can test your code with a query like this:

% testMakeBlanks:- name('csc\_is\_awesome', List), makeBlanks(List, BlankList), name(Towrite, BlankList), write(Towrite).

makeBlanks(AnsCodes, BlankCodes) :-

maplist(answer\_blank, AnsCodes, BlankCodes).

answer\_blank(Ans, Blank) :-

Ans == 0'\_ -> Blank = Ans ; Blank = 0'\* .

% substitute(AnsList, BlankList, GuessName, NewBlanks) Takes character code lists AnsList and BlankList,

% and GuessName, which is the character code for the guessed letter. The NewBlanks should again be a

% character code list, which puts all the guesses into the display word and keeps the \*'s and \_'s otherwise.

% For example, if the answer is 'csc\_is\_awesome' and the display is 'c\*c\_\*\*\_\*\*\*\*\*\*\*' and the guess is 's', the

% new display should be 'csc\_\*s\_\*\*\*s\*\*\*'.

% You can test your predicate with a query like this:

% testSubstitute:- name('csc\_is\_awesome', AnsList), name('c\*c\_\*\*\_\*\*\*\*\*\*\*', BlankList), name('s',[GuessName]), substitute(AnsList, BlankList, GuessName, NewBlanks),

% name(Towrite, NewBlanks), write(Towrite).

% Also, since the predicate doesn't deal directly with character codes, this should also work:

% substitute(['c','s','c'],['c','\*','c'],'s',L). L should be ['c','s','c'].

substitute(AnsCodes, BlankCodes, GuessName, NewBlanks) :-

maplist(place\_guess(GuessName), AnsCodes, BlankCodes, NewBlanks).

place\_guess(Guess, Ans, Blank, Display) :-

Guess == Ans -> Display = Ans ; Display = Blank.

**Viva Question:**

1-What test cases did you use to validate your Hangman game program?

2-How do you ensure that the game behaves as expected under different scenarios and inputs?

3-How do you ensure that the selected word is appropriate for the chosen difficulty level?

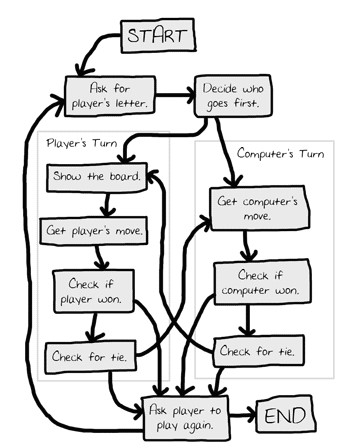
4-What input validation mechanisms are in place to handle user inputs?

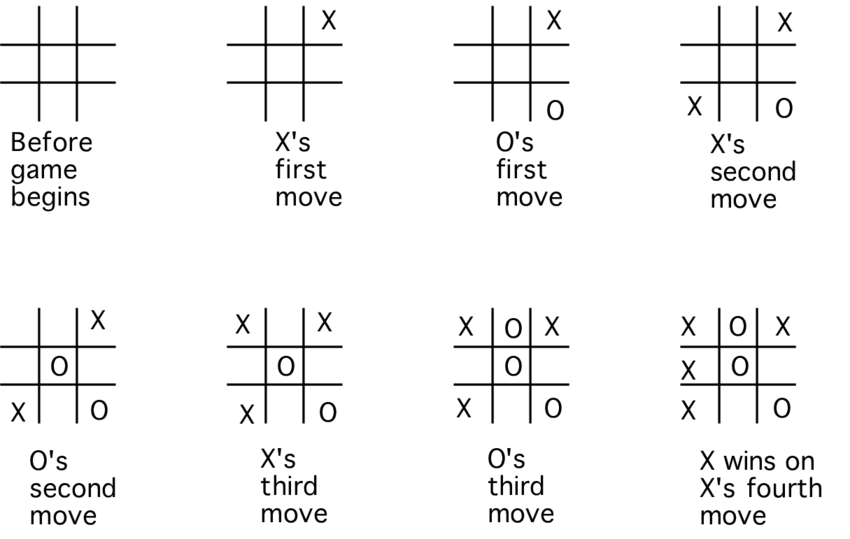
**Experiment No. 10**

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

**Solution:** In the Tic Tac Toe computer program the player chooses if they want to be X or O. Who takes the first turn is randomly chosen. Then the player and computer take turns making moves. The boxes on the left side of the flowchart are what happens during the player’s turn. The right side shows what happens on the computer's turn. After the player or computer makes a move, the program checks if they won or caused a tie, and then the game switches turns. After the game is over, the program asks the player if they want to play again.

**Program:**

****

****

% To play a game with the computer, type

% playo.

% To watch the computer play a game with itself, type

% selfgame.

%

% original at https://courses.cs.washington.edu/courses/cse341/03sp/slides/PrologEx/tictactoe.pl.txt

% Predicates that define the winning conditions:

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

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

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

rowwin(Board, Player) :- Board = [Player,Player,Player,\_,\_,\_,\_,\_,\_].

rowwin(Board, Player) :- Board = [\_,\_,\_,Player,Player,Player,\_,\_,\_].

rowwin(Board, Player) :- Board = [\_,\_,\_,\_,\_,\_,Player,Player,Player].

colwin(Board, Player) :- Board = [Player,\_,\_,Player,\_,\_,Player,\_,\_].

colwin(Board, Player) :- Board = [\_,Player,\_,\_,Player,\_,\_,Player,\_].

colwin(Board, Player) :- Board = [\_,\_,Player,\_,\_,Player,\_,\_,Player].

diagwin(Board, Player) :- Board = [Player,\_,\_,\_,Player,\_,\_,\_,Player].

diagwin(Board, Player) :- Board = [\_,\_,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),

!,

display(Newboard),

game(Newboard,Otherplayer).

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

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

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

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

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

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

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

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

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

display([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.

selfgame :- game([b,b,b,b,b,b,b,b,b],x).

% 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) reponse

% from the current Board.

orespond(Board,Newboard) :-

move(Board, o, Newboard),

win(Newboard, o),

!.

orespond(Board,Newboard) :-

move(Board, o, Newboard),

not(x\_can\_win\_in\_one(Newboard)).

orespond(Board,Newboard) :-

move(Board, o, Newboard).

orespond(Board,Newboard) :-

not(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,B,C,D,E,F,G,H,I], 1, [x,B,C,D,E,F,G,H,I]).

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

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

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

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

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

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

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

xmove([A,B,C,D,E,F,G,H,b], 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([b,b,b,b,b,b,b,b,b]).

explain :-

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

nl,

display([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),

display(Newboard),

orespond(Newboard, Newnewboard),

display(Newnewboard),

playfrom(Newnewboard).

**Viva Question:**

1-Can you discuss the data structures used to store the state of the game board?

2-What are the winning conditions in Tic-Tac-Toe?

3-How do you handle a draw scenario in the game?

5-Can you discuss any potential improvements or enhancements you would make to the game's user experience and functionality?

**Viva Question Bank**

1. **What is Prolog?**

Prolog stands for "Programming in Logic" and is a logic programming language commonly used in artificial intelligence and computational linguistics.

1. **How does Prolog represent knowledge?**

Prolog represents knowledge using facts and rules. Facts represent assertions about the world, while rules define relationships and logical implications.

1. **What is a predicate in Prolog?**

A predicate in Prolog is a logical statement or rule that defines a relationship between objects or terms. Predicates are composed of atoms, variables, and logical operators.

1. **What is the difference between facts and rules in Prolog?**

Facts in Prolog are assertions about the world and are stated as simple statements, whereas rules define logical implications and relationships between terms based on certain conditions.

1. **How does Prolog perform logical inference?**

Prolog performs logical inference by matching queries with facts and rules in its knowledge base using a process called backtracking.

1. **What is unification in Prolog?**

Unification in Prolog is the process of finding substitutions for variables in predicates to make them match and satisfy the query.

1. **How does Prolog handle recursive rules?**

Prolog handles recursive rules by using backtracking and iterative deepening to explore the search space until a solution is found.

1. **What is the cut operator in Prolog?**

The cut operator (!) in Prolog is used to prune the search space and prevent backtracking beyond a certain point, increasing efficiency but potentially limiting completeness.

1. **Explain the difference between breadth-first search and depth-first search in Prolog.**

Breadth-first search explores all possible solutions at each level of the search tree before moving to the next level, while depth-first search explores one branch of the search tree fully before backtracking.

1. **How does Prolog handle negation?**

Prolog handles negation using negation as failure, where it assumes a query is false if it cannot find any evidence to prove it true.

1. **What is backtracking in Prolog?**

Backtracking in Prolog is the process of exploring alternative solutions by undoing choices made during the execution of a program.

1. **How does Prolog handle lists and recursion?**

Prolog treats lists as recursive structures, with the empty list denoted as [ ], and recursively processes list elements using pattern matching and recursion.

1. **Explain the difference between deterministic and non-deterministic predicates in Prolog.**

Deterministic predicates in Prolog have only one possible solution, while non-deterministic predicates may have multiple solutions, requiring backtracking to find all possible solutions.

1. **What is the cut operator used for in Prolog, and what are its implications?**

The cut operator (!) in Prolog is used to prune the search space and prevent backtracking beyond a certain point, potentially improving efficiency but may also lead to unintended consequences if used incorrectly.

1. **How does Prolog handle arithmetic operations?**

Prolog provides built-in predicates for arithmetic operations such as addition, subtraction, multiplication, division, and comparison.

1. **Explain the concept of logical variables in Prolog.**

Logical variables in Prolog are placeholders that can be unified with terms to find solutions to queries. They are denoted by uppercase letters or underscores.

1. **What is the role of the cut operator (!) in Prolog, and how does it affect program execution?**

The cut operator (!) in Prolog is used to prune the search space by committing to a particular choice and preventing backtracking beyond that point, which can improve efficiency but may also lead to unintended consequences.

1. **How does Prolog handle the resolution of predicates?**

Prolog resolves predicates by matching queries with facts and rules in its knowledge base using a process called unification and backtracking.

1. **How does Prolog handle input and output?**

Prolog provides built-in predicates such as write, read, and consult for input and output operations, allowing users to interact with the Prolog interpreter.

1. **Explain the difference between Prolog's "is" and "=" operators.**

The "is" operator in Prolog is used for arithmetic evaluation, while the "=" operator is used for unification and comparison of terms.

1. **How does Prolog represent and manipulate trees?**

Prolog represents trees using recursive data structures, with nodes represented as terms and branches represented as lists of subtrees.

1. **What are some advantages of using Prolog for artificial intelligence applications?**

Prolog's declarative nature, pattern matching capabilities, built-in backtracking, and support for logical inference make it well-suited for symbolic reasoning and problem-solving tasks in artificial intelligence.

1. **Can Prolog handle complex data structures such as graphs?**

Yes, Prolog can handle complex data structures such as graphs by representing nodes and edges as facts and using recursive predicates to traverse and manipulate the graph.