Task 1 – List in Prolog

Write a program in Prolog to implement each of the following:

1. To check if an item is in a list
2. Calculate the length of a list
3. Concatenate two lists
4. Delete an item from a List
5. Append an item to a List
6. Insert an item in a List

(i) To check if an item is in a list:

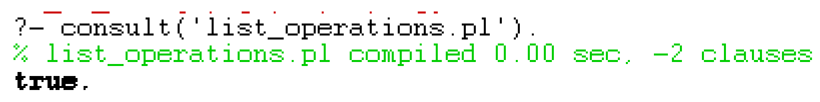
% Base case: If the item is the head of the list, it's in the list.

is_in_list(Item, [Item|_]).

% Recursive case: Check the rest of the list.

is_in_list(Item, [_|Tail]) :- is_in_list(Item, Tail).

Screenshot:



```
?- consult('list_operations.pl').
% list_operations.pl compiled 0.00 sec, -2 clauses
true.
```

(ii) Calculate the length of a list:

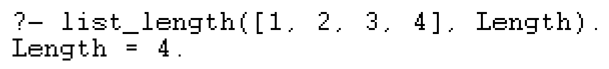
% Base case: An empty list has length 0.

list_length([], 0).

% Recursive case: Increment the length by 1 for each element in the list.

list_length([_|Tail], Length) :- list_length(Tail, TailLength), Length is TailLength + 1.

Screenshot:



```
?- list_length([1, 2, 3, 4], Length).
Length = 4.
```

(iii) Concatenate two lists:

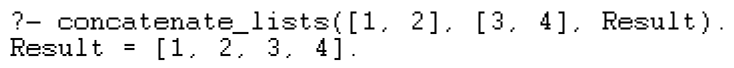
% Base case: Concatenating an empty list with another list gives the same list.

concatenate_lists([], List, List).

% Recursive case: Concatenate the Head of the first list with the result of the concatenation of the rest.

concatenate_lists([Head|Tail], List2, [Head|Result]) :- concatenate_lists(Tail, List2, Result).

Screenshot:



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

(iv) Delete an item from a List:

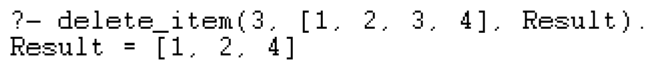
% Base case: If the item is the head of the list, remove it.

delete_item(Item, [Item|Tail], Tail).

% Recursive case: Keep the Head of the list and continue deleting in the Tail.

delete_item(Item, [Head|Tail], [Head|Result]) :- delete_item(Item, Tail, Result).

Screenshot:



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

(v) Append an item to a List:

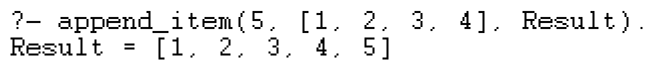
% To append an item to an empty list, we create a list with just the item.

append_item(Item, [], [Item]).

% To append an item to a non-empty list, we add the item to the Head of the list.

append_item(Item, [Head|Tail], [Head|Result]) :- append_item(Item, Tail, Result).

Screenshot:



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

vi) Insert an item in a List:

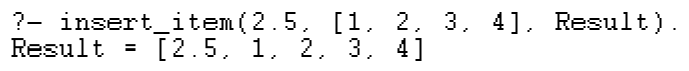
% To insert an item as the first element of the list, we just create a new list with the item at the head and the original list as the tail.

insert_item(Item, List, [Item|List]).

% To insert an item in the middle of the list, we keep the Head and recursively insert the item in the Tail.

insert_item(Item, [Head|Tail], [Head|Result]) :- insert_item(Item, Tail, Result).

Screenshot:



```
?- insert_item(2.5, [1, 2, 3, 4], Result).
Result = [2.5, 1, 2, 3, 4]
```

Task 2 - Write a program in Prolog to find the maximum value in a list of integer numbers.

To find the maximum value in a list of integer numbers, you can use recursion to traverse the list and keep track of the current maximum value.

```
% Base case: The maximum of a single-element list is the element itself.

max_in_list([X], X).

% Recursive case: Compare the Head of the list with the maximum value in the Tail.

max_in_list([Head | Tail], Max) :-

max_in_list(Tail, MaxTail),

Max is max(Head, MaxTail).
```

1. The base case states that the maximum of a single-element list is the element itself.
2. In the recursive case, we split the list into the Head (first element) and the Tail (the rest of the list). We then recursively find the maximum value in the Tail using the `max_in_list/2` predicate.
3. After obtaining the maximum value in the Tail, we use the built-in `max/2` predicate to compare the Head with the maximum value found in the Tail. The result is assigned to the variable `Max`.

Now, you can use the `max_in_list/2` predicate to find the maximum value in a list. For example:

```
?- max_in_list([1, 5, 3, 8, 2], Max).

Max = 8.

?- max_in_list([10, 5, 15, 3], Max).

Max = 15.

?- max_in_list([-3, -7, -1, -11], Max).

Max = -1.
```

To solve this task, we need to define the predicates `brother/2`, `cousin/2`, `grandson/2`, and `descendant/2` based on the given facts of the form `father(name1, name2)`.

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

```
% Two individuals X and Y are brothers if they have the same father.

brother(X, Y) :-

    father(Father, X),

    father(Father, Y),

    X \= Y. % Ensure X and Y are not the same individual.
```

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

```
% Two individuals X and Y are cousins if they have fathers who are
brothers.

cousin(X, Y) :-

    father(FatherX, X),

    father(FatherY, Y),

    brother(FatherX, FatherY),

    X \= Y. % Ensure X and Y are not the same individual.
```

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

```
% X is a grandson of Y if Y is the father of X's father.

grandson(X, Y) :-

    father(FatherX, X),

    father(Y, FatherX).
```

(iv) Define a predicate descendant(X, Y) which holds iff X is a descendant of Y.

```
% X is a descendant of Y if Y is the father of X, or Y is the father
of some ancestor of X.

descendant(X, Y) :-

    father(Y, X).

descendant(X, Y) :-

    father(Z, X),

    descendant(Z, Y).
```

Now, you can use these predicates to find information about the relationships between individuals in the family tree based on the given facts.

For example, if we have the following facts:

```
father(john, mike).

father(john, sara).

father(mike, alex).

father(john, alice).

father(alex, tom).
```

You can use the defined predicates to answer questions:

prolog

```
?- brother(mike, sara).

true.

?- cousin(alex, alice).

true.

?- grandson(tom, john).
```

```
true.

?- descendant(tom, john).

true.
```

These queries will return whether the given individuals are brothers, cousins, or descendants based on the provided family tree facts.