**EXPERIMENT - 1**

AIM ::

Study of Prolog.

Theory ::

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

## Key Concepts

### Facts and Rules -

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

parent(alice, bob).

parent(alice, carol).

parent(bob, david).

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

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

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

### Unification

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

### Backtracking

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

### Declarative Programming

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

## Applications of Prolog -

### Artificial Intelligence

* Expert systems
* Natural language processing
* Machine learning

### Symbolic Computation

* Theorem proving
* Constraint satisfaction problems

### Other

* Logic programming
* Database systems
* Software engineering

## Advantages of Prolog -

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

## Disadvantages of Prolog -

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

**Facts in Prolog**

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

## Structure of Facts -

### **Components**

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

### Syntax

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

### Example

Let's consider a simple family relationship:

parent(alice, bob).

In this example:

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

## Key Points

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

# Rules in Prolog

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

## Example Rules

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

## Syntax Elements

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

## 

## Examples in Prolog Syntax

happy(lili) :- dances(lili).

hungry(tom) :- search\_for\_food(tom).

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

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

# **Queries in Prolog**

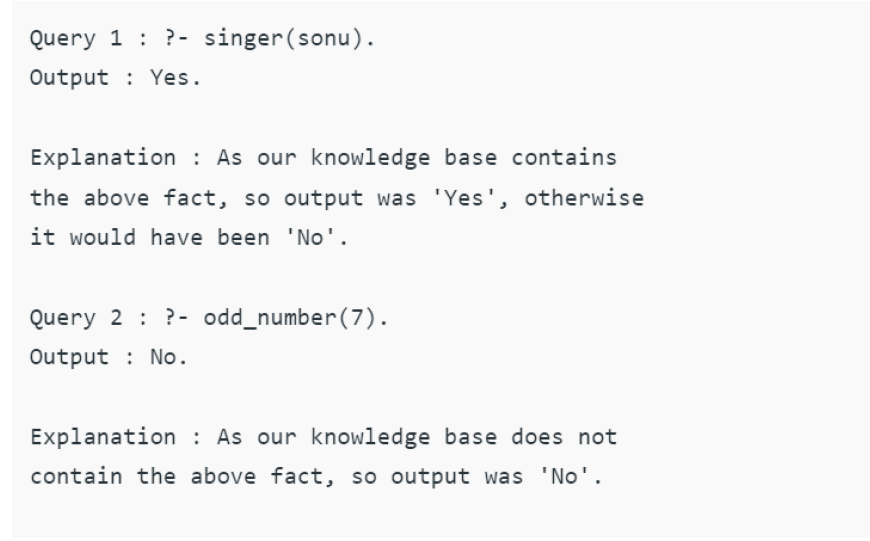
Queries are questions about relationships between objects and object properties. They allow us to extract information from Prolog's knowledge base.

## Example Queries

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

## How Queries Work?

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



# Predicates in Prolog

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

## Key Characteristics -

### Represents Relationships

Examples:

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

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

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

### Used in Facts and Rules

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

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

* **Rules**: Define complex relationships

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

### Arguments

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

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

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

### Arity

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

# Clauses in Prolog

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

## Types of Clauses -

### Facts

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

parent(alice, bob).

### Rules

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

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

## Key Points

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

# Features of Prolog

## 1. Logic Programming Paradigm

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

## 2. Core Mechanisms

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

## 3. Key Features

* Symbolic Computation
* AI Applications
* Database Integration
* Programming Style Flexibility

## 4. Advantages

* Expressive power
* Declarative nature
* Robust search mechanisms

## 5. Disadvantages

* Efficiency concerns
* Debugging challenges

# Symbolic Language Aspects

## 1. Knowledge Representation

### Facts

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

### Rules

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

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

### Variables

* Start with uppercase or underscore
* Represent unknown values

## 2. Symbolic Computation

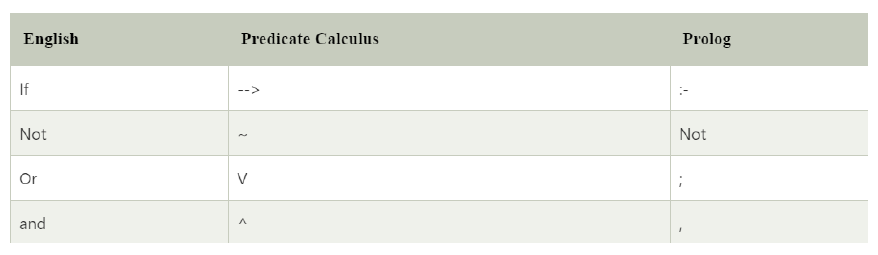
* Unification for pattern matching
* Inference engine for conclusions

## 3. Advantages

* Expressive representation
* Flexible adaptation
* Human-readable code

**Symbols in Prolog**

Using the following truth-functional symbols, the Prolog expressions are comprised. These symbols have the same interpretation as in the predicate calculus.



# Data Types in Prolog

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

## Basic Terms -

### Atoms

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

### Numbers

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

### Variables

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

### Compound Terms -

* Composed of:
  + Functor (an atom)
  + Arguments (other terms)
* Written as: functor(arg1, arg2, ...)
* Example: person\_friends(zelda,[tom,jim])

## Special Cases

### Lists

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

### Strings

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

# Metaprogramming in Prolog

## Key Concepts -

### Introspection Capabilities

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

### Meta-predicates

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

### Term Manipulation

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

## Example Meta-interpreter

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

solve(Goal) :-

Goal.

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

solve(Goal) :-

clause(Goal, Body),

solve(Body).

## Applications

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

## Advantages

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

# Expert System Design in Prolog

## Core Components -

### Knowledge Base

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

### Inference Engine

* Built-in unification
* Backtracking mechanism

### User Interface

* Command-line or GUI
* Input/output handling

## Design Process

### 1. Knowledge Acquisition

* Gather expert domain knowledge
* Identify key relationships

### 2. Knowledge Representation

* Convert to Prolog facts/rules
* Design data structures

### 3. Inference Implementation

* Control strategy design
* Optimization

### 4. Interface Design

* User interaction
* Result presentation

### 5. Testing

* Scenario validation
* Accuracy verification

## Example: Disease Diagnosis System

*% Facts*

has\_symptom(patient1, cough).

has\_symptom(patient1, runny\_nose).

has\_symptom(patient1, sore\_throat).

*% Rules*

diagnose(patient, common\_cold) :-

has\_symptom(patient, cough),

has\_symptom(patient, runny\_nose),

has\_symptom(patient, sore\_throat).

*% Query*

?- diagnose(patient1, Disease).

## Advantages

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

## Limitations

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

**Installing Prolog on Windows**

To install Prolog on Windows, follow these steps:

#### ****Step 1: Download SWI-Prolog****

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

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

#### ****Step 2: Install SWI-Prolog****

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

#### ****Step 3: Verify Installation****

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

prolog.

If Prolog starts successfully, the installation is complete.

## ****Basic "Hello, World!" Program in Prolog****

Code ::

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

Output ::



**EXPERIMENT - 2**

AIM ::

Write simple fact for the statements using PROLOG.

Ram likes mango.

Seema is a girl.

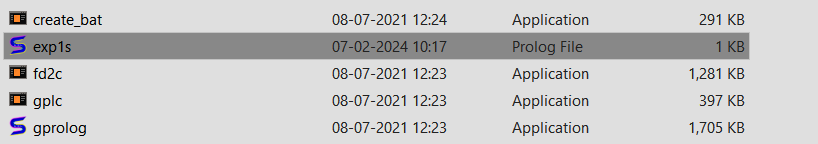
Bill likes Cindy.

Rose is red.

John owns gold.

Program ::

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

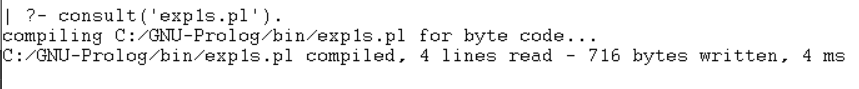


1. Write the facts according to the question.



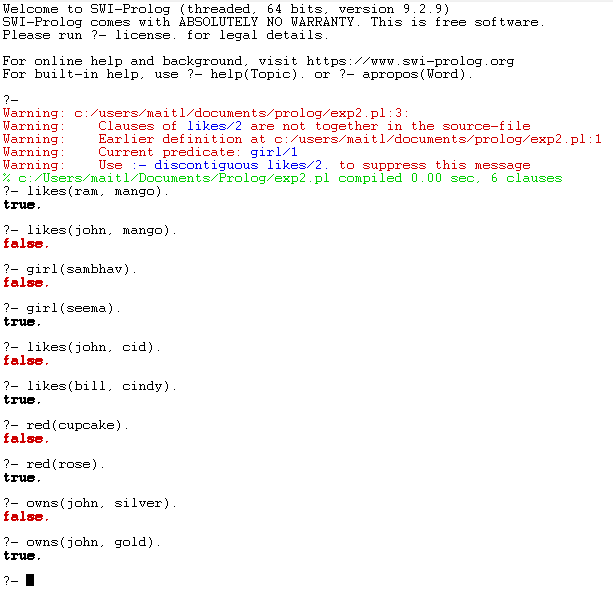
1. Open the prolog console and use the ‘consult’ command to compile the file and use

it as a knowledge base.



1. Write the appropriate queries in the console for verification. Prolog uses negation

as failure i.e. not p is true if p cannot be derived



**EXPERIMENT - 3**

AIM ::

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

Program ::

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

F is (C \* 9 / 5) + 32.

f\_to\_c(F, C) :-

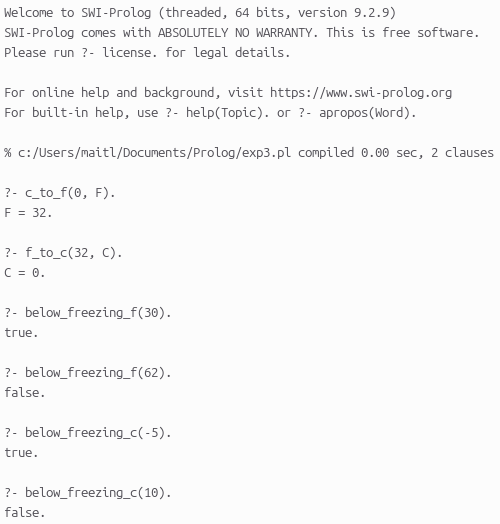
C is (F - 32) \* 5 / 9.

below\_freezing\_f(F) :-

F =< 32.

below\_freezing\_c(C) :-

C =< 0.



Output ::

**EXPERIMENT - 4**

AIM ::

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

Program ::

Tree ::

**a**

**/ \**

**b c**

**/ \ / \**

**d e f g**

**% Define the graph as facts (edges)**

**edge(a, b).**

**edge(a, c).**

**edge(b, d).**

**edge(b, e).**

**edge(c, f).**

**edge(c, g).**

**% Check if a node exists in the graph (as a parent or child)**

**node\_exists(Node) :-**

**edge(Node, \_); % Check if the node has an outgoing edge (parent)**

**edge(\_, Node). % Check if the node has an incoming edge (child)**

**% Breadth-First Search Traversal**

**bfs(Start, Goal) :-**

**bfs\_helper([Start], Goal, []).**

**bfs\_helper([Goal | \_], Goal, Path) :-**

**reverse([Goal | Path], FinalPath),**

**print\_path(FinalPath).**

**bfs\_helper([Node | Rest], Goal, Visited) :-**

**findall(Neighbor, (edge(Node, Neighbor), \+ member(Neighbor, Visited)), Neighbors),**

**append(Rest, Neighbors, NewQueue),**

**bfs\_helper(NewQueue, Goal, [Node | Visited]).**

**% Print the path from Start to Goal**

**print\_path([]) :-**

**writeln('No path found.').**

**print\_path([Node]) :-**

**write(Node), nl.**

**print\_path([Node | Rest]) :-**

**write(Node), write(' - '),**

**print\_path(Rest).**

**% Print the path from Start to Goal in a parent-child relationship**

**print\_parent([]).**

**print\_parent([Node]) :-**

**write(Node), nl.**

**print\_parent([Node | Rest]) :-**

**write(Node), write(' - '),**

**print\_parent(Rest).**

**% Print the path from Start to Goal in a reverse relationship**

**print\_reverse\_path([]).**

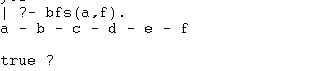
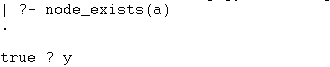
**print\_reverse\_path([Node]) :-**

**write(Node), nl.**

**print\_reverse\_path([Node | Rest]) :-**

**print\_reverse\_path(Rest),**

**write(Node), write(' - ').**

****

Output ::

**EXPERIMENT - 5**

AIM ::

Write a program to implement Water Jug Problem.

**:- dynamic visited\_state/2. % Declare visited\_state as dynamic**

Program ::

**% Define a predicate that takes user input and starts the process**

**fill(X, Y) :-**

**retractall(visited\_state(\_, \_)), % Clear visited states before starting**

**assertz(visited\_state(X, Y)),**

**state(X, Y, [(X, Y)]). % Pass an empty list to collect steps**

**% Goal State: 4-Litre jug should have exactly 2 Litres**

**state(2, Y, Steps) :-**

**format("Goal reached: (2, ~d)\n", [Y]),**

**print\_steps(Steps), % Print the steps**

**!. % Stop further execution**

**% Fill the 4-Litre Jug**

**state(X, Y, Steps) :-**

**X < 4,**

**\+ visited\_state(4, Y),**

**assertz(visited\_state(4, Y)),**

**format("Fill 4-Litre Jug: (~d, ~d) --> (4, ~d)\n", [X, Y, Y]),**

**state(4, Y, [(X, Y) | Steps]). % Record the current state in the steps**

**% Fill the 3-Litre Jug**

**state(X, Y, Steps) :-**

**Y < 3,**

**\+ visited\_state(X, 3),**

**assertz(visited\_state(X, 3)),**

**format("Fill 3-Litre Jug: (~d, ~d) --> (~d, 3)\n", [X, Y, X]),**

**state(X, 3, [(X, Y) | Steps]). % Record the current state in the steps**

**% Empty the 4-Litre Jug**

**state(X, Y, Steps) :-**

**X > 0,**

**\+ visited\_state(0, Y),**

**assertz(visited\_state(0, Y)),**

**format("Empty 4-Litre Jug: (~d, ~d) --> (0, ~d)\n", [X, Y, Y]),**

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

**% Empty the 3-Litre Jug**

**state(X, Y, Steps) :-**

**Y > 0,**

**\+ visited\_state(X, 0),**

**assertz(visited\_state(X, 0)),**

**format("Empty 3-Litre Jug: (~d, ~d) --> (~d, 0)\n", [X, Y, X]),**

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

**% Pour from 3-Litre to 4-Litre (until full)**

**state(X, Y, Steps) :-**

**X + Y >= 4,**

**Y > 0,**

**NEW\_Y is Y - (4 - X),**

**\+ visited\_state(4, NEW\_Y),**

**assertz(visited\_state(4, NEW\_Y)),**

**format("Pour from 3-Litre to 4-Litre: (~d, ~d) --> (4, ~d)\n", [X, Y, NEW\_Y]),**

**state(4, NEW\_Y, [(X, Y) | Steps]). % Record the current state in the steps**

**% Pour from 4-Litre to 3-Litre (until full)**

**state(X, Y, Steps) :-**

**X + Y >= 3,**

**X > 0,**

**NEW\_X is X - (3 - Y),**

**\+ visited\_state(NEW\_X, 3),**

**assertz(visited\_state(NEW\_X, 3)),**

**format("Pour from 4-Litre to 3-Litre: (~d, ~d) --> (~d, 3)\n", [X, Y, NEW\_X]),**

**state(NEW\_X, 3, [(X, Y) | Steps]). % Record the current state in the steps**

**% Pour all from 3-Litre to 4-Litre**

**state(X, Y, Steps) :-**

**X + Y < 4,**

**Y > 0,**

**NEW\_X is X + Y,**

**\+ visited\_state(NEW\_X, 0),**

**assertz(visited\_state(NEW\_X, 0)),**

**format("Pour all from 3-Litre to 4-Litre: (~d, ~d) --> (~d, 0)\n", [X, Y, NEW\_X]),**

**state(NEW\_X, 0, [(X, Y) | Steps]). % Record the current state in the steps**

**% Pour all from 4-Litre to 3-Litre**

**state(X, Y, Steps) :-**

**X + Y < 3,**

**X > 0,**

**NEW\_Y is X + Y,**

**\+ visited\_state(0, NEW\_Y),**

**assertz(visited\_state(0, NEW\_Y)),**

**format("Pour all from 4-Litre to 3-Litre: (~d, ~d) --> (0, ~d)\n", [X, Y, NEW\_Y]),**

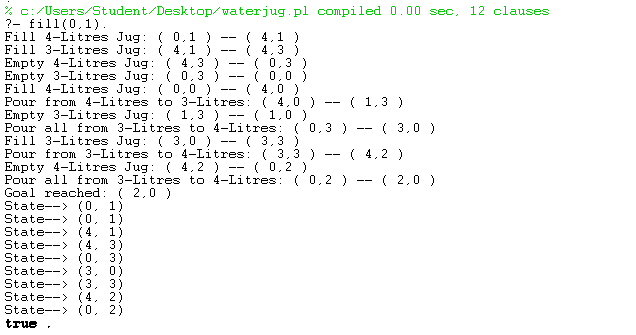
**state(0, NEW\_Y, [(X, Y) | Steps]). % Record the current state in the steps**

**% Predicate to print all steps in the solution path**

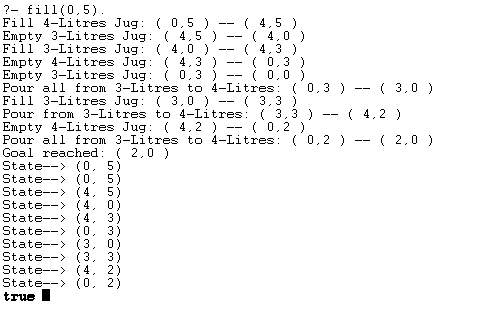
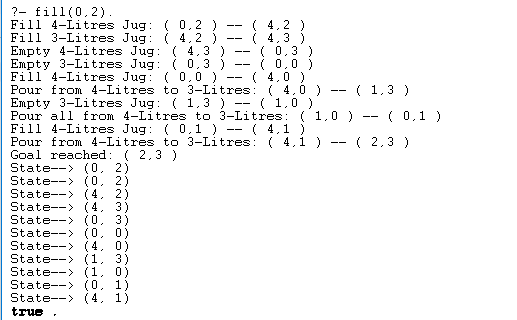
**print\_steps([]).**

**print\_steps([(X, Y) | Rest]) :-**

**print\_steps(Rest),**

**f****ormat("State --> (~d, ~d)\n", [X, Y]).**

Output ::



**EXPERIMENT - 6**

AIM ::

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

**% Define the initial empty board**

Program ::

**initial\_board([empty, empty, empty, empty, empty, empty, empty, empty, empty]).**

**% Start the game**

**start\_game :-**

**initial\_board(Board),**

**play\_game(Board, x).**

**% Game loop**

**play\_game(Board, Player) :-**

**print\_board(Board),**

**( winner(Board, x) -> write('Player X wins!'), nl**

**; winner(Board, o) -> write('Player O wins!'), nl**

**; full\_board(Board) -> write('It\'s a draw!'), nl**

**; make\_move(Board, Player, NewBoard),**

**next\_player(Player, NextPlayer),**

**play\_game(NewBoard, NextPlayer)**

**).**

**% Print the Tic-Tac-Toe board**

**print\_board([A, B, C, D, E, F, G, H, I]) :-**

**write(' '), display\_symbol(A), write(' | '), display\_symbol(B), write(' | '), display\_symbol(C), nl,**

**write('-----------'), nl,**

**write(' '), display\_symbol(D), write(' | '), display\_symbol(E), write(' | '), display\_symbol(F), nl,**

**write('-----------'), nl,**

**write(' '), display\_symbol(G), write(' | '), display\_symbol(H), write(' | '), display\_symbol(I), nl, nl.**

**% Display 'X', 'O', or an empty position**

**display\_symbol(empty) :- write(' ').**

**display\_symbol(X) :- write(X).**

**% Check if a player has won**

**winner(Board, Player) :-**

**( row\_win(Board, Player)**

**; col\_win(Board, Player)**

**; diag\_win(Board, Player)**

**).**

**% Check winning row conditions**

**row\_win([A, B, C, \_, \_, \_, \_, \_, \_], Player) :- A = Player, B = Player, C = Player.**

**row\_win([\_, \_, \_, D, E, F, \_, \_, \_], Player) :- D = Player, E = Player, F = Player.**

**row\_win([\_, \_, \_, \_, \_, \_, G, H, I], Player) :- G = Player, H = Player, I = Player.**

**% Check winning column conditions**

**col\_win([A, \_, \_, D, \_, \_, G, \_, \_], Player) :- A = Player, D = Player, G = Player.**

**col\_win([\_, B, \_, \_, E, \_, \_, H, \_], Player) :- B = Player, E = Player, H = Player.**

**col\_win([\_, \_, C, \_, \_, F, \_, \_, I], Player) :- C = Player, F = Player, I = Player.**

**% Check winning diagonal conditions**

**diag\_win([A, \_, \_, \_, E, \_, \_, \_, I], Player) :- A = Player, E = Player, I = Player.**

**diag\_win([\_, \_, A, \_, E, \_, I, \_, \_], Player) :- A = Player, E = Player, I = Player.**

**% Check if the board is full**

**full\_board(Board) :-**

**\+ member(empty, Board).**

**% Make a move for the current player**

**make\_move(Board, Player, NewBoard) :-**

**repeat,**

**write('Enter a position (1-9): '), read(Pos),**

**( valid\_move(Board, Pos) -> replace(Pos, Player, Board, NewBoard), !**

**; write('Invalid move! Try again.'), nl, fail**

**).**

**% Validate the move (position must be between 1-9 and must be empty)**

**valid\_move(Board, Pos) :-**

**integer(Pos), Pos >= 1, Pos =< 9, nth1(Pos, Board, empty).**

**% Replace a position on the board with the player's symbol**

**replace(Pos, Player, Board, NewBoard) :-**

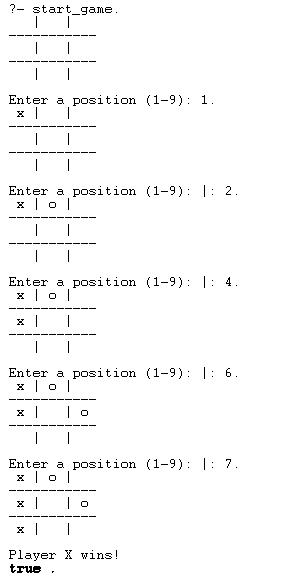
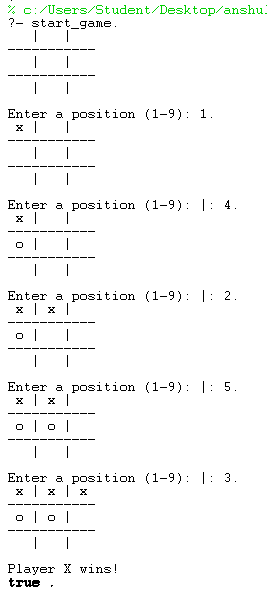
**nth1(Pos, Board, empty, TempBoard),**

**nth1(Pos, NewBoard, Player, TempBoard).**

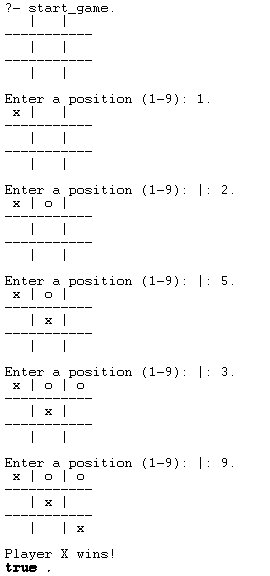
**% Switch to the next player**

**next\_player(x, o).**

**next\_player(o, x).**



Output ::

****

**EXPERIMENT - 7**

AIM ::

Write a Prolog program to remove Punctuations from text.

**% Define punctuation characters by their ASCII codes**

Program ::

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

**member(Char, [46, 44, 59, 58, 33, 63, 39, 34, 45, 40, 41, 91, 93, 123, 125]).**

**% Predicate to remove punctuations from a list of character codes**

**remove\_punctuations([], []). % Base case: empty list returns empty list**

**remove\_punctuations([H | T], Result) :-**

**is\_punctuation(H), % If the character is punctuation, skip it**

**remove\_punctuations(T, Result).**

**remove\_punctuations([H | T], [H | Result]) :- % Otherwise, keep the character**

**\+ is\_punctuation(H),**

**remove\_punctuations(T, Result).**

**% Convert input string to output without punctuations**

**clean\_string(Input, Output) :-**

**string\_codes(Input, Codes), % Convert string to list of character codes**

**remove\_punctuations(Codes, CleanCodes), % Remove punctuation marks**

**string\_codes(Output, CleanCodes). % Convert back to string**

**% Main program that takes user input and displays cleaned string**

**start :-**

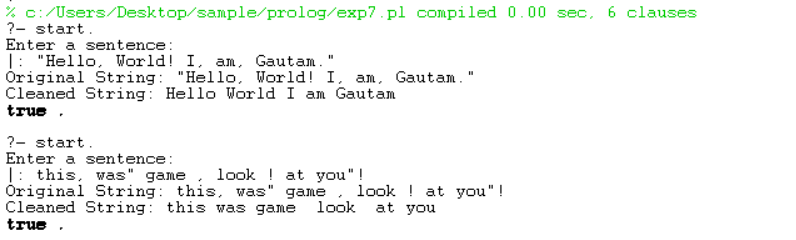
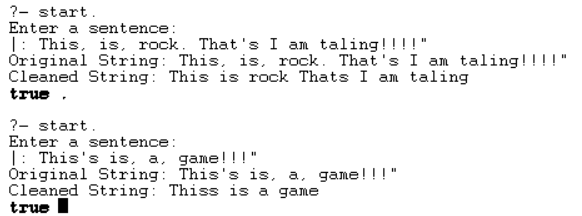
**write('Enter a sentence: '), nl,**

**read\_line\_to\_string(user\_input, Input), % Read input from user**

**clean\_string(Input, Output),**

**write('Original String: '), write(Input), nl,**

**write('Cleaned String: '), write(Output), nl.**



Output ::

**EXPERIMENT - 8**

AIM ::

Write a Prolog program to sort a sentence

**% Main predicate to read input and sort the sentence**

Program ::

**sort\_sentence :-**

**write('Enter a sentence: '), nl,**

**read\_line\_to\_string(user\_input, Sentence), % Read sentence from user input**

**split\_string(Sentence, " ", "", Words), % Split the sentence into words**

**msort(Words, SortedWords), % Sort words alphabetically**

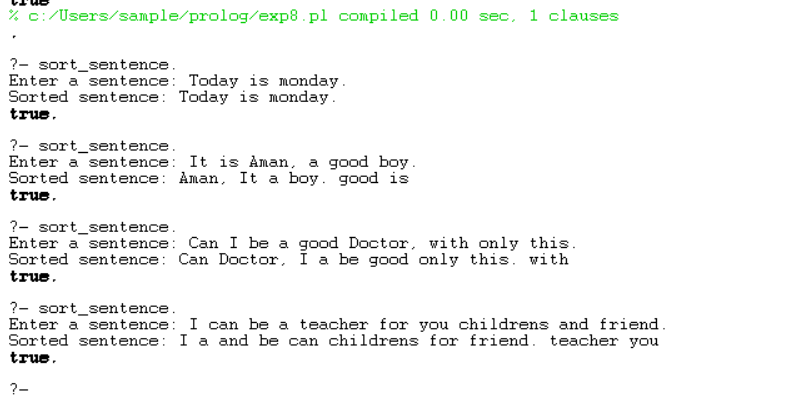
**atomic\_list\_concat(SortedWords, ' ', SortedSentence), % Combine sorted words back into a sentence**

**write('Sorted sentence: '), writeln(SortedSentence).**

**% To run:**

**% ?- sort\_sentence.**

**% Then input your sentence when prompted.**



Output ::

**EXPERIMENT - 9**

AIM ::

Write a program to implement Hangman game using Python.

**import random**

Program ::

**# List of words to choose from**

**words = ['python', 'java', 'hangman', 'computer', 'programming', 'developer']**

**# Function to choose a random word from the list**

**def choose\_word():**

**return random.choice(words)**

**# Function to display the current state of the word**

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

**display = ""**

**for letter in word:**

**if letter in guessed\_letters:**

**display += letter**

**else:**

**display += "\_"**

**return display**

**# Main hangman game function**

**def hangman():**

**word = choose\_word()**

**guessed\_letters = []**

**attempts = 6 # Number of incorrect attempts before the game is over**

**print("Welcome to Hangman!")**

**print("Try to guess the word, you have", attempts, "attempts.")**

**while attempts > 0:**

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

**print(f"Guessed letters: {', '.join(guessed\_letters)}")**

**print(f"Attempts left: {attempts}")**

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

**# Validate the input**

**if len(guess) != 1 or not guess.isalpha():**

**print("Please enter a single valid letter.")**

**continue**

**# Check if the letter has already been guessed**

**if guess in guessed\_letters:**

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

**continue**

**# Add the guessed letter to the guessed\_letters list**

**guessed\_letters.append(guess)**

**# Check if the guessed letter is in the word**

**if guess not in word:**

**attempts -= 1**

**print(f"Wrong guess! {guess} is not in the word.")**

**# Check if the user has guessed the word**

**if all(letter in guessed\_letters for letter in word):**

**print(f"Congratulations! You've guessed the word: {word}")**

**break**

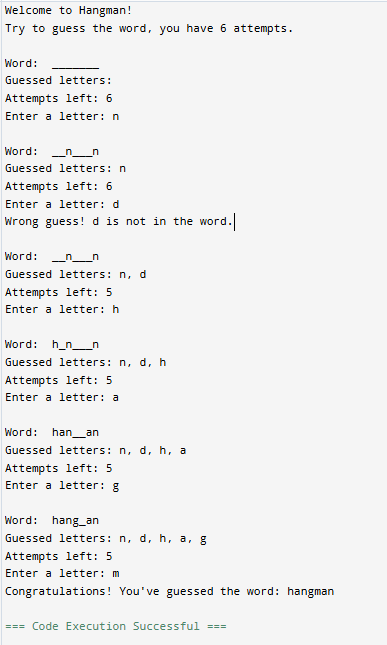
**else:**

**print(f"Game Over! The word was: {word}")**

**# Run the hangman game**

**if \_\_name\_\_ == "\_\_main\_\_":**

**hangman()**



Output ::

**EXPERIMENT - 10**

AIM ::

Write a program to implement Hangman game using PROLOG.

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

Program ::

**% Start the game**

**hangman :-**

**word(Word),**

**atom\_chars(Word, WordList),**

**length(WordList, N),**

**play(WordList, N, [], 6). % Max 6 wrong attempts**

**% Play function**

**play(WordList, \_, Guessed, 0) :-**

**write('Game Over! You ran out of attempts.'), nl,**

**write('The correct word was: '), write(WordList), nl, !.**

**play(WordList, \_, Guessed, \_) :-**

**word\_complete(WordList, Guessed),**

**write('Congratulations! You guessed the word: '), write(WordList), nl, !.**

**play(WordList, N, Guessed, Attempts) :-**

**display\_progress(WordList, Guessed),**

**write('Enter your guess: '),**

**read(Char),**

**( member(Char, WordList) ->**

**write('Correct Guess!'), nl,**

**play(WordList, N, [Char|Guessed], Attempts)**

**;**

**write('Wrong Guess!'), nl,**

**NewAttempts is Attempts - 1,**

**write('Remaining Attempts: '), write(NewAttempts), nl,**

**play(WordList, N, Guessed, NewAttempts)**

**).**

**% Check if all letters are guessed**

**word\_complete([], \_).**

**word\_complete([H|T], Guessed) :-**

**member(H, Guessed),**

**word\_complete(T, Guessed).**

**% Display word progress**

**display\_progress([], \_):- nl.**

**display\_progress([H|T], Guessed) :-**

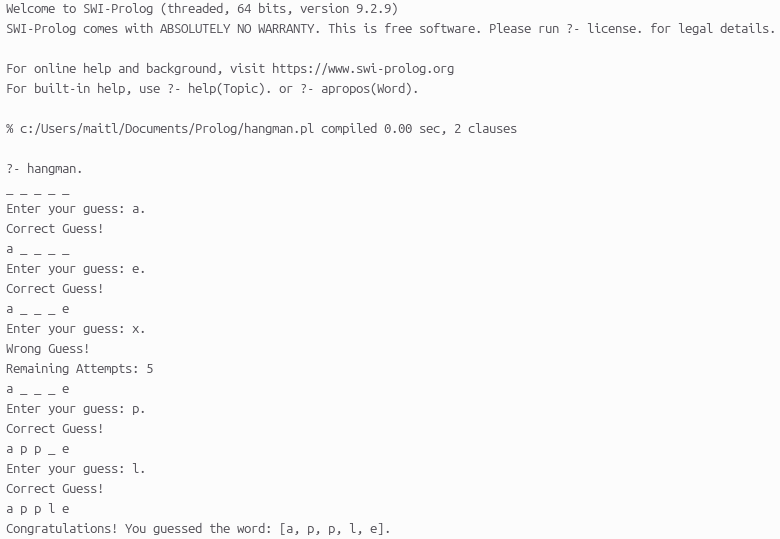
**( member(H, Guessed) -> write(H) ; write('\_') ),**

**write(' '),**

**display\_progress(T, Guessed).**

**% Start game**

**?- hangman.**



Output ::

**EXPERIMENT - 11**

AIM ::

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

Program ::

**import nltk**

**from nltk.corpus import stopwords**

**from nltk.tokenize import word\_tokenize**

**# Function to remove stopwords from text**

**def remove\_stopwords(text):**

**stop\_words = set(stopwords.words('english')) # Load NLTK's stop words**

**words = word\_tokenize(text) # Tokenize text into words**

**filtered\_words = [word for word in words if word.lower() not in stop\_words]**

**return ' '.join(filtered\_words) # Join words back into a sentence**

**# Read text from file**

**input\_file = "input.txt"**

**with open(input\_file, "r") as file:**

**text = file.read()**

**# Process text to remove stopwords**

**filtered\_text = remove\_stopwords(text)**

**# Write the cleaned text to output file**

**output\_file = "output.txt"**

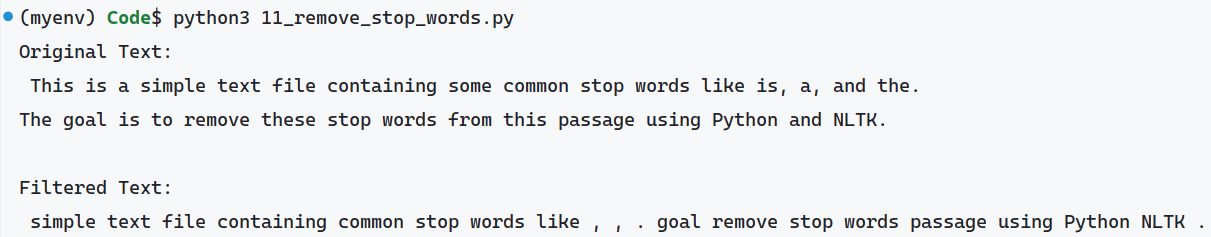
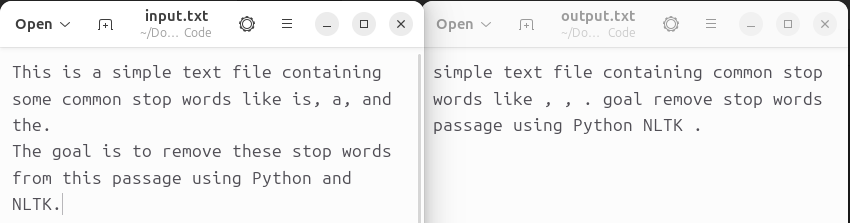
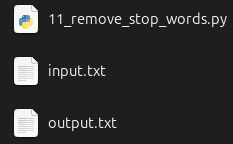
**with open(output\_file, "w") as file:**

**file.write(filtered\_text)**

**# Print result**

**print("Original Text:\n", text)**

**print("Filtered Text:\n", filtered\_text)**



Output ::

**EXPERIMENT - 12**

AIM ::

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

**import nltk**

Program ::

**from nltk.stem import PorterStemmer**

**from nltk.tokenize import word\_tokenize**

**# Function to perform stemming**

**def stem\_sentence(sentence):**

**# Initialize the Porter Stemmer**

**porter\_stemmer = PorterStemmer()**

**# Tokenize the sentence into words**

**words = word\_tokenize(sentence)**

**# Stem each word in the sentence**

**stemmed\_words = [porter\_stemmer.stem(word) for word in words]**

**# Join the stemmed words back into a sentence**

**return ' '.join(stemmed\_words)**

**# Example usage**

**sentence = "The cat is jumping and dogs are running fast."**

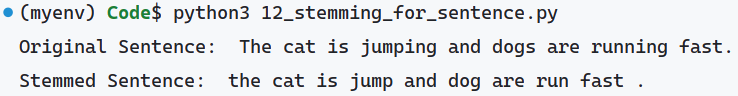
**stemmed\_sentence = stem\_sentence(sentence)**

**# Print the original and stemmed sentences**

**print("Original Sentence: ", sentence)**

**print("Stemmed Sentence: ", stemmed\_sentence)**

Output ::



**EXPERIMENT - 13**

AIM ::

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

**import nltk**

Program ::

**from nltk.tokenize import word\_tokenize**

**from nltk import pos\_tag**

**# Function to perform POS tagging**

**def pos\_tagging(sentence):**

**# Tokenize the sentence into words**

**words = word\_tokenize(sentence)**

**# Perform POS tagging**

**tagged\_words = pos\_tag(words)**

**return tagged\_words**

**# Example usage**

**sentence = "The cat is jumping and dogs are running fast."**

**tagged\_sentence = pos\_tagging(sentence)**

**# Print the original sentence and its POS tags**

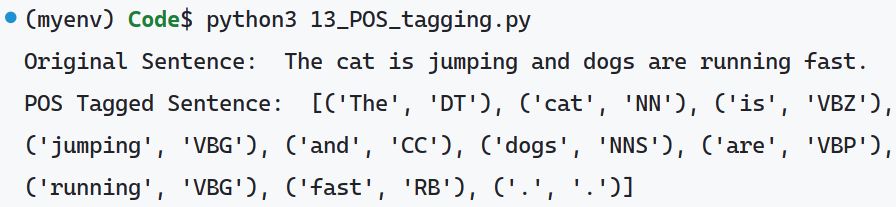
**print("Original Sentence: ", sentence)**

**print("POS Tagged Sentence: ", tagged\_sentence)**

### ****Explanation of POS Tags**:**

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

Output ::



**EXPERIMENT - 14**

AIM ::

Write a program to implement Lemmatization using NLTK.

**import nltk**

Program ::

**from nltk.tokenize import word\_tokenize**

**from nltk.stem import WordNetLemmatizer**

**# Initialize the WordNet Lemmatizer**

**lemmatizer = WordNetLemmatizer()**

**# Function to perform Lemmatization**

**def lemmatize\_sentence(sentence):**

**# Tokenize the sentence into words**

**words = word\_tokenize(sentence)**

**# Lemmatize each word**

**lemmatized\_words = [lemmatizer.lemmatize(word, pos='v') for word in words] # 'v' for verb, you can change it depending on the word**

**return ' '.join(lemmatized\_words)**

**# Example usage**

**sentence = "The cats are running and they are better than others."**

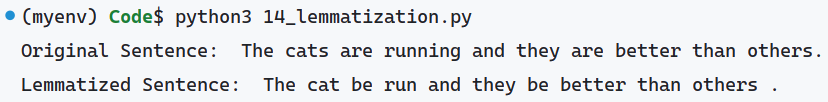
**lemmatized\_sentence = lemmatize\_sentence(sentence)**

**# Print the original and lemmatized sentences**

**print("Original Sentence: ", sentence)**

**print("Lemmatized Sentence: ", lemmatized\_sentence)**

Output ::



**EXPERIMENT - 15**

AIM ::

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

**import nltk**

Program ::

**from nltk.tokenize import word\_tokenize**

**from nltk.corpus import stopwords**

**from nltk.classify import NaiveBayesClassifier**

**from nltk.classify.util import accuracy**

**# Sample labeled data for training the classifier**

**training\_data = [**

**("I love programming", "positive"),**

**("I hate bugs", "negative"),**

**("This is so great", "positive"),**

**("I am so angry at this error", "negative"),**

**("Coding is awesome", "positive"),**

**("This is terrible", "negative")**

**]**

**# Preprocess the data: tokenization and removing stopwords**

**stop\_words = set(stopwords.words("english"))**

**def preprocess(text):**

**tokens = word\_tokenize(text.lower()) # Tokenize and convert to lowercase**

**return {word: True for word in tokens if word not in stop\_words and word.isalpha()}**

**# Feature extraction**

**train\_set = [(preprocess(text), label) for text, label in training\_data]**

**# Train the Naive Bayes classifier**

**classifier = NaiveBayesClassifier.train(train\_set)**

**# Test the classifier with a new sentence**

**def classify\_sentence(sentence):**

**features = preprocess(sentence)**

**return classifier.classify(features)**

**# Example usage**

**test\_sentence = "I love coding but debugging is hard."**

**print(f"Sentence: {test\_sentence}")**

**print(f"Classification: {classify\_sentence(test\_sentence)}")**

**# Print classifier accuracy on training data**

**print(f"Classifier accuracy: {accuracy(classifier, train\_set) \* 100}%")**

Output ::

