(PART – B): Prolog Programs

Practical - 1

Aim: Write a PROLOG program to represent Facts. Make some queries based on the facts.

- **Objective:** understand about facts and making queries and being familiar with prolog programing using GNU Prolog.
- **Theory:**

  In Prolog, facts are basic assertions about objects, their properties, or relationships between objects. They represent information that is known to be true and do not contain any conditions. A fact is generally written as a predicate that describes a relationship or attribute of objects.

  Representing Facts in Prolog

  A fact is written in the form:

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

  Example Facts:

  ```
  parent(john, mary).
  parent(mary, anne).
  ```

  Representing properties:

  ```
  likes(alice, pizza).
  likes(bob, ice_cream).
  ```

  Making Queries Based on Facts

  Queries in Prolog are used to check if certain facts are true or to find out what satisfies a particular condition. A query is written in a similar format to a fact but is used interactively to search for information.

  Check if a fact is true:

  ```
  ?- parent(john, mary).
  ```

  Find all instances:

  ```
  ?- parent(mary, X).
  ```

  Find who likes pizza:

  ```
  ?- likes(X, pizza).
  ```

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

To write and run a Prolog program using GNU Prolog, follow these steps:

Step 1: Install GNU Prolog

Step 2: Write a Prolog Program

1. Create a new file with a `.pl` extension. You can use any text editor like `vim`, `nano`, `gedit`, or any IDE.
2. Write your Prolog facts, rules, and queries in this file. For example, create a file named `family.pl` with the following content:

```
parent(john, mary).
  parent(mary, anne).
  parent(john, tom).
  likes(alice, pizza).
  likes(bob, ice_cream).
```

3. Save the file.

Step 3: Start the GNU Prolog Interpreter

1. Open a terminal or command prompt.
2. Navigate to the directory where your Prolog file is located.
3. Start the Prolog interpreter by typing:

```
gprolog
```

Step 4: Load the Prolog File

1. Once inside the Prolog interpreter (you will see the `| ?-` prompt), load your Prolog file using the `consult/1` predicate:

```
| ?- consult('family.pl').
```

Or you can use the shorthand:

```
| ?- ['family.pl'].
```

Step 5: Run Queries

1. After loading the file, you can run queries to test the facts and rules. For example:

Check if John is a parent of Mary:

```
| ?- parent(john, mary).
```

Prolog will return `true.` if the fact exists.

Find all children of John:

```
| ?- parent(john, X).
   This will return:
   X = mary ;
   X = tom ;
```

Check if Alice likes pizza:

```
| ?- likes(alice, pizza).
```

2. To ask another query, type a semicolon (`;`) to continue or press `Enter` to finish the current query.

Step 6: Exit the Prolog Interpreter

To exit the GNU Prolog interpreter, type:

```
| ?- halt.
```

- **Code:**

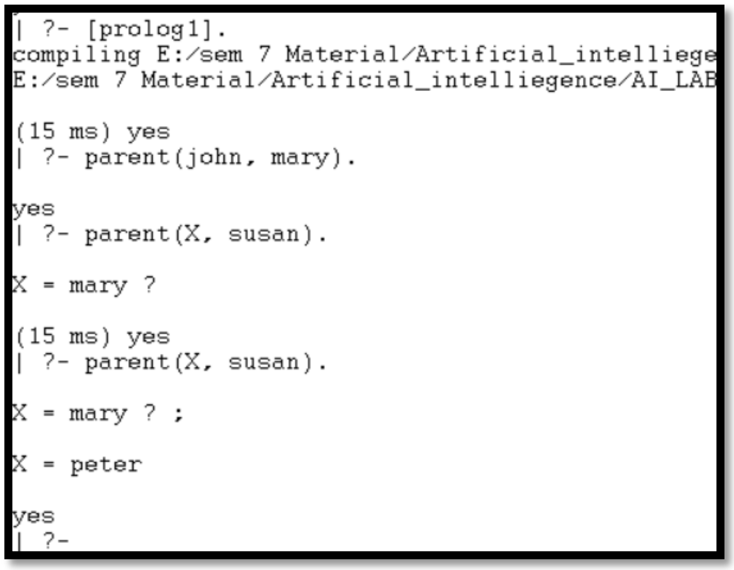
Facts:

```
parent(john, mary).
parent(jane, mary).
parent(mary, susan).
parent(peter, susan).
```

Query:

```
?- parent(john, mary).
?- parent(X, susan).
```

- **Output:**



```
| ?- [prolog1].
compiling E:/sem 7 Material/Artificial_intelliege
E:/sem 7 Material/Artificial_intelliegence/AI_LAB

(15 ms) yes
| ?- parent(john, mary).

yes
| ?- parent(X, susan).

X = mary ?

(15 ms) yes
| ?- parent(X, susan).

X = mary ? ;

X = peter

yes
| ?-
```

- **Signature:**

Practical - 2

Aim: Write a PROLOG program to represent Rules. Make some queries based on the facts and rules.

- **Objective:** understand about rules and making queries and being familiar with prolog programing using GNU Prolog.
- **Theory:**

In Prolog, rules define relationships between facts using logical implications. Rules allow you to express more complex relationships and derive new facts from existing ones. A rule is composed of a head and a body. The body contains conditions that must be met for the head to be considered true.

Structure of a Rule

A rule in Prolog is written as:

```
head :- body.
```

`head`: The conclusion or fact that the rule defines.

`body`: A series of conditions that must be true for the head to be true. Conditions are usually connected using logical conjunctions (`,` for AND).

Example Rules

Let's look at a few examples to understand how rules are defined:

Example 1: Defining a Grandparent Relationship

If we want to define what it means for someone to be a grandparent, we can write:

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

`X` is a grandparent of `Y` if `X` is a parent of `Z` and `Z` is a parent of `Y`.

This rule allows us to infer a grandparent relationship from the parent relationships.

Example 2: Defining a Sibling Relationship

A rule that defines siblings could look like this:

```
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
```

`X` is a sibling of `Y` if `X` and `Y` share the same parent `Z`, and `X` is not equal to `Y`.

The condition `X \= Y` ensures that a person cannot be considered their own sibling.

Example 3: Defining a Likes Relationship Based on Shared Preferences

You can use rules to infer relationships like shared preferences:

```
friends(X, Y) :- likes(X, Z), likes(Y, Z), X \= Y.
```

`X` and `Y` are friends if they both like the same thing `Z`, and `X` is not the same as `Y`.

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

How to Represent Rules in Prolog

To define and use rules in Prolog:

1. Write the rules in a `.pl` file, just like you write facts.

2. Example content for `relationships.pl`:

```
parent(john, mary).
parent(mary, anne).
parent(john, tom).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
```

3. Load this file into the Prolog interpreter:

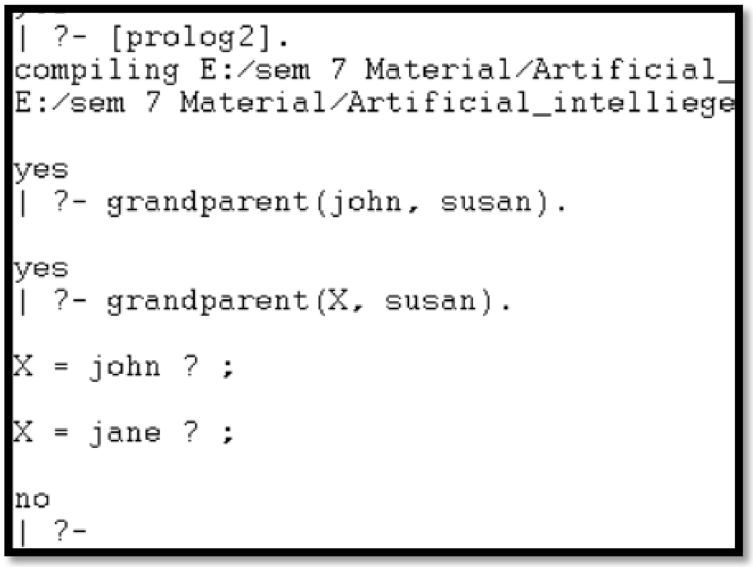
```
| ?- ['relationships.pl'].
```

- **Code:**

Facts:

```
parent(john, mary).
parent(jane, mary).
parent(mary, susan).
parent(peter, susan).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

- **Output:**



```
| ?- [prolog2].
compiling E:/sem 7 Material/Artificial_
E:/sem 7 Material/Artificial_intelliege

yes
| ?- grandparent(john, susan).

yes
| ?- grandparent(X, susan).

X = john ? ;

X = jane ? ;

no
| ?-
```

- **Signature:**

Practical - 3

Aim: Write a PROLOG program to define the relations using following predicates:

a) Define a predicate brother(X,Y) which holds iff X and Y are brothers.

b) Define a predicate cousin(X,Y) which holds iff X and Y are cousins.

c) Define a predicate grandson(X,Y) which holds iff X is a grandson of Y.

d) Define a predicate descendent(X,Y) which holds iff X is a descendent of Y.

Consider the following genealogical tree:

```
father(a,b).
father(a,c).
father(b,d).
father(b,e).
father(c,f).
```

Say which answers, and in which order, are generated by your definitions for the following queries in Prolog:

```
?- brother(X, Y).
?- cousin(X, Y).
?- grandson(X, Y).
?- descendent(X, Y).
```

- **Objective:** understand about making relation using prdicates and making queries and being familiar with prolog programing using GNU Prolog.
- **Theory:**

In Prolog, relations between objects are represented using predicates. A predicate is a symbolic expression that describes a relationship between a set of terms (objects or entities). It allows us to express logical relationships in the form of facts or rules.

Understanding Relations Using Predicates

A predicate can be thought of as a function that represents some property or relationship between entities. For example, parent(X, Y) might represent the relationship that "X is a parent of Y." Here, parent is the predicate, and X and Y are terms that represent individuals involved in this relationship.

Structure of a Predicate

A predicate is defined in the form:

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

predicate_name: The name of the relationship or property being represented.

arguments: The entities that participate in the relationship. They can be variables (starting with uppercase letters) or constants (starting with lowercase letters).

Types of Relations Using Predicates

1. **Unary Relations (Properties)**
    - A unary predicate has a single argument and describes a property of that argument.
    - **Example :**

```
is_student(alice).
```

2. **Binary Relations (Relations between Two Entities)**
   - A binary predicate has two arguments and describes a relationship between them.
   - **Example:**
     ```
     parent(john, mary).
     ```
3. **N-ary Relations (Relations among Multiple Entities)**
   - An n-ary predicate has n arguments and can describe more complex relationships.
   - **Example:**
     ```
     teaches(professor_smith, data_structures, university).
     ```

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

Step 1: Create a Prolog Source File

Open a text editor (such as vim, nano, gedit, or an IDE).
Create a new file with a .pl extension, for example, relations.pl.

Write Prolog facts and rules in the file.
Example: Defining Family Relationships

```
% Facts
parent(john, mary).
parent(john, tom).
parent(mary, anne).
parent(mary, mike).
parent(tom, lucy).

% Rule: X is a grandparent of Y if X is a parent of Z, and Z is a parent of Y.
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

% Rule: X and Y are siblings if they share the same parent.
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

% Rule: X likes Y if Y likes X.
likes(john, coffee).
likes(mary, tea).
likes(X, Y) :- likes(Y, X).
```

Save the file as relations.pl.

Step 2: Open the Prolog Interpreter

Open a terminal (or command prompt) on your computer.

Navigate to the directory where relations.pl is located using the cd command.

Start the GNU Prolog interpreter by typing:

```
gprolog
```

You should see the Prolog prompt | ?- indicating that the interpreter is running.

Step 3: Load the Prolog Source File

To load the relations.pl file into the interpreter, type the following at the Prolog prompt:

```
| ?- consult('relations.pl').
```

Or use the shorthand:

```
| ?- ['relations.pl'].
```

If the file loads successfully, Prolog will return yes.. Any syntax errors in the file will be reported here, and you'll need to fix them before proceeding.

Step 4: Query the Relations

Now that the facts and rules are loaded, you can make queries to test the relationships.

Find the grandparents of Anne:

```
| ?- grandparent(X, anne).
```

Prolog will return:

```
X = john.
```

This indicates that john is a grandparent of anne.

Check if Mary and Tom are siblings:

```
| ?- sibling(mary, tom).
```

Prolog will return:

```
false.
```

This indicates that mary and tom are not considered siblings based on the defined facts.

Find all children of John:

```
| ?- parent(john, X).
```

Prolog will return:

```
X = mary ;
X = tom.
```

The semicolon (;) allows you to see the next possible answer.

Check who likes tea:

```
| ?- likes(X, tea).
```

Prolog will return:

```
X = mary ;
X = tea.
```

This indicates that mary likes tea based on the facts and rules.

Find all sibling pairs:

```
| ?- sibling(X, Y).
```

Prolog will return pairs of siblings such as:

```
X = mary, Y = mike ;
X = mike, Y = mary.
```

Step 5: Exit the Prolog Interpreter

When you're done, exit the Prolog interpreter by typing:

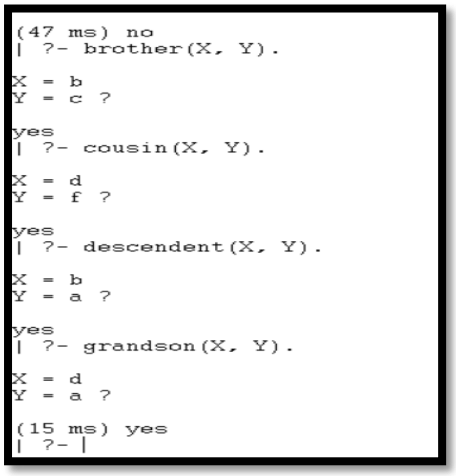
```
| ?- halt.
```

- **Code:**

  Facts:

```
father(a, b).
father(a, c).
father(b, d).
father(b, e).
father(c, f).

brother(X, Y) :- father(F, X), father(F, Y), X \= Y.
cousin(X, Y) :- father(F1, X), father(F2, Y), brother(F1, F2).
grandson(X, Y) :- father(Y, P), father(P, X).
descendent(X, Y) :- father(Y, X).
descendent(X, Y) :- father(Y, Z), descendent(X, Z).

?- brother(X, Y).
?- cousin(X, Y).
?- grandson(X, Y).
?- descendent(X, Y).
```

- **Output:**



```
(47 ms) no
| ?- brother(X, Y).

X = b
Y = c ?

yes
| ?- cousin(X, Y).

X = d
Y = f ?

yes
| ?- descendent(X, Y).

X = b
Y = a ?

yes
| ?- grandson(X, Y).

X = d
Y = a ?

(15 ms) yes
| ?- |
```

Practical - 4

Aim: Write a PROLOG program to perform addition, subtraction, multiplication and division of two numbers using arithmetic operators.

- **Objective:** to do mathematical operation using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

In Prolog, arithmetic operations such as addition, subtraction, multiplication, and division can be performed using built-in predicates and operators. Prolog treats arithmetic expressions differently from other languages—it requires explicit evaluation of expressions.

Arithmetic in Prolog: Overview

- Prolog uses the `is` operator to evaluate arithmetic expressions.

- Expressions like `+`, `-`, `*`, `/` need to be evaluated explicitly using `is` to perform calculations.

- Arithmetic operations are not automatically evaluated when stated as part of a query. Instead, you need to use `is` to compute the result.

Prolog uses the `is` operator to evaluate arithmetic expressions. The general form is:

Result is Expression.

`Result` is the variable where the result will be stored.

`Expression` is the arithmetic expression to be evaluated.

Example Operations

Let's look at how to perform basic arithmetic operations in Prolog:

Addition

To add two numbers:

```
?- X is 5 + 3.
```

- This will result in:

```
  X = 8.
```

Subtraction

To subtract one number from another:

```
?- X is 10 - 4.
```

This will result in:

X = 6.

Multiplication

To multiply two numbers:

```
?- X is 6 * 7.
```

This will result in:

```
 X = 42.
```

Division

To divide two numbers (resulting in a floating-point number):

```
?- X is 10 / 4.
```

This will result in:

```
 X = 2.5.
```

Integer Division

To perform integer division (discarding any fractional part):

```
?- X is 10 // 4.
```

This will result in:

```
 X = 2.
```

Modulus

To get the remainder of a division:

```
?- X is 10 mod 4.
```

This will result in:

```
  X = 2.
```

Defining Rules with Arithmetic Operations

You can define rules that include arithmetic operations to create more complex calculations.

Example: Calculate the Area of a Rectangle

```
area_rectangle(Width, Height, Area) :-
    Area is Width * Height.
```

Here, `area_rectangle/3` is a rule that calculates the area of a rectangle given its width and height.

To use this rule:

```
 ?- area_rectangle(5, 10, Area).
```

This will result in:

Area = 50.

Example: Find the Sum of Two Numbers

```
sum(A, B, Result) :-
    Result is A + B.
```

This rule computes the sum of `A` and `B` and stores it in `Result`.

To use this rule:

```
 ?- sum(3, 7, X).
```

- This will result in:

X = 10.

Important Considerations

`is` operator: It must be used whenever you want to perform and retrieve the result of an arithmetic calculation.

Arithmetic terms: Prolog does not automatically evaluate terms like `5 + 3`. For example:

```
 ?- X = 5 + 3.
```

This query will not evaluate `5 + 3`; it will unify `X` with the term `5 + 3` itself. To get the numeric result, you must use:

```
 ?- X is 5 + 3.
```

Variables must be instantiated: When using `is`, the expression must be fully instantiated before evaluation. For instance:

```
?- X is A + 3.
```

This will result in an error if `A` is not already bound to a value.

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

Step-by-Step Procedure to Implement Arithmetic Operations in Prolog

1. **Set Up Prolog Environment**:
   - Ensure that **GNU Prolog** or another Prolog interpreter is installed on your system.
   - Open the terminal or command prompt to interact with the Prolog interpreter.
2. **Create a Prolog Source File**:
   - Open a text editor.
   - Create a new file with a .pl extension (e.g., arithmetic.pl).
   - Write the logic for arithmetic operations using predicates in the file.
3. **Define Predicates for Operations**:
   - In the Prolog file, write the rules and facts needed to perform calculations (e.g., sum, difference, product).
   - Use the is operator within rules to perform calculations.
4. **Save the Prolog File**:
   - Save the file after writing the predicates and rules.
5. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.
   - Navigate to the directory where the .pl file is saved using the cd command.
6. **Load the Prolog Program**:
   - Start the Prolog interpreter by typing gprolog or the equivalent command for your Prolog installation.
   - Load the Prolog source file into the interpreter using:
     ```
     | ?- consult('arithmetic.pl').
     ```
   - Alternatively, use the shorthand:
     ```
     | ?- ['arithmetic.pl'].
     ```
7. **Query the Arithmetic Predicates**:
   - After loading the file, test the arithmetic rules using queries.
   - At the Prolog prompt, enter queries to perform addition, subtraction, multiplication, or division.
   - Prolog will evaluate the expressions using the is operator and display the results.
8. **Interpret the Results**:
   - Prolog will display the output for each query.
   - If there are multiple possible solutions (e.g., for finding values satisfying a rule), use ; to see more

results.

9. **Debugging and Modifying Rules**:
    - If any errors or unexpected results appear, edit the .pl file, correct the logic, and reload the file in the interpreter.
    - Repeat the process until the rules perform as expected.
10. **Exit the Prolog Interpreter**:
    - Once done with testing, exit the Prolog interpreter using the halt command:

```
| ?- halt.
```

- **Code:**

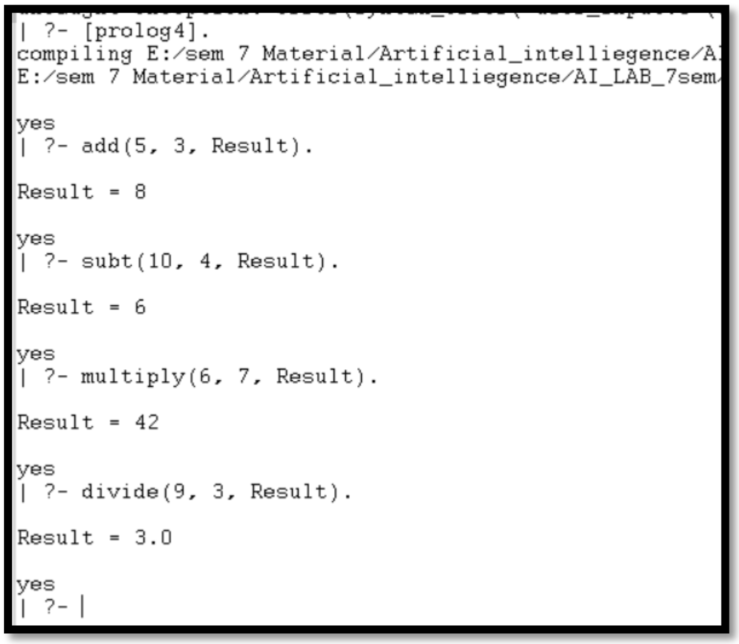
Facts:

```
add(X, Y, Result) :- Result is X + Y.
subt(X, Y, Result) :- Result is X - Y.
multiply(X, Y, Result) :- Result is X * Y.
divide(X, Y, Result) :- Result is X / Y.
```

Queries:

```
?- add(5, 3, Result).
?- subt(10, 4, Result).
?- multiply(6, 7, Result).
?- divide(9, 3, Result).
```

- **Output:**



```
| ?- [prolog4].
compiling E:/sem 7 Material/Artificial_intelliegence/A
E:/sem 7 Material/Artificial_intelliegence/AI_LAB_7sem

yes
| ?- add(5, 3, Result).

Result = 8

yes
| ?- subt(10, 4, Result).

Result = 6

yes
| ?- multiply(6, 7, Result).

Result = 42

yes
| ?- divide(9, 3, Result).

Result = 3.0

yes
| ?- |
```

- **Signature:**

Practical - 5

Aim: Write a PROLOG program to display the numbers from 1 to 10 by simulating the loop using recursion.

- **Objective: :** to do recursion operation using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

In **Prolog**, recursion is a powerful technique used for defining rules that perform iterative tasks, such as displaying numbers from **1 to 10**. Understanding how recursion works in Prolog is crucial because it allows you to define a rule that repeatedly calls itself until a certain condition is met, making it possible to work through lists, generate sequences, or handle complex data structures.

Recursion in Prolog: Overview

- **Recursion** in Prolog is the process where a rule calls itself directly or indirectly.
- It consists of two essential components:
    - **Base case**: A condition that stops the recursion to avoid infinite loops.
    - **Recursive case**: A rule that reduces the problem size and calls itself.

Recursion for Displaying Numbers

When you want to display numbers from **1 to 10** using recursion in Prolog, you define a recursive rule that starts at the first number and increments until it reaches the limit. The goal is to print numbers in a sequence from **1 to 10**, which Prolog achieves through recursion.

Structure of the Recursive Rule

The typical structure of a recursive rule for displaying a sequence of numbers follows this form:

1. **Base Case**: Defines when to stop the recursion.
2. **Recursive Case**: Describes how to move towards the base case by making a recursive call.

Theory Explanation

To print numbers **1 to 10**, you need to define a rule that:

- Prints the current number.
- Recursively calls itself with the next number.
- Stops when it reaches **10**.

Base Case

The **base case** in Prolog defines the condition under which recursion should stop. For displaying numbers from **1 to 10**, the base case can be defined as:

- "If the current number is greater than **10**, stop the recursion."

This prevents the rule from making further recursive calls and thus avoids infinite looping.

Recursive Case

The **recursive case** handles the actual task of:

- Displaying the current number.
- Incrementing the current number.
- Calling the rule again with the next number.

In Prolog:

- The rule displays the number using a built-in predicate (like write/1).
- Then, it increments the number and calls itself recursively with the incremented value.

Execution Flow of Recursion

1. The rule is called with the starting value (**1**).
2. It prints **1**, increments to **2**, and calls itself with **2**.
3. This process continues with each recursive call, printing the current number and calling itself with the next incremented number.
4. When the current number becomes **11**, the base case is met, and the recursion stops.

- ## Tools / Material Needed:
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- ## Procure:

Step-by-Step Procedure to Display Numbers 1 to 10 using Recursion in Prolog

1. **Set Up Prolog Environment**:
   - Ensure that **GNU Prolog** or another Prolog interpreter is installed on your system.
   - Open the terminal or command prompt for interacting with the Prolog interpreter.
2. **Create a Prolog Source File**:
   - Open a text editor of your choice.
   - Create a new file with a .pl extension, for example, print_numbers.pl.
   - Write the logic for printing numbers recursively in the file.
3. **Define the Recursive Rule**:
   - Write a rule that includes:
     - A **base case** that stops the recursion when the current number exceeds **10**.
     - A **recursive case** that displays the current number and then calls itself with the incremented number.
4. **Save the Prolog File**:
   - After writing the rule for displaying numbers, save the file.
5. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.
   - Navigate to the directory where the .pl file is saved using the cd command.
6. **Load the Prolog Program**:
   - Start the Prolog interpreter by typing gprolog or the equivalent command for your Prolog installation.
   - Load the Prolog source file into the interpreter using:
     ```
     | ?- consult('print_numbers.pl').
     ```
   - Alternatively, use the shorthand:
     ```
     | ?- ['print_numbers.pl'].
     ```
7. **Run the Recursive Rule**:

- After loading the file, call the recursive rule from the Prolog prompt:

  ```
  | ?- print_numbers(1).
  ```

- This initiates the recursive process starting from **1**.

8. **Observe the Output**:
   - Prolog will print the numbers **1 to 10** in sequence as the recursive rule is executed.
   - The rule will stop when it reaches the base case (i.e., when the current number is greater than **10**).
9. **Debugging and Adjustments**:
   - If the output is not as expected, check for logic or syntax errors in the .pl file.
   - Edit the file as needed, save, and reload it into the interpreter.
10. **Exit the Prolog Interpreter**:
    - Once done, you can exit the Prolog interpreter using:

      ```
      | ?- halt.
      ```

- **Code:**

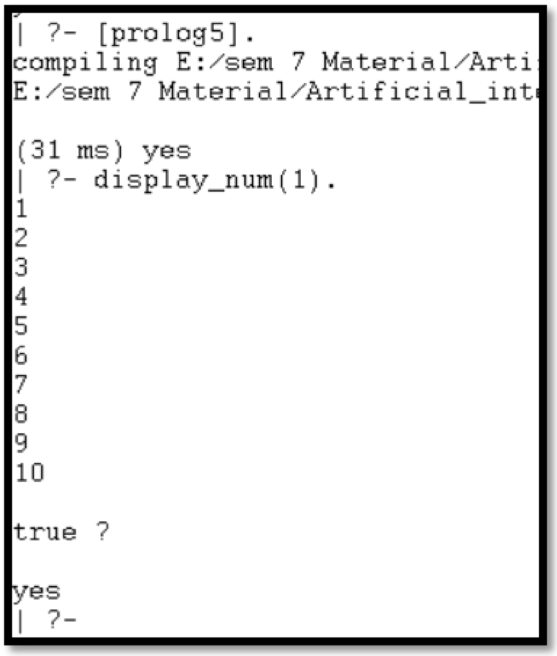
Facts:

```
display_num(10) :- write(10), nl.
display_num(N) :- N < 10, write(N), nl, N1 is N + 1, display_num(N1).
```

Queries:

```
display_num(1).
```

- **Output:**



```
| ?- [prolog5].
compiling E:/sem 7 Material/Arti
E:/sem 7 Material/Artificial_int

(31 ms) yes
| ?- display_num(1).
1
2
3
4
5
6
7
8
9
10

true ?

yes
| ?-
```

- **Signature:**

Practical - 6

Aim: Write a PROLOG program to count number of elements in a list.

- **Objective:** to do count operation on list using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

In **Prolog**, counting the number of elements in a list is a common task that can be accomplished using **recursion**. Lists in Prolog are fundamental data structures represented using square brackets, such as [a, b, c], where each element is separated by a comma. To count the elements, Prolog uses a recursive approach, breaking down the list into smaller parts until a base condition is met.

Understanding Recursion for Counting Elements in Prolog

Basic Concept of a List

- A list in Prolog is either:
    - An **empty list** [].
    - A **non-empty list** [Head | Tail], where Head is the first element, and Tail is the list of remaining elements.
- Using this representation, we can recursively traverse a list.

Components of a Recursive Rule

To count the number of elements in a list using recursion, a rule generally consists of:

1. **Base Case**: Defines when the recursion should stop, which is when the list becomes empty.
2. **Recursive Case**: Describes how to decompose the problem and move towards the base case, counting the head of the list and recursively counting the elements in the tail.

Explanation of the Approach

Base Case

- The **base case** occurs when the list is empty [].
- When the list is empty, it contains **0 elements**, so we return **0**.
- This stops the recursion because there are no more elements to count.

Recursive Case

- The **recursive case** handles the situation where the list is non-empty [Head | Tail]:
    - Head represents the first element (which we can count as **1**).
    - Tail is the rest of the list.
- To count the elements, we:
    - Count **1** for the Head.
    - Recursively call the same rule to count the elements in Tail.
    - Add **1** (for the current head) to the result of counting elements in Tail.

Process Flow

1. **Start with a list**: For example, [a, b, c].
2. **Break down the list**: [a | [b, c]] means Head = a and Tail = [b, c].
3. **Count the head**: Count **1** for a.

4. **Recurse with the tail**: Apply the rule to [b, c].
5. **Continue until reaching an empty list**:
   - [b | [c]] counts **1** for b and recurses with [c].
   - [c | []] counts **1** for c and recurses with [].
   - [] is the base case and returns **0**.
6. **Combine counts**: Sum the counts returned from each recursive call to get the total.

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

Step-by-Step Procedure to Count Elements in a List Using Recursion in Prolog

1. **Set Up Prolog Environment**:
   - Ensure that **GNU Prolog** or another Prolog interpreter is installed on your system.
   - Open the terminal or command prompt to interact with the Prolog interpreter.
2. **Create a Prolog Source File**:
   - Open a text editor of your choice.
   - Create a new file with a .pl extension, for example, count_elements.pl.
   - Write the logic for counting elements in a list recursively in the file.
3. **Define the Recursive Rule**:
   - Write a rule that includes:
     - A **base case** for when the list is empty ([]), returning a count of **0**.
     - A **recursive case** for when the list is not empty, which:
       - Counts the first element (Head).
       - Recursively counts the elements of the Tail.
       - Adds **1** for the current element to the count of elements in the Tail.
4. **Save the Prolog File**:
   - After writing the rule for counting elements, save the file.
5. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.
   - Navigate to the directory where the .pl file is saved using the cd command.
6. **Load the Prolog Program**:
   - Start the Prolog interpreter by typing gprolog or the equivalent command for your Prolog installation.
   - Load the Prolog source file into the interpreter using:

     ```
     | ?- consult('count_elements.pl').
     ```
   - Alternatively, use the shorthand:

     ```
     | ?- ['count_elements.pl'].
     ```
7. **Run the Recursive Rule**:

- After loading the file, call the recursive rule from the Prolog prompt to count elements in a list:

  ```
  | ?- count_elements([a, b, c], Count).
  ```

- Here, count_elements/2 is the rule, [a, b, c] is the list, and Count is the variable where the number of elements will be stored.

8. **Observe the Output**:
   - Prolog will calculate the count of elements in the list by using the recursive rule and display the result:

     ```
     Count = 3.
     ```

9. **Debugging and Adjustments**:
   - If the output is not as expected, check for any logic or syntax errors in the .pl file.
   - Edit the file as needed, save, and reload it into the interpreter.
10. **Exit the Prolog Interpreter**:
    - Once done, you can exit the Prolog interpreter using:

      ```
      | ?- halt.
      ```

- **Code:**

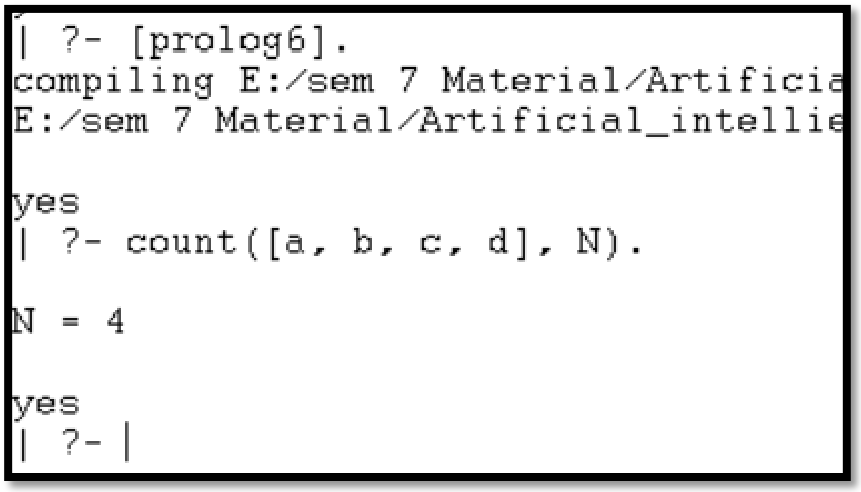
Facts:

```
count([], 0).
count([_|T], N) :- count(T, N1), N is N1 + 1.
```

Queries:

```
count([a, b, c, d], N).
```

- **Output:**



```
| ?- [prolog6].
compiling E:/sem 7 Material/Artificia
E:/sem 7 Material/Artificial_intellie

yes
| ?- count([a, b, c, d], N).

N = 4

yes
| ?- 
```

- **Signature:**

Practical - 7

Aim: Write a PROLOG program to perform following operations on a list:

a. To insert an element into the list.

b. To replace an element from the list.

c. To delete an element from the list.

- **Objective:** to do crud operation on list using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

  Prolog is well-suited for operations on lists due to its recursive nature and the way lists are represented. Lists in Prolog can store a sequence of elements, and they are a fundamental data structure in the language. This makes Prolog a powerful tool for performing various operations like adding, removing, concatenating, or reversing lists.

  Overview of Lists in Prolog

  - A **list** in Prolog is either:
    - An **empty list**, represented as [].
    - A **non-empty list**, represented using the **head** and **tail** notation: [Head | Tail], where:
      - Head represents the first element of the list.
      - Tail is itself a list containing the remaining elements.

  Common List Operations in Prolog

  Here are some of the typical operations that can be performed on lists using Prolog, along with their theoretical explanation:

  1. Checking for Membership (Element Existence)

  - **Purpose**: To determine if a particular element exists in a list.
  - **Theory**:
    - Use recursion to traverse through the list.
    - Base Case: The element is found as the Head of the list.
    - Recursive Case: If the element is not the Head, check for its existence in the Tail until the list is empty.
    - This approach is analogous to iterating through each element of the list until a match is found or all elements have been checked.

  2. Concatenating Two Lists

  - **Purpose**: To join two lists into a single list.
  - **Theory**:
    - Use recursion to append elements from the first list to the second.
    - Base Case: If the first list is empty ([]), the result is just the second list.
    - Recursive Case: Take the Head of the first list and prepend it to the result of concatenating the Tail of the first list with the second list.
    - This recursively builds a new list with the elements of the first list followed by those of the second.

  3. Reversing a List

  - **Purpose**: To reverse the order of elements in a list.
  - **Theory**:

- Use recursion to process each element of the list, adding it to the front of a new list.
- Base Case: Reversing an empty list results in an empty list.
- Recursive Case: Take the Head of the original list and add it to the reversed Tail.
- This results in the elements being arranged in reverse order as the recursion unwinds.

4. Finding the Length of a List

- **Purpose**: To determine the number of elements in a list.
- **Theory**:
    - Use recursion to count each element.
    - Base Case: The length of an empty list ([]) is 0.
    - Recursive Case: Count 1 for the Head and add it to the length of the Tail.
    - This sums up the number of elements as each recursive call processes one element of the list.

5. Inserting an Element at the Beginning or End of a List

- **Purpose**: To add a new element to the start or end of a list.
- **Theory**:
    - To **insert at the beginning**, simply use [Element | List].
    - To **insert at the end**, use recursion:
        - Base Case: If the list is empty ([]), create a new list with the element.
        - Recursive Case: Rebuild the list by taking the Head and appending the result of inserting the element into the Tail.
    - This allows for adding elements while maintaining the order of the existing elements.

6. Deleting an Element from a List

- **Purpose**: To remove a specific element from a list.
- **Theory**:
    - Use recursion to check each element.
    - Base Case: If the list is empty, there is nothing to delete.
    - Recursive Case:
        - If the Head matches the element to be deleted, skip it and continue with the Tail.
        - If the Head does not match, include it in the result and continue deleting the element from the Tail.
    - This approach ensures that only the specified element is removed while preserving other elements in order.

How Prolog Handles Lists Recursively

- **Recursive Patterns**: Prolog excels in processing lists because it naturally handles head-tail decomposition using recursion. Each list operation generally involves breaking the list down into a Head and Tail and then applying a rule recursively on the Tail.
- **Unification**: Prolog's pattern-matching and unification mechanism simplifies list operations. It allows elements to be matched directly to the Head and the remaining part to the Tail, making it easier to define recursive rules.
- **No Loops**: Unlike imperative languages, Prolog doesn't use loops for list processing. Instead, recursion is the primary means to iterate over lists.

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

  Step-by-Step Procedure to Insert, Replace, and Delete Elements in a List Using Prolog

  1. **Set Up Prolog Environment**:
     - Make sure that **GNU Prolog** or another Prolog interpreter is installed on your system.
     - Open the terminal or command prompt to interact with the Prolog interpreter.
  2. **Create a Prolog Source File**:
     - Open a text editor of your choice.
     - Create a new file with a .pl extension, for example, list_operations.pl.
     - This file will contain the logic for inserting, replacing, and deleting elements in a list.
  3. **Define the Rule for Inserting an Element**:
     - Define a rule that adds an element at a specific position or at the end of the list.
     - Example rules:
       - Insert at the beginning using [Element | List].
       - Insert at a specific position using recursion until the position is reached.
  4. **Define the Rule for Replacing an Element**:
     - Define a rule that replaces a specific element with another.
     - Example logic:
       - Traverse the list recursively.
       - When the target element is found, replace it with the new element.
       - Continue with the rest of the list unchanged.
  5. **Define the Rule for Deleting an Element**:
     - Define a rule that removes a specific element from the list.
     - Example logic:
       - Recursively traverse the list.
       - When the element matches the target, skip it and continue with the remaining list.
       - If it doesn’t match, keep it and continue with the next element.
  6. **Save the Prolog File**:
     - After writing the rules for insertion, replacement, and deletion, save the file.
  7. **Open the Prolog Interpreter**:
     - Open a terminal or command prompt.
     - Navigate to the directory where the .pl file is saved using the cd command.
  8. **Load the Prolog Program**:
     - Start the Prolog interpreter by typing gprolog or the equivalent command for your Prolog installation.
     - Load the Prolog source file into the interpreter using:

```
| ?- consult('list_operations.pl').
```

- Alternatively, use the shorthand:

```
| ?- ['list_operations.pl'].
```

9. **Run the Rules for Insertion, Replacement, and Deletion**:
   - Call the rules from the Prolog prompt to perform the operations:
     - To **insert** an element, for example at position 2:

   | ?- insert_element([a, b, c], d, 2, Result).

     - To **replace** an element, for example, replace b with x:

```
| ?- replace_element([a, b, c], b, x, Result).
```

     - To **delete** an element, for example, remove b:

```
| ?- delete_element([a, b, c], b, Result).
```

   - In each case, Result is the output list after performing the operation.
10. **Observe the Output**:
   - Prolog will perform the specified operation (insert, replace, or delete) and display the modified list:

```
Result = [a, d, b, c].
Result = [a, x, c].
Result = [a, c].
```

11. **Debugging and Adjustments**:
   - If the output is not as expected, check for any logic or syntax errors in the .pl file.
   - Edit the file as needed, save, and reload it into the interpreter.
12. **Exit the Prolog Interpreter**:
   - Once done, you can exit the Prolog interpreter using:

```
| ?- halt.
```

- **Code:**

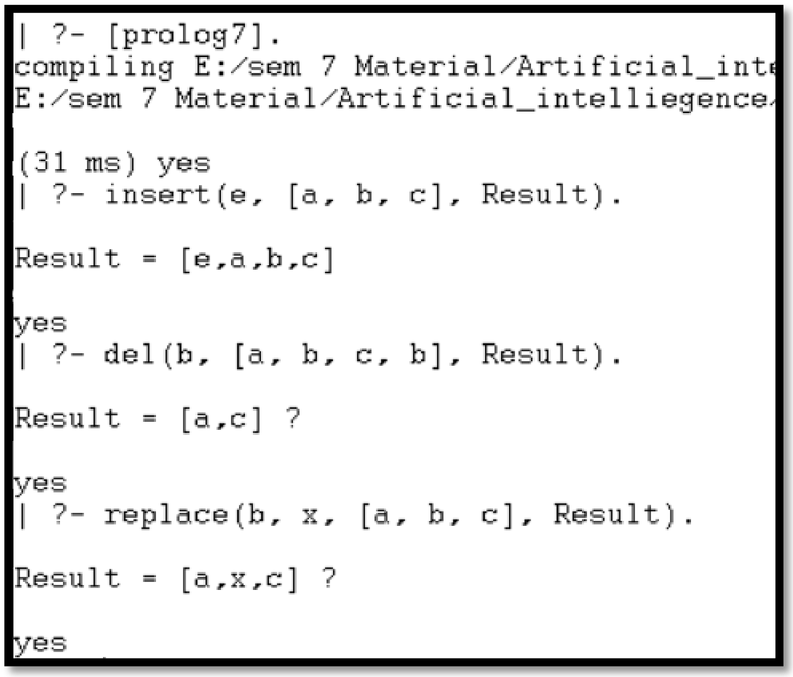
Facts:

```
insert(Element, List, [Element|List]).

replace(_, _, [], []).
replace(Old, New, [Old|T], [New|T1]) :- replace(Old, New, T, T1).
replace(Old, New, [H|T], [H|T1]) :- replace(Old, New, T, T1).

%del(X, List, Result)
del(_, [], []).
del(X, [X|T], Result) :- del(X, T, Result).
del(X, [H|T], [H|T1]) :- H \= X,  del(X, T, T1).
```

Queries:

```
insert(e, [a, b, c], Result).
replace(b, x, [a, b, c], Result).
delete(b, [a, b, c, b], Result).
```

- **Output:**



```
| ?- [prolog7].
compiling E:/sem 7 Material/Artificial_int
E:/sem 7 Material/Artificial_intelliegence

(31 ms) yes
| ?- insert(e, [a, b, c], Result).

Result = [e,a,b,c]

yes
| ?- del(b, [a, b, c, b], Result).

Result = [a,c] ?

yes
| ?- replace(b, x, [a, b, c], Result).

Result = [a,x,c] ?

yes
```

- **Signature:**

Practical - 8

Aim: Write a PROLOG program to reverse the list.

- **Objective:** to reverse a list using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

Reversing a list in **Prolog** is a common task that involves rearranging the elements of a list such that the last element becomes the first, the second-to-last becomes the second, and so on, until the entire list is reversed. Prolog's ability to process lists recursively makes it well-suited for defining such operations.

Understanding List Reversal in Prolog

List Representation in Prolog

- A **list** in Prolog is represented either as:
  - An **empty list**: [].
  - A **non-empty list**: [Head | Tail], where:
    - Head represents the first element.
    - Tail is a list containing the remaining elements.

Reversing a List using Recursion

The process of reversing a list in Prolog is typically done using **recursion**. Recursion allows Prolog to break down a problem into smaller sub-problems, solving each step until a base condition is met. The structure for reversing a list involves a **base case** and a **recursive case**:

1. Base Case

- The **base case** for reversing a list is when the list is empty ([]).
- The reverse of an empty list is simply an empty list:

```
reverse([], []).
```

- This stops the recursion, as there are no more elements to process.

2. Recursive Case

- The **recursive case** handles the situation when the list is not empty.
- The strategy is to:
  - Take the **Head** of the list and keep it aside.
  - Recursively reverse the **Tail** of the list.
  - After reversing the **Tail**, **append** the **Head** to the end of the reversed list.
- This is generally expressed as:

```
reverse([Head | Tail], ReversedList) :-
    reverse(Tail, ReversedTail),
    append(ReversedTail, [Head], ReversedList).
```

- This rule says:
  - Reverse the Tail of the list.
  - Use append/3 to add the Head to the end of the reversed Tail.
  - The result is the fully reversed list.

How the Reversal Process Works

1. **Example**: To reverse [a, b, c]:
    - Call reverse([a, b, c], Result).
    - Break down the list:
        - Head is a, Tail is [b, c].
    - Recursively reverse [b, c]:
        - Head is b, Tail is [c].
    - Recursively reverse [c]:
        - Head is c, Tail is [].
        - The reverse of [] is [], so append([], [c]) results in [c].
        - Moving up a step, append([c], [b]) results in [c, b].
        - Finally, append([c, b], [a]) results in [c, b, a].
2. **Process Visualization**:
    - Reverse [a, b, c]:
        - reverse([a, b, c], Result) calls reverse([b, c], ReversedTail).
        - reverse([b, c], ReversedTail) calls reverse([c], ReversedTail).
        - reverse([c], ReversedTail) calls reverse([], ReversedTail).
        - reverse([], []) returns an empty list.
        - append([], [c]) gives [c].
        - append([c], [b]) gives [c, b].
        - append([c, b], [a]) gives [c, b, a].

Alternative Approach: Using an Accumulator

An alternative and more **efficient** way to reverse a list in Prolog is by using an **accumulator** to avoid repeated append operations:

- Define a helper predicate that takes an accumulator as an extra argument.
- Start with an empty accumulator, and as you traverse the list, keep adding the Head to the front of the accumulator.
- When the original list is empty, the accumulator contains the reversed list.

Why Use Recursion for List Reversal?

- **Natural Fit**: Prolog's recursive nature makes it suitable for problems like list processing, where each step deals with a smaller sub-problem.
- **Pattern Matching**: Using the [Head | Tail] pattern allows Prolog to easily deconstruct and reconstruct lists during recursion.
- **Recursive Decomposition**: By breaking the list into Head and Tail, we can solve the problem for smaller lists and build up the solution step by step.

- **Tools / Material Needed:**
    - **Hardware:** Laptop / Computer
    - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

    Step-by-Step Procedure to Reverse a List Using Prolog

1. **Set Up Prolog Environment**:
   - Make sure **GNU Prolog** or any other Prolog interpreter is installed on your system.
   - Open the terminal or command prompt to interact with the Prolog interpreter.
2. **Create a Prolog Source File**:
   - Open a text editor (e.g., Notepad, VS Code).
   - Create a new file with a .pl extension, such as reverse_list.pl.
   - This file will contain the logic to reverse a list using Prolog.
3. **Define the Base Case for Reversing a List**:
   - The base case should handle an empty list, which returns an empty list when reversed.
   - This stops the recursion when the list is empty.
4. **Define the Recursive Case for Reversing a List**:
   - In the recursive case:
     - Take the **Head** of the list and reverse the **Tail** recursively.
     - Use append/3 to add the Head to the end of the reversed Tail.
   - Example logic for the recursive rule:

```
reverse([], []).  % Base case: an empty list is reversed to an empty
list.
reverse([Head | Tail], ReversedList) :-
    reverse(Tail, ReversedTail),
    append(ReversedTail, [Head], ReversedList).
```

5. **Save the Prolog File**:
   - After writing the rules for reversing the list, save the file.
6. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.
   - Navigate to the directory where the .pl file is saved using the cd command.
7. **Load the Prolog Program**:
   - Start the Prolog interpreter by typing gprolog or the equivalent command for your Prolog installation.
   - Load the Prolog source file into the interpreter using:

```
| ?- consult('reverse_list.pl').
```

   - Alternatively, use the shorthand:

```
| ?- ['reverse_list.pl'].
```

8. **Run the Reverse List Rule**:
   - Call the rule from the Prolog prompt to reverse a list:

```
| ?- reverse([1, 2, 3, 4], Result).
```

   - Result is the output list after being reversed.
9. **Observe the Output**:
   - Prolog will compute the reversed list and display the result:

```
Result = [4, 3, 2, 1].
```

   - This means that [1, 2, 3, 4] has been successfully reversed.

10. **Debugging and Adjustments**:
    - If the output is not as expected, check for any logic or syntax errors in the .pl file.
    - Edit the file as needed, save, and reload it into the interpreter.
11. **Exit the Prolog Interpreter**:
    - Once done, you can exit the Prolog interpreter using:

```
| ?- halt.
```

- **Code:**

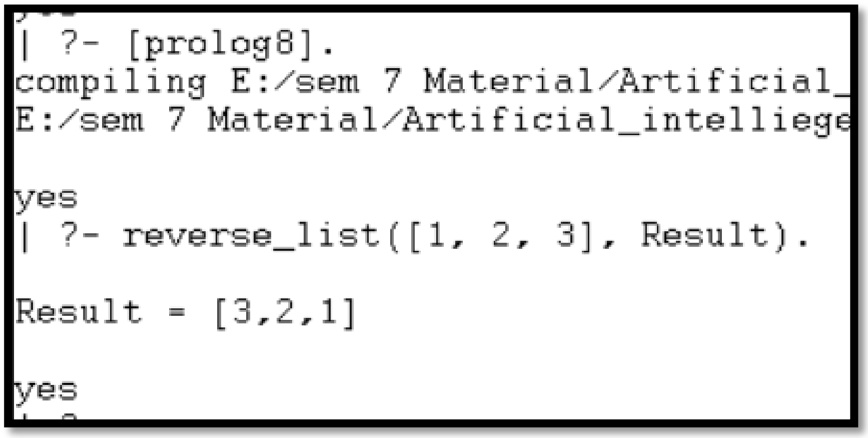
Facts:

```
% reverse_list(Input, Reversed)
reverse_list([], []).
reverse_list([H|T], Reversed) :-
    reverse_list(T, RevT),
       append(RevT, [H], Reversed).
```

Queries:

```
reverse_list([1, 2, 3], Result).
```

- **Output:**



```
| ?- [prolog8].
compiling E:/sem 7 Material/Artificial_
E:/sem 7 Material/Artificial_intelliege

yes
| ?- reverse_list([1, 2, 3], Result).

Result = [3,2,1]

yes
```

- **Signature:**

Practical - 9

Aim: Write a PROLOG program to demonstrate the use of CUT and FAIL predicate.

- **Objective:** to demonstrate CUT and FAIL predicate using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

The **cut (!)** and **fail** predicates in Prolog are special control predicates that help manage the flow of logic in a program. They allow us to control backtracking, influence the search space, and define conditions that must or must not be met. Understanding these predicates is essential for optimizing Prolog programs and enforcing specific behaviours during the execution of rules.

Understanding the CUT (!) Predicate in Prolog

The **cut** (!) is a control operator in Prolog that is used to **prevent backtracking** beyond a certain point in a rule. It allows Prolog to commit to a particular choice, discarding other possible alternatives that might have been tried if backtracking were to occur.

Key Points About CUT (!):

- **Purpose**: The cut is used to stop Prolog from backtracking to earlier choice points.
- **Effect**: When Prolog encounters a ! in a rule, it commits to all decisions made up to that point.
- **Usage**: cut is often used when only one solution is needed, or when we want to specify that a particular condition should be met without reconsidering other alternatives.

Example of CUT:

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

- **Explanation**:
  - In this example, max/3 finds the maximum of two numbers X and Y.
  - The rule max(X, Y, X) :- X >= Y, !. states that if X is greater than or equal to Y, then X is the result, and the cut (!) prevents any further evaluation.
  - If X is not greater than or equal to Y, then the second rule max(X, Y, Y) is considered.
- **Effect**: When ! is encountered, Prolog will not try other rules even if they could apply, thus optimizing the rule evaluation.

Understanding the FAIL Predicate in Prolog

The **fail** predicate is used to force a goal to fail. When Prolog encounters fail, it immediately stops and backtracks to try other alternatives, if any exist.

Key Points About FAIL:

- **Purpose**: fail is used to explicitly indicate that a rule should not succeed under any circumstances.
- **Effect**: When a fail is encountered, the current rule or goal fails, and Prolog tries the next alternative, if available.
- **Usage**: fail is often used in combination with cut to create **negation** (not) or to iterate through all possibilities without success.

Example of FAIL:

```
print_and_fail(X) :-
    write(X), nl, fail.
```

- **Explanation**:
  - This rule will print X but will **always fail** because of the fail predicate.
  - Even though write(X) is executed and displays the output, fail causes the entire predicate print_and_fail/1 to backtrack and try other rules, if any.
- **Effect**: This can be useful for debugging or for performing actions like printing all possible values without succeeding.

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

Step-by-Step Procedure for Using CUT and FAIL in Prolog

1. **Set Up Prolog Environment**:
   - Ensure that **GNU Prolog** or another Prolog interpreter is installed on your system.
   - Open the terminal or command prompt to interact with the Prolog interpreter.
2. **Create a Prolog Source File**:
   - Open a text editor of your choice.
   - Create a new file with a .pl extension, such as cut_fail_example.pl.
   - This file will contain the rules that use cut (!) and fail.
3. **Define Rules Using CUT (!)**:
   - Define a rule that uses cut to control backtracking.
   - For example, write a rule that determines the maximum of two numbers using cut:
     ```
     max(X, Y, X) :- X >= Y, !.
     max(X, Y, Y).
     ```
   - Explanation:
     - The ! ensures that if X is greater than or equal to Y, Prolog will not look for other alternatives, skipping the second rule.
4. **Define Rules Using FAIL**:
   - Define a rule that always fails after performing some action.
   - For example, create a rule that prints an element but then fails:
     ```
     print_and_fail(X) :-
         write(X), nl, fail.
     ```
   - Explanation:
     - This rule will print X, but due to fail, it will backtrack and never succeed.
5. **Define Rules Using CUT and FAIL Together**:
   - Write a rule that demonstrates the combination of cut and fail.
   - For example, define a rule that checks if an element is **not** in a list:
     ```
     not_member(_, []) :- !, fail.
     not_member(X, [X | _]) :- !, fail.
     ```

```
not_member(X, [_ | Tail]) :- not_member(X, Tail).
```

- Explanation:
  - This rule uses ! to stop further checks if the condition is met, and fail to indicate that the element is found, hence not satisfying not_member.

6. **Save the Prolog File**:
   - Save the file after writing the rules for cut and fail.
7. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.
   - Navigate to the directory where the .pl file is saved using the cd command.
8. **Load the Prolog Program**:
   - Start the Prolog interpreter by typing gprolog or the equivalent command for your Prolog installation.
   - Load the Prolog source file into the interpreter using:

     ```
     | ?- consult('cut_fail_example.pl').
     ```
   - Alternatively, use the shorthand:

     ```
     | ?- ['cut_fail_example.pl'].
     ```
9. **Run the Rules and Observe the Behavior**:
   - To test the cut behavior in max/3:

     ```
     | ?- max(3, 2, Result).
     ```
     - Prolog should return Result = 3 without checking other alternatives.
   - To test the fail behavior:

     ```
     | ?- print_and_fail('Hello').
     ```
     - Prolog will print Hello but will fail and not return true.
   - To test the combination of cut and fail:

     ```
     | ?- not_member(3, [1, 2, 3]).
     ```
     - Prolog will return false because 3 is found in the list.
     - If you try:

     ```
     | ?- not_member(4, [1, 2, 3]).
     ```
       - Prolog will return true, indicating 4 is not in the list.
10. **Debugging and Adjustments**:
    - If the output is not as expected, check for any logic or syntax errors in the .pl file.
    - Edit the file as needed, save, and reload it into the interpreter.
11. **Exit the Prolog Interpreter**:
    - Once done, you can exit the Prolog interpreter using:

      ```
      | ?- halt.
      ```

- **Code:**

Facts:

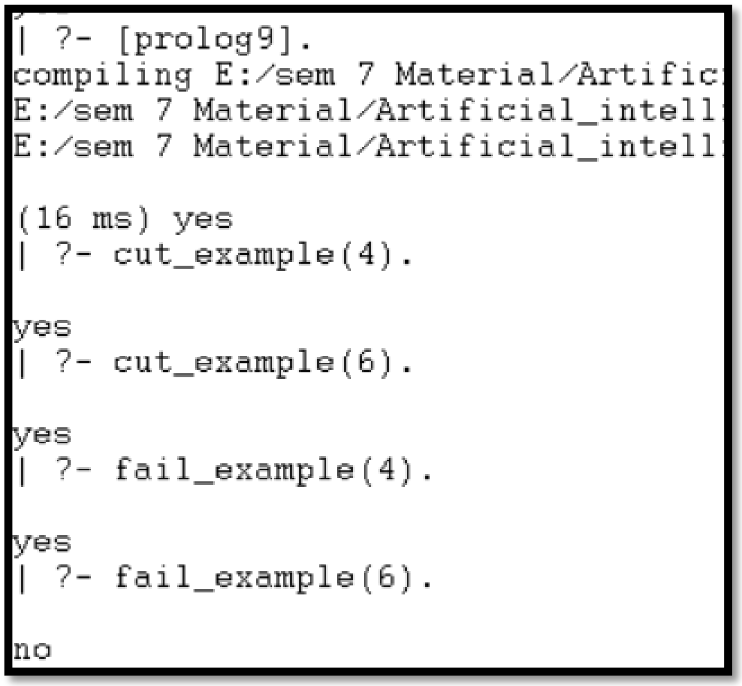
```
cut_example(X) :- X > 5, !.
```

```
cut_example(X) :- X =< 5.

fail_example(X) :- X > 5, !, fail.
fail_example(X).
```

Queries:

```
?- cut_example(4).
?- cut_example(6).
?- fail_example(4).
?- fail_example(6).
```

- **Output:**



```
| ?- [prolog9].
compiling E:/sem 7 Material/Artific
E:/sem 7 Material/Artificial_intell
E:/sem 7 Material/Artificial_intell

(16 ms) yes
| ?- cut_example(4).

yes
| ?- cut_example(6).

yes
| ?- fail_example(4).

yes
| ?- fail_example(6).

no
```

- **Signature:**

Practical - 10

Aim: Write a PROLOG program to solve the Tower of Hanoi problem.

- **Objective:** to solve Tower of Hanoi problem using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

  The **Tower of Hanoi** is a classic problem in computer science and mathematics, involving recursion. It consists of moving a stack of disks from one peg to another, following a set of rules. **Prolog**, with its inherent recursive capabilities, is well-suited for solving this problem.

  Understanding the Tower of Hanoi Problem

  Problem Description:

  - **Goal**: Move n disks from a **Source** peg to a **Destination** peg using an **Auxiliary** peg.
  - **Rules**:
    1. Only one disk can be moved at a time.
    2. A larger disk cannot be placed on top of a smaller disk.
    3. Only the top disk of a stack can be moved.

  Example:

  - Three pegs are labeled as A (Source), B (Auxiliary), and C (Destination).
  - If there are 3 disks, the problem requires moving all disks from peg A to peg C using peg B.

  Recursive Nature of the Solution:

  The solution to the Tower of Hanoi problem is inherently **recursive** because each move can be broken down into smaller sub-problems. The recursive approach allows you to think of moving n disks as a combination of moving smaller subsets of disks.

  Recursive Strategy:

  1. **Base Case**: If there is only **1 disk**, move it directly from the **Source** to the **Destination**.
  2. **Recursive Case**:
     - To move n disks from Source to Destination:
       1. Move the top n-1 disks from the Source peg to the Auxiliary peg using the Destination peg.
       2. Move the **nth** (largest) disk directly from the Source peg to the Destination peg.
       3. Move the n-1 disks from the Auxiliary peg to the Destination peg using the Source peg as a helper.

  Breakdown of Recursive Solution:

  - For n = 3 disks:
    - Move 2 disks from A to B using C.
    - Move the 3rd disk from A to C.
    - Move 2 disks from B to C using A.

  Solving Tower of Hanoi Using Prolog

  Prolog is a logic programming language, and solving the Tower of Hanoi in Prolog involves defining **rules** and **facts** that represent the moves between the pegs.

  How the Solution Works in Prolog:

1. **Base Case**:
    - Define the rule for moving a **single disk** directly from one peg to another.
    - This is the simplest operation.
    - Example rule: hanoi(1, Source, Destination, _) means moving 1 disk from Source to Destination.
2. **Recursive Case**:
    - Define the rule for moving n disks by breaking it down into smaller steps:
        - Move n-1 disks from Source to Auxiliary.
        - Move the nth disk from Source to Destination.
        - Move n-1 disks from Auxiliary to Destination.
3. **Output the Moves**:
    - Use Prolog's write or display functionality to print the moves at each step.
    - This helps visualize the solution.

- **Tools / Material Needed:**
    - **Hardware:** Laptop / Computer
    - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

Step-by-Step Procedure for Solving Tower of Hanoi in Prolog

1. **Set Up Prolog Environment**:
    - Make sure **GNU Prolog** or any other Prolog interpreter is installed on your system.
    - Open a terminal or command prompt to access the Prolog interpreter.
2. **Create a Prolog Source File**:
    - Open a text editor of your choice.
    - Create a new file with a .pl extension, such as hanoi.pl.
    - This file will contain the Prolog rules for solving the Tower of Hanoi problem.
3. **Define the Base Case**:
    - Write the rule for moving a single disk directly from one peg to another.
    - This is the simplest operation in the Tower of Hanoi problem.
    - Example rule:

```
hanoi(1, Source, Destination, _) :-
    write('Move disk from '), write(Source),
    write(' to '), write(Destination), nl.
```

    - **Explanation**: The rule hanoi(1, Source, Destination, _) specifies that if there is only 1 disk, it can be directly moved from Source to Destination.
4. **Define the Recursive Case**:
    - Write a rule for moving N disks from the **Source** peg to the **Destination** peg using an **Auxiliary** peg.
    - The recursive rule should:
        - Move N-1 disks from Source to Auxiliary.

  - Move the Nth disk from Source to Destination.
  - Move N-1 disks from Auxiliary to Destination.
- Example rule:

```
hanoi(N, Source, Destination, Auxiliary) :-
    N > 1,
    M is N - 1,
    hanoi(M, Source, Auxiliary, Destination),
    hanoi(1, Source, Destination, _),
    hanoi(M, Auxiliary, Destination, Source).
```

- **Explanation**: The recursive rule hanoi/4 handles cases where N is greater than 1 and breaks down the problem into smaller moves.

5. **Save the Prolog File**:
   - Save the file after defining the rules for the base and recursive cases.
6. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.
   - Navigate to the directory where the hanoi.pl file is saved using the cd command.
7. **Load the Prolog Program**:
   - Start the Prolog interpreter by typing gprolog or the appropriate command for your Prolog setup.
   - Load the Prolog file using:

```
| ?- consult('hanoi.pl').
```

   - Alternatively, use the shorthand:

```
| ?- ['hanoi.pl'].
```

8. **Run the Program with a Query**:
   - To solve the Tower of Hanoi problem for 3 disks with pegs A, B, and C, run the following query:

```
| ?- hanoi(3, 'A', 'C', 'B').
```

   - **Explanation**: This query instructs Prolog to move 3 disks from peg A (Source) to peg C (Destination) using peg B (Auxiliary).
   - The Prolog interpreter will display each step of the disk movements.
9. **Observe the Output**:
   - Prolog will print each step of moving the disks from one peg to another.
   - The output will follow the sequence required to solve the Tower of Hanoi problem for the given number of disks.
10. **Adjust and Experiment**:
    - Modify the number of disks in the query (e.g., hanoi(4, 'A', 'C', 'B')) to observe how the solution scales.
    - This allows you to explore the recursive nature of the problem and see how Prolog handles the backtracking for larger numbers of disks.
11. **Exit the Prolog Interpreter**:
    - Once you are done with the execution and testing, you can exit the Prolog interpreter using:

```
| ?- halt.
```

- **Code:**

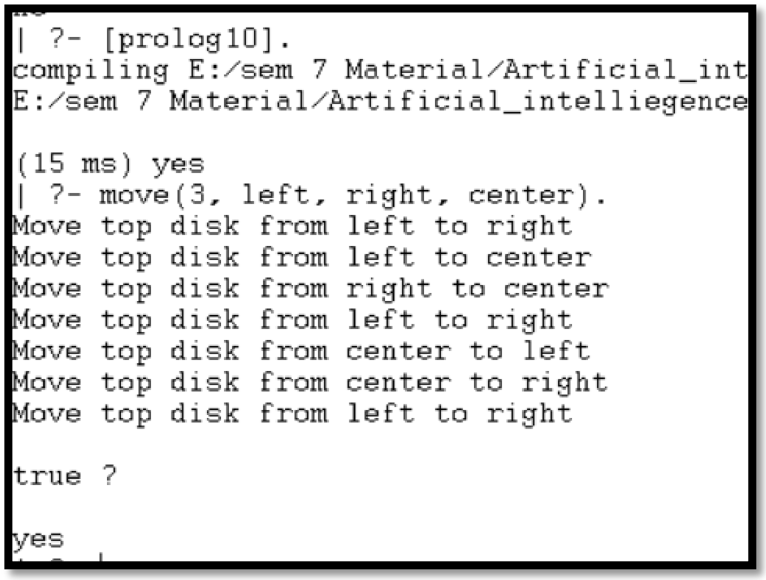
Facts:

```
move(1, From, To, _) :- write('Move top disk from '), write(From), write(' to '),
   write(To), nl.
move(N, From, To, Aux) :-
    N > 1,
    M is N - 1,
    move(M, From, Aux, To),
    move(1, From, To, _),
    move(M, Aux, To, From).
```

Queries:

```
move(3, left, right, center).
```

- **Output:**



```
| ?- [prolog10].
compiling E:/sem 7 Material/Artificial_int
E:/sem 7 Material/Artificial_intelliegence

(15 ms) yes
| ?- move(3, left, right, center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right

true ?

yes
```

- **Signature:**

Practical - 11

Aim: Write a PROLOG program to solve the N-Queens problem.

- **Objective:** to solve N-Queens problem using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

  The **N-Queens Problem** is a classic problem in computer science and artificial intelligence that involves placing N queens on an N x N chessboard such that no two queens threaten each other. This means that no two queens should share the same row, column, or diagonal.

  Problem Statement:

  - **Goal**: Place N queens on an N x N chessboard so that no two queens are in the same row, column, or diagonal.
  - **Constraints**:
    1. A queen cannot share the same **row** as another queen.
    2. A queen cannot share the same **column** as another queen.
    3. A queen cannot share the same **diagonal** (both major and minor diagonals) as another queen.

  Understanding the Problem:

  - The board is represented by a list of positions, where each position corresponds to a column, and the value at each position corresponds to the row where the queen is placed in that column.
  - For example, in a 4-queen problem, a solution could be represented as [2, 4, 1, 3], meaning:
    - Queen in column 1 is placed at row 2.
    - Queen in column 2 is placed at row 4.
    - Queen in column 3 is placed at row 1.
    - Queen in column 4 is placed at row 3.

  Approach to Solve the N-Queens Problem Using Prolog:

  The N-Queens problem is solved using **backtracking**, and Prolog is particularly well-suited for this approach due to its ability to handle recursion and constraints directly.

  Strategy:

  1. **Modeling the Problem**:
     - Represent the board as a list of positions, where each element represents the row position of a queen in a specific column.
     - Use a predicate to ensure that no two queens can attack each other.
  2. **Backtracking**:
     - Recursively try placing queens in each column, ensuring that the placement satisfies the constraints.
     - If placing a queen in a column violates any constraint, backtrack and try a different position.
  3. **Constraint Checking**:
     - Ensure that no two queens are on the same row.
     - Ensure that no two queens are on the same diagonal.
       - For two queens placed at (X1, Y1) and (X2, Y2), they are on the same diagonal if abs(X1 - X2) = abs(Y1 - Y2).

  How Prolog Solves the Problem:

1. **Define the Solution Space**:
   - Generate a list of numbers representing potential row positions for each queen.
   - Use permutation and constraints to find valid placements.
2. **Define Constraints**:
   - Use predicates to ensure that no two queens are on the same row or diagonal.
   - Use Prolog's not/1 or \+ operator for negating conditions when checking for conflicts.
3. **Generate Solutions**:
   - Use Prolog's recursion to place each queen and backtrack when a conflict arises.
   - Output each valid arrangement as a solution.

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

Step-by-Step Procedure for Solving N-Queens in Prolog

1. **Set Up Prolog Environment**:
   - Ensure you have a Prolog interpreter installed (e.g., **GNU Prolog**).
   - Open a terminal or command prompt to access the Prolog interpreter.
2. **Create a Prolog Source File**:
   - Open a text editor and create a new file with a .pl extension, such as n_queens.pl.
   - This file will contain the Prolog code for solving the N-Queens problem.
3. **Define the Main Predicate**:
   - Start by defining the main predicate n_queens/2, which will find a valid arrangement of N queens on an N x N board.
   - Example:

```
n_queens(N, Solution) :-
    length(Solution, N),  % Create a list of length N
    generate_positions(N, Solution),  % Generate possible positions
    safe(Solution).  % Check if the positions are safe
```

4. **Generate Possible Positions**:
   - Define a helper predicate generate_positions/2 that creates a list of integers from 1 to N, representing the row positions for the queens.
   - Example:

```
generate_positions(N, Positions) :-
    numlist(1, N, Positions).  % Generates a list of numbers from 1 to N
```

5. **Check for Safety**:
   - Define the predicate safe/1 to ensure that no two queens can attack each other.
   - This involves checking that no two queens are in the same row or on the same diagonal.
   - Example:

```
safe([]).  % An empty board is safe
safe([Queen|Others]) :-
    safe(Others),
    not(attack(Queen, Others)).  % Check for attacks on Others
```

6. **Define the attack/2 predicate to determine if a queen attacks another:**

```
attack(Q, Others) :-  % Check if Queen Q attacks any queen in Others
    member(O, Others),
    (O = Q;  % Same row
     abs(Q - O) =:= abs(Index - IndexOther)).  % Same diagonal
```

7. **Implement Backtracking**:
   - Prolog's backtracking mechanism allows it to explore different configurations of queens and backtrack if a configuration leads to an attack.
   - The main predicate will naturally utilize this when trying to satisfy the conditions defined.
8. **Save the Prolog File**:
   - Save the n_queens.pl file after defining the predicates.
9. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.
   - Navigate to the directory where the n_queens.pl file is saved using the cd command.
10. **Load the Prolog Program**:
    - Start the Prolog interpreter by typing gprolog or the appropriate command for your Prolog setup.
    - Load the Prolog file using:

      ```
      | ?- consult('n_queens.pl').
      ```
    - Alternatively, use the shorthand:

      ```
      | ?- ['n_queens.pl'].
      ```
11. **Run the Program with a Query**:
    - To find a solution for the N-Queens problem (e.g., for N=4), run the following query:

      ```
      | ?- n_queens(4, Solution).
      ```
    - **Explanation**: This query instructs Prolog to find an arrangement of 4 queens on a 4 x 4 board, storing the solution in the variable Solution.
12. **Observe the Output**:
    - Prolog will print each solution found for the N-Queens problem.
    - You can use the ; operator to find additional solutions if they exist.
13. **Adjust and Experiment**:
    - Modify the value of N in the query (e.g., n_queens(8, Solution)) to find solutions for larger boards.
    - Experiment with different values to see how the backtracking works with varying board sizes.
14. **Exit the Prolog Interpreter**:
    - Once you are done with the execution and testing, you can exit the Prolog interpreter using:

      ```
      | ?- halt.
      ```

- **Code:**

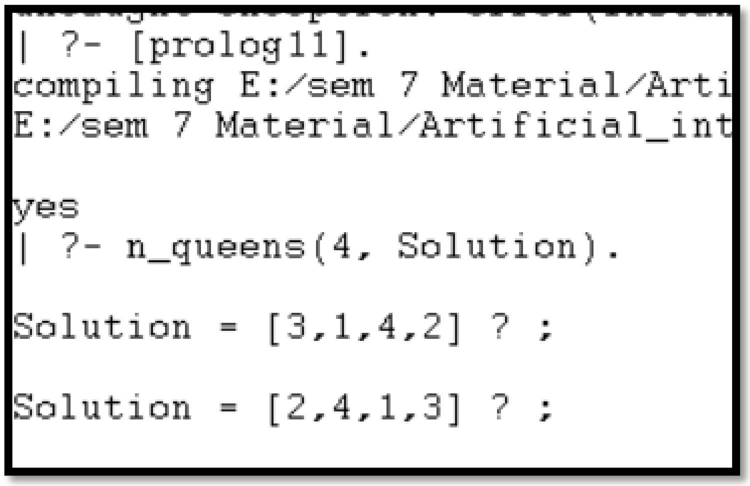
Facts:

```
% n_queens(N, Solution)
n_queens(N, Solution) :-
    length(Solution, N),
    place_queens(Solution, N),
    safe_queens(Solution).

% place_queens(Queens, N)
place_queens([], _).
place_queens([X|Rest], N) :-
    place_queens(Rest, N),
    between(1, N, X),
    \+ member(X, Rest).

% safe_queens(Queens)
safe_queens([]).
safe_queens([Q|Others]) :-
    safe_from(Q, Others, 1),
    safe_queens(Others).

% safe_from(Q, Others, Dist)
safe_from(_, [], _).
safe_from(Q, [Q1|Others], Dist) :-
    Q =\= Q1 + Dist,
    Q =\= Q1 - Dist,
    Dist1 is Dist + 1,
    safe_from(Q, Others, Dist1).
```

Queries:

```
n_queens(4, Solution).
```

- **Output:**



```
| ?- [prolog11].
compiling E:/sem 7 Material/Arti
E:/sem 7 Material/Artificial_int

yes
| ?- n_queens(4, Solution).

Solution = [3,1,4,2] ? ;

Solution = [2,4,1,3] ? ;
```

- **Signature:**

Practical - 12

Aim: Write a PROLOG program to solve the Travelling Salesman problem.

- **Objective:** to solve Travelling Salesman problem using prolog and to get familiar with prolog programming & GNU prolog.
- **Theory:**

  The **Traveling Salesman Problem (TSP)** is a classic optimization problem in computer science and operations research. The objective is to find the shortest possible route that visits a set of cities exactly once and returns to the original city. TSP is a well-known NP-hard problem, which means that there is no known efficient algorithm to solve it in polynomial time.

  Problem Statement:

  - **Input**: A list of cities and the distances between each pair of cities.
  - **Output**: The shortest possible route that visits each city exactly once and returns to the starting city.

  Characteristics of TSP:

  1. **Cities and Distances**:
     - The cities are often represented as nodes in a graph.
     - The distances between cities can be represented as a weighted edge between nodes.
  2. **Constraints**:
     - Each city must be visited exactly once.
     - The route must return to the starting city.

  Types of TSP:

  1. **Symmetric TSP**: The distance from city A to city B is the same as from city B to city A.
  2. **Asymmetric TSP**: The distance from city A to city B may be different from the distance from city B to city A.

  Approaches to Solve TSP:

  1. **Brute Force**:
     - Generate all possible permutations of city visits, calculate the total distance for each permutation, and choose the minimum distance.
     - This approach is computationally expensive (O(n!)) and becomes infeasible for large numbers of cities.
  2. **Dynamic Programming**:
     - Use a recursive approach with memoization to store results of subproblems.
     - This significantly reduces the number of computations compared to brute force.
  3. **Heuristic Methods**:
     - Techniques such as genetic algorithms, simulated annealing, or ant colony optimization can provide good approximations for larger datasets where exact solutions are impractical.

  Implementing TSP in Prolog:

  Prolog, with its declarative nature and built-in support for backtracking, is well-suited for exploring combinatorial problems like TSP.

  Key Components in Prolog:

1. **Representation of Cities and Distances**:
   - Use facts to represent distances between cities, e.g.:

     ```
     distance(a, b, 5).
     distance(a, c, 10).
     distance(b, c, 3).
     ```
2. **Generating Permutations**:
   - Define a predicate to generate all permutations of the cities to explore different routes.
   - Prolog's backtracking can be leveraged to find all possible routes.
3. **Calculating Route Length**:
   - Define a predicate to calculate the total distance for a given route.
   - This will involve summing up the distances between consecutive cities in the route.
4. **Finding the Optimal Route**:
   - Combine the above components to find the route with the minimum distance by comparing the distances of all permutations.

- **Tools / Material Needed:**
  - **Hardware:** Laptop / Computer
  - **Software:** VS code or GNU prolog or any other IDE
- **Procedure:**

Step-by-Step Procedure for Solving TSP in Prolog

1. **Set Up Prolog Environment**:
   - Ensure you have a Prolog interpreter installed (e.g., **GNU Prolog**).
   - Open a terminal or command prompt to access the Prolog interpreter.
2. **Create a Prolog Source File**:
   - Open a text editor and create a new file with a .pl extension, such as tsp.pl.
   - This file will contain the Prolog code for solving the TSP.
3. **Define Distance Facts**:
   - Start by defining the distances between the cities as facts in your Prolog file.
   - Example:

     ```
     distance(a, b, 5).
     distance(a, c, 10).
     distance(b, c, 3).
     distance(b, d, 8).
     distance(c, d, 2).
     distance(a, d, 7).
     ```
   - This represents the distances between pairs of cities.
4. **Generate Permutations of Cities**:
   - Create a predicate to generate all possible routes (permutations) of the cities.
   - Example:

```
route(Cities, Route) :-
    permutation(Cities, Route).
```

- This predicate uses Prolog's built-in permutation/2 to generate all possible arrangements of the cities.

5. **Calculate Route Length**:
   - Define a predicate that calculates the total distance of a given route.
   - Example:

```
route_length([_], 0).  % Base case: no distance for a single city
route_length([City1, City2 | Rest], Length) :-
    distance(City1, City2, D),
    route_length([City2 | Rest], RestLength),
    Length is D + RestLength.
```

   - This predicate recursively sums the distances between consecutive cities.

6. **Find Optimal Route**:
   - Create a predicate that finds the optimal route by iterating through all generated routes and keeping track of the minimum length.
   - Example:

```
tsp(Cities, OptimalRoute, MinLength) :-
    findall(Route, route(Cities, Route), Routes),
    find_min_route(Routes, OptimalRoute, MinLength).

find_min_route([Route], Route, Length) :-
    route_length(Route, Length).
find_min_route([Route1, Route2 | Rest], OptimalRoute, MinLength) :-
    route_length(Route1, Length1),
    route_length(Route2, Length2),
    Length1 < Length2,
    find_min_route([Route1 | Rest], OptimalRoute, MinLength).
find_min_route([Route1, Route2 | Rest], OptimalRoute, MinLength) :-
    route_length(Route1, Length1),
    route_length(Route2, Length2),
    Length1 >= Length2,
    find_min_route([Route2 | Rest], OptimalRoute, MinLength).
```

   - The find_min_route/3 predicate compares the lengths of routes and keeps track of the shortest one.

7. **Save the Prolog File**:
   - Save the tsp.pl file after defining the facts and predicates.

8. **Open the Prolog Interpreter**:
   - Open a terminal or command prompt.

- Navigate to the directory where the tsp.pl file is saved using the cd command.

9. **Load the Prolog Program**:
   - Start the Prolog interpreter by typing gprolog or the appropriate command for your Prolog setup.
   - Load the Prolog file using:
     ```
     ?- consult('tsp.pl').
     ```
   - Alternatively, use the shorthand:
     ```
     ?- ['tsp.pl'].
     ```
10. **Run the Program with a Query**:
    - To find the optimal route for the cities defined in your facts (e.g., for cities a, b, c, and d), run the following query:
      ```
      ?- tsp([a, b, c, d], OptimalRoute, MinLength).
      ```
    - **Explanation**: This query asks Prolog to find the best route for visiting all the specified cities.
11. **Observe the Output**:
    - Prolog will output the optimal route and its length.
    - For example, it may return:
      ```
      OptimalRoute = [a, b, d, c, a],
      MinLength = 20.
      ```
12. **Adjust and Experiment**:
    - Modify the distances in your facts or add new cities to explore how the solution changes.
    - Test with different configurations and observe how the backtracking mechanism finds the optimal solution.
13. **Exit the Prolog Interpreter**:
    - Once you are done testing, you can exit the Prolog interpreter using:
      ```
      ?- halt.
      ```

- **Code:**

Facts:

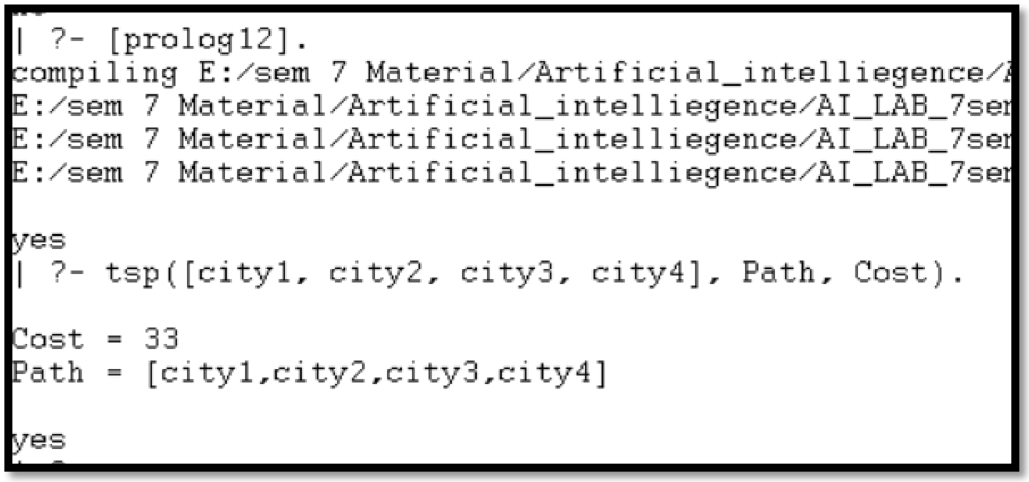
```
distance(city1, city2, 10).
distance(city2, city3, 15).
distance(city3, city4, 8).
distance(city4, city1, 20).

tsp(Cities, Path, Cost) :-
    permutation(Cities, Path),
    length(Path, N),
    tsp_cost(Path, Cost),
    !.

tsp_cost([C1, C2|Cs], Cost) :-
    distance(C1, C2, D),
```

```
        tsp_cost([C2|Cs], R),
        Cost is D + R.
    tsp_cost([C], 0).
```

Queries:

```
    tsp([city1, city2, city3, city4], Path, Cost).
```

- **Output:**



```
| ?- [prolog12].
compiling E:/sem 7 Material/Artificial_intelliegence/A
E:/sem 7 Material/Artificial_intelliegence/AI_LAB_7sem
E:/sem 7 Material/Artificial_intelliegence/AI_LAB_7sem
E:/sem 7 Material/Artificial_intelliegence/AI_LAB_7sem

yes
| ?- tsp([city1, city2, city3, city4], Path, Cost).

Cost = 33
Path = [city1,city2,city3,city4]

yes
```

- **Signature:**