1) Write a Prolog program to merge two ordered list to generate a third ordered list.
2) Write a Prolog program to add an element in a last position in a given list.
3) Write a Prolog program to find the n th element of a list.
4) Write a Prolog program to check whether a year is a leap year or not.
5) Write a Prolog program to check if a list is a palindrome.
6) Write a Prolog program to remove duplicates from a list.
7) Write a Prolog program to split a list into two halves.
8) Write a Prolog program to rotate a list N places to the left.
9) Write a Prolog program to find the union of two lists.
10) Write a Prolog program to replace all occurrences of an element in a list with another element.

Experiment NO: 01

Experiment Name: Write a Prolog program to merge two ordered list to generate a third ordered list.

1. Objective: The objective of this lab is to implement a Prolog program that merges two ordered lists into a third ordered list. This program takes two sorted lists as input and produces a new sorted list containing all elements from both input lists in ascending order.

2. Introduction: Merging two ordered lists is a fundamental operation in computer science, commonly used in sorting algorithms like Merge Sort. In Prolog, list operations rely on recursion and pattern matching, making it an ideal language for implementing list merging.

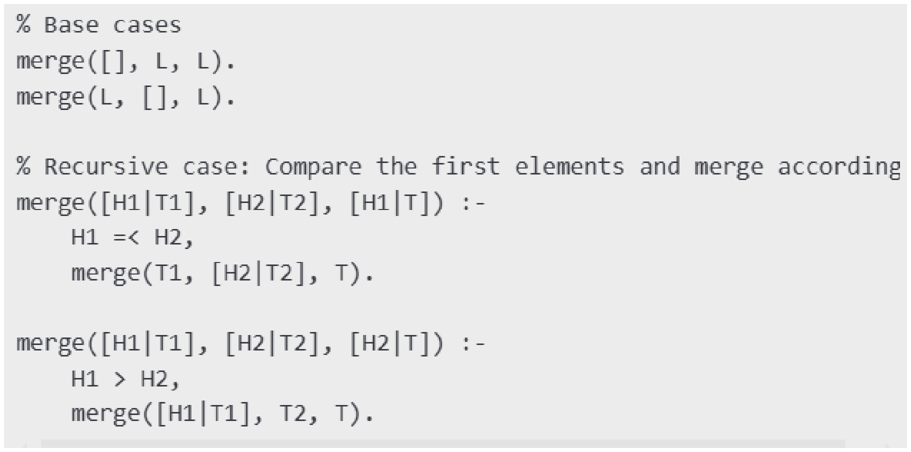
3. Materials and Methods:

- Programming Language: Prolog
- Software: SWI-Prolog or any other Prolog interpreter
- Method: Recursion and pattern matching to merge two sorted lists efficiently.

4. Algorithm:

1. If the first list is empty, return the second list.
2. If the second list is empty, return the first list.
3. Compare the first elements of both lists:
    - If the first element of the first list is smaller or equal, add it to the merged list and recursively call the function with the rest of the first list and the full second list.
    - Otherwise, add the first element of the second list to the merged list and recursively call the function with the rest of the second list and the full first list.

5. Prolog Code Implementation:

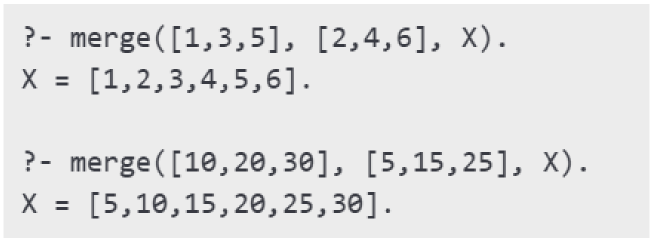


```
% Base cases
merge([], L, L).
merge(L, [], L).

% Recursive case: Compare the first elements and merge according
merge([H1|T1], [H2|T2], [H1|T]) :-
    H1 =< H2,
    merge(T1, [H2|T2], T).

merge([H1|T1], [H2|T2], [H2|T]) :-
    H1 > H2,
    merge([H1|T1], T2, T).
```

6. Explanation of the Code:

1) The first two base cases handle empty lists. If one of the lists is empty, the result is simply the non-empty list.
2) The recursive rules compare the first elements of both lists:
    I. If the first element of the first list is smaller or equal, it is placed in the merged list, and the function is called recursively with the rest of the first list and the second list.
    II. Otherwise, the first element of the second list is placed in the merged list, and the function is called recursively with the rest of the second list and the first list.

7. Example Execution:



```
?- merge([1,3,5], [2,4,6], X).
X = [1,2,3,4,5,6].

?- merge([10,20,30], [5,15,25], X).
X = [5,10,15,20,25,30].
```

8. Analysis and Discussion:

1) The implementation ensures that the merged list is also sorted because elements are appended in order.

2) The algorithm runs in **O(n + m)** time complexity, where **n** and **m** are the sizes of the two input lists.
3) The program is purely recursive and does not require additional memory for sorting.
4) The base cases handle scenarios where either or both input lists are empty.

9. Conclusion: This lab demonstrated how to merge two sorted lists using recursion in Prolog. The implementation effectively maintains the order of elements while merging. This fundamental operation is useful in various applications such as sorting and data merging tasks.

Experiment NO: 02

Experiment Name: Write a Prolog program to add an element in a last position in a given list.

Objective:

The objective of this lab is to write a Prolog program that adds an element to the last position of a given list and understand the recursive approach used in Prolog for list manipulation.

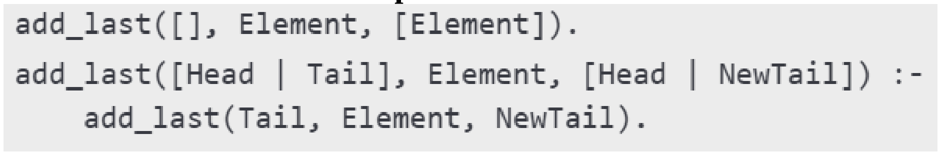
Theory:

Prolog is a logic programming language primarily used for symbolic computation and AI-based applications. List manipulation is a fundamental aspect of Prolog, and recursive rules are often used to process lists.
A list in Prolog is represented as:

```
[Head | Tail]
```

where `Head` is the first element, and `Tail` is the remaining list.
To add an element at the last position of a list, recursion is used to traverse the list until the base condition (empty list) is met, and then the element is appended.

Implementation:

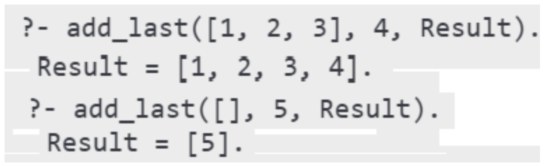


```
add_last([], Element, [Element]).
add_last([Head | Tail], Element, [Head | NewTail]) :-
    add_last(Tail, Element, NewTail).
```

Explanation:

1. **Base Case:** If the list is empty (`[]`), the result is a new list containing only the element to be added.
2. **Recursive Case:** The function keeps the first element (`Head`) intact and recursively processes the remaining list (`Tail`) until it reaches the base case.

Example Execution:



```
?- add_last([1, 2, 3], 4, Result).
 Result = [1, 2, 3, 4].
?- add_last([], 5, Result).
 Result = [5].
```

Observations:

1) The program successfully appends an element to the last position of any given list.
2) The recursion ensures that all elements before the last position remain unchanged.
3) The base case ensures correct termination when the list becomes empty.

Conclusion:

This experiment demonstrates how Prolog's recursive nature can be effectively utilized for list manipulation. The `add_last/3` predicate successfully adds an element to the last position of a given list by using recursion to traverse the list structure. This approach is efficient and aligns with Prolog's declarative paradigm.

Experiment NO: 03

Experiment Name: Write a Prolog program to add an element in a last position in a given list.

Objective:

The objective of this lab is to write a Prolog program that retrieves the Nth element from a given list and understand the recursive approach used in Prolog for list manipulation.

Theory:

Prolog is a logic programming language primarily used for symbolic computation and AI-based applications. List manipulation is a fundamental aspect of Prolog, and recursive rules are often used to process lists.A list in Prolog is represented as:

`[Head | Tail]`
where `Head` is the first element, and `Tail` is the remaining list. To find the Nth element of a list, recursion is used to traverse the list until the desired position is reached.

Implementation:

```
nth_element([Head | _], 1, Head).
nth_element([_ | Tail], N, Element) :-
    N > 1,
    N1 is N - 1,
    nth_element(Tail, N1, Element).
```

Explanation:

1. **Base Case:** If `N` is `1`, return the first element (`Head`) of the list.
2. **Recursive Case:** Decrease `N` by `1` and recursively process the tail of the list until `N` becomes `1`.

Example Execution:

```
?- nth_element([10, 20, 30, 40, 50], 3, Result).
 Result = 30.
?- nth_element([a, b, c, d, e], 5, Result).
 Result = e.
```

Observations:

1) The program successfully retrieves the Nth element from any given list.
2) The recursion ensures that all elements before the desired position are ignored.
3) The base case ensures correct termination when the desired index is reached.

Conclusion:
This experiment demonstrates how Prolog's recursive nature can be effectively utilized for list manipulation. The `nth_element/3` predicate successfully retrieves the Nth element from a given list by using recursion to traverse the list structure. This approach is efficient and aligns with Prolog's declarative paradigm.

Experiment NO: 04

Experiment Name: Write a Prolog program to check whether a year is a leap year or not.

Objective: The objective of this lab is to write a Prolog program that checks whether a given year is a leap year or not using logical conditions.

Theory: Prolog is a declarative programming language used for symbolic computation and logic-based applications. In Prolog, decision-making is accomplished using rules and facts. A leap year follows these conditions:

1. If a year is divisible by 400, then it is a leap year.
2. If a year is divisible by 100 but not by 400, then it is not a leap year.
3. If a year is divisible by 4 but not by 100, then it is a leap year.
4. Otherwise, it is not a leap year.

Implementation: The Prolog rule to check for a leap year is defined as follows:

```
is_leap_year(Year) :-
    Year mod 400 =:= 0.
is_leap_year(Year) :-
    Year mod 100 =\= 0,
    Year mod 4 =:= 0.
is_leap_year(Year) :-
    Year mod 4 =\= 0,
    write(Year), write(' is not a leap year.'), nl, fail.
```

Explanation:

1. **Divisibility by 400:** If the year is divisible by 400, it is a leap year.
2. **Divisibility by 100 but not by 400:** The year is not a leap year if it is only divisible by 100.
3. **Divisibility by 4 but not by 100:** If the year is divisible by 4 but not by 100, it is a leap year.
4. **Else Condition:** If none of the above conditions are met, it is not a leap year.

Example Execution:

```
?- is_leap_year(2020).
true.
?- is_leap_year(1900).
false.
```

Observations:

1) The program successfully determines whether a year is a leap year or not.
2) The conditions follow the standard leap year rules.
3) The program handles different cases correctly using Prolog’s logical rules.

Conclusion: This experiment demonstrates how Prolog can be used for logical decision-making. The `is_leap_year/1` predicate successfully evaluates whether a given year is a leap year based on the standard leap year rules. The approach is efficient and showcases Prolog's ability to handle rule-based logic effectively.

Experiment NO: 05

Experiment Name: Write a Prolog program to check if a list is a palindrome.

Objective: The objective of this lab is to write a Prolog program that checks whether a given list is a palindrome. A list is considered a palindrome if it reads the same forward and backward.

Theory: Prolog is a declarative programming language that is widely used in artificial intelligence and symbolic computation. List manipulation is an important concept in Prolog, and recursion is often used to process lists.

A palindrome list follows these conditions:

1. An empty list (`[]`) or a single-element list (`[X]`) is always a palindrome.
2. A list is a palindrome if the first and last elements are the same, and the sublist between them is also a palindrome.
3. The reverse of a palindrome list is the same as the original list.

To check if a list is a palindrome, we can use the built-in `reverse/2` predicate to reverse the list and compare it with the original list.

Implementation: The Prolog rule to check if a list is a palindrome is defined as follows:

```
is_palindrome(List) :-
    reverse(List, ReversedList),
    List = ReversedList.
```

Explanation:

1. The `reverse/2` predicate is used to generate the reversed version of the input list.
2. The original list is compared with the reversed list.
3. If both lists are equal, the original list is a palindrome; otherwise, it is not.

Example Execution:

```
?- is_palindrome([r, a, d, a, r]).
true.
?- is_palindrome([1, 2, 3, 4, 5]).
false.
```

Observations:

1) The program correctly identifies whether a list is a palindrome.
2) The use of `reverse/2` makes the implementation straightforward and efficient.
3) The program works with lists containing numbers, characters, or symbols.

Conclusion: This experiment demonstrates how Prolog's built-in predicates can simplify list processing. The `is_palindrome/1` predicate effectively determines whether a given list is a palindrome by using the `reverse/2` predicate. This approach aligns well with Prolog’s declarative nature and logical problem-solving capabilities.

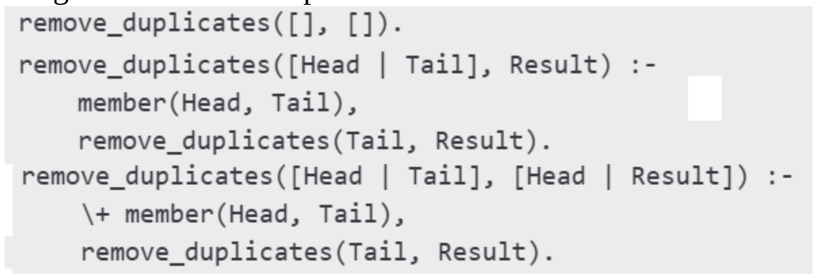
Experiment NO: 06

Experiment Name: Write a Prolog program to remove duplicates from a list.

Objective: The objective of this lab is to write a Prolog program that removes duplicate elements from a given list using recursion.
Theory: Prolog is a declarative logic programming language used for symbolic computation and artificial intelligence applications. List processing is an essential feature of Prolog, and recursion is commonly used to manipulate lists.
To remove duplicates from a list, we follow these steps:

1. If the list is empty, the result is also an empty list.
2. If the head of the list is already present in the tail, ignore it and process the rest.
3. Otherwise, include the head and continue processing the tail.

Implementation: The Prolog rule to remove duplicates from a list is defined as follows:

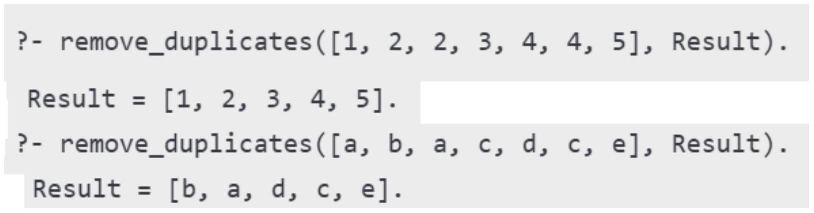


```
remove_duplicates([], []).
remove_duplicates([Head | Tail], Result) :-
    member(Head, Tail),
    remove_duplicates(Tail, Result).
remove_duplicates([Head | Tail], [Head | Result]) :-
    \+ member(Head, Tail),
    remove_duplicates(Tail, Result).
```

Explanation:

1. **Base Case:** If the list is empty, return an empty list.
2. **Recursive Case 1:** If the head of the list is already in the tail, remove it and process the remaining elements.
3. **Recursive Case 2:** If the head is not in the tail, keep it and process the remaining elements.

Example Execution:



```
?- remove_duplicates([1, 2, 2, 3, 4, 4, 5], Result).
Result = [1, 2, 3, 4, 5].
?- remove_duplicates([a, b, a, c, d, c, e], Result).
Result = [b, a, d, c, e].
```

Observations:

1) The program successfully removes duplicate elements while preserving the relative order.
2) The use of `member/2` helps in checking for existing elements.
3) The recursive approach ensures that all elements are processed.

Conclusion: This experiment demonstrates how Prolog's list manipulation capabilities can be used to remove duplicate elements from a list. The `remove_duplicates/2` predicate efficiently eliminates duplicates while maintaining order, showcasing Prolog's ability to handle recursive list processing.

<u>Experiment NO: 07</u>

Experiment Name: Write a Prolog program to split a list into two halves.
Objective: The objective of this lab is to write a Prolog program that splits a given list into two halves.
Theory: Prolog is a logic programming language widely used for symbolic computation and artificial intelligence applications. List manipulation is a fundamental operation in Prolog, and recursion is commonly used to traverse and modify lists.
To split a list into two halves, we follow these steps:

1. Find the length of the list.
2. Divide the length by 2 to determine the splitting point.
3. Use recursion to extract the first half and place the remaining elements in the second half.

Implementation: The Prolog rule to split a list into two halves is defined as follows:

```
split_list([], [], []).
split_list([X], [X], []).
split_list(List, FirstHalf, SecondHalf) :-
    length(List, Len),
    HalfLen is Len // 2,
    split_at(HalfLen, List, FirstHalf, SecondHalf).
split_at(0, List, [], List).
split_at(N, [H|T], [H|First], Second) :-
    N > 0,
    N1 is N - 1,
    split_at(N1, T, First, Second).
```

Explanation:

1. **Base Case:** An empty list splits into two empty lists.
2. **Single Element Case:** If the list has one element, the first half contains the element, and the second half is empty.
3. **Recursive Case:**
   I. Compute the length of the list and determine the splitting index.
   II. Use `split_at/4` to extract the first `HalfLen` elements and separate the rest as the second half.

Example Execution:

```
?- split_list([1, 2, 3, 4, 5, 6], FirstHalf, SecondHalf).
FirstHalf = [1, 2, 3],
SecondHalf = [4, 5, 6].
?- split_list([a, b, c, d, e], FirstHalf, SecondHalf).
FirstHalf = [a, b],
SecondHalf = [c, d, e].
```

Observations:

1) The program correctly divides lists into two halves.
2) The `split_at/4` predicate efficiently extracts the first half using recursion.
3) It handles both even and odd-length lists properly.

Conclusion: This experiment demonstrates how Prolog's list processing capabilities can be used to split a list into two halves. The `split_list/3` predicate effectively determines the midpoint and separates the elements recursively.

Experiment NO: 08

Experiment Name: Write a Prolog program to rotate a list N places to the left.

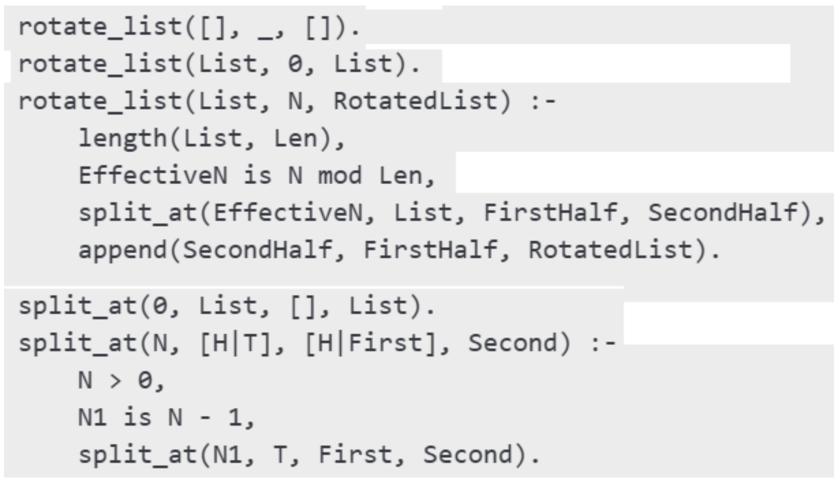
Objective: The objective of this lab is to write a Prolog program that rotates a given list N places to the left.

Theory: Prolog is a logic programming language widely used for symbolic computation and artificial intelligence applications. List manipulation is a fundamental operation in Prolog, and recursion is commonly used to traverse and modify lists.

To rotate a list N places to the left, we follow these steps:

1. Determine the length of the list.
2. Compute the effective number of rotations (N mod Length).
3. Split the list at the computed index.
4. Concatenate the second half with the first half to achieve the rotation.

Implementation: The Prolog rule to rotate a list N places to the left is defined as follows:

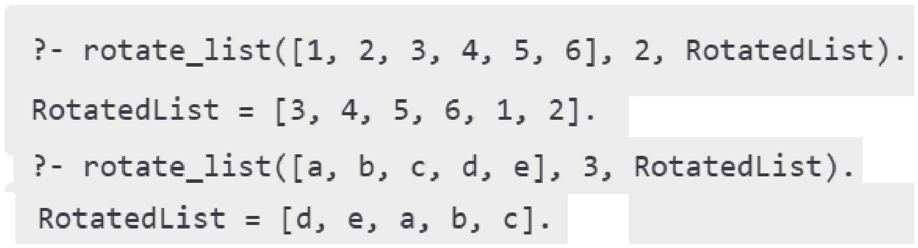


```
rotate_list([], _, []).
rotate_list(List, 0, List).
rotate_list(List, N, RotatedList) :-
    length(List, Len),
    EffectiveN is N mod Len,
    split_at(EffectiveN, List, FirstHalf, SecondHalf),
    append(SecondHalf, FirstHalf, RotatedList).

split_at(0, List, [], List).
split_at(N, [H|T], [H|First], Second) :-
    N > 0,
    N1 is N - 1,
    split_at(N1, T, First, Second).
```

Explanation:

1. **Base Case:** Rotating an empty list results in an empty list.
2. **Zero Rotation Case:** If N is 0, the list remains unchanged.
3. **General Case:**
    I. Compute `EffectiveN` as `N mod Length` to handle cases where N is greater than the length of the list.
    II. Use `split_at/4` to divide the list into two parts.
    III. Append the second half to the first half to complete the rotation.

Example Execution:



```
?- rotate_list([1, 2, 3, 4, 5, 6], 2, RotatedList).
RotatedList = [3, 4, 5, 6, 1, 2].
?- rotate_list([a, b, c, d, e], 3, RotatedList).
RotatedList = [d, e, a, b, c].
```

Observations:

I. The program correctly rotates lists by any given number of places.
II. The use of `mod` ensures that unnecessary full rotations are avoided.
III. The `split_at/4` predicate efficiently extracts the required portion of the list.

Conclusion: This experiment demonstrates how Prolog's list processing capabilities can be used to rotate a list N places to the left. The `rotate_list/3` predicate effectively determines the rotation index, splits the list, and concatenates the parts accordingly.

Experiment NO: 09

Experiment Name: Write a Prolog program to find the union of two lists.

Objective: The objective of this lab is to write a Prolog program that finds the union of two given lists.

Theory: Prolog is a logic programming language widely used for symbolic computation and artificial intelligence applications. List manipulation is a fundamental operation in Prolog, and recursion is commonly used to traverse and modify lists.

The union of two lists consists of all unique elements from both lists. To find the union, we follow these steps:

1. Include elements from the first list.
2. Check each element of the second list to see if it is already in the first list.
3. If the element is not present, add it to the result.
4. Recursively process the remaining elements.

Implementation: The Prolog rule to find the union of two lists is defined as follows:

```
list_union([], L, L).
list_union([H|T], L2, Result) :-
    member(H, L2),
    list_union(T, L2, Result).
list_union([H|T], L2, [H|Result]) :-
    \+ member(H, L2),
    list_union(T, L2, Result).
```

Explanation:

1. **Base Case:** If the first list is empty, return the second list as the union.
2. **Recursive Case 1:**
   I. If the head of the first list exists in the second list, skip it.
   II. Recursively process the rest of the list.
3. **Recursive Case 2:**
   I. If the head does not exist in the second list, add it to the result.
   II. Continue processing the remaining elements.

Example Execution:

```
?- list_union([1, 2, 3], [3, 4, 5], Result).
Result = [1, 2, 3, 4, 5].
?- list_union([a, b, c], [c, d, e], Result).
Result = [a, b, c, d, e].
```

Observations:

1) The program correctly identifies unique elements from both lists.
2) The `member/2` predicate helps in checking for duplicates.
3) The recursive approach ensures all elements are processed efficiently.

Conclusion: This experiment demonstrates how Prolog's list processing capabilities can be used to find the union of two lists. The `list_union/3` predicate effectively combines both lists while eliminating duplicates.

Experiment NO: 10

Experiment Name: Write a Prolog program to replace all occurrences of an element in a list with another element.

Objective: The objective of this lab is to write a Prolog program that replaces all occurrences of a specific element in a given list with another element.

Theory: Prolog is a logic programming language widely used for symbolic computation and artificial intelligence applications. List manipulation is a fundamental operation in Prolog, and recursion is commonly used to traverse and modify lists.

To replace all occurrences of an element in a list, we follow these steps:

1. If the list is empty, return an empty list.
2. Compare the head of the list with the target element.
3. If the head matches the target element, replace it with the new element and process the rest of the list.
4. If the head does not match, keep it unchanged and process the rest of the list recursively.

Implementation: The Prolog rule to replace all occurrences of an element in a list is defined as follows:

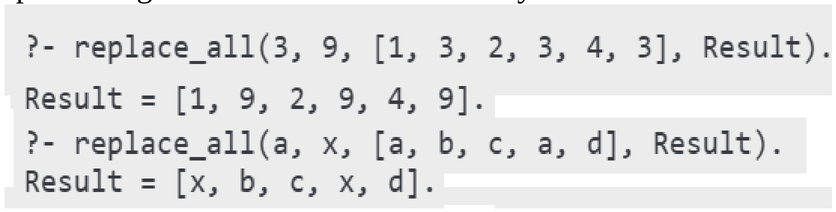
```
replace_all(_, _, [], []).
replace_all(X, Y, [X|T], [Y|Result]) :-
    replace_all(X, Y, T, Result).
replace_all(X, Y, [H|T], [H|Result]) :-
    X \= H,
    replace_all(X, Y, T, Result).
```

Explanation:

1. **Base Case:** If the input list is empty, return an empty list.
2. **Recursive Case 1:**
   I. If the head of the list matches the element to be replaced (`X`), substitute it with `Y`.

    II. Recursively process the remaining elements.

3. **Recursive Case 2:**
    I. If the head does not match `X`, retain it as is.
    II. Continue processing the rest of the list recursively.



```
?- replace_all(3, 9, [1, 3, 2, 3, 4, 3], Result).
Result = [1, 9, 2, 9, 4, 9].
?- replace_all(a, x, [a, b, c, a, d], Result).
Result = [x, b, c, x, d].
```

Observations:

1) The program correctly replaces all occurrences of a given element.
2) The recursive approach ensures all elements are processed efficiently.
3) The `\=` operator ensures that only matching elements are replaced.

Conclusion: This experiment demonstrates how Prolog's list processing capabilities can be used to replace elements within a list. The `replace_all/4` predicate effectively traverses the list and modifies it as needed.