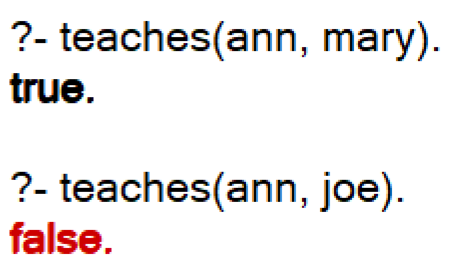
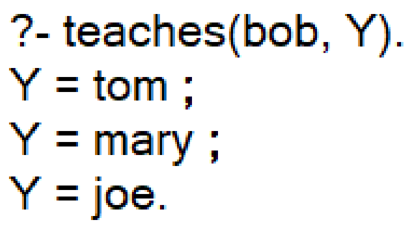
Assignment 2 – Cathal Lawlor – 21325456

Question 1

```
%# 1. Write a prolog rule called 'teaches' that
returns true if a given instructor teaches a given
student.

% Using conjunction as AND operator where both must
be in the same class Z
teaches(X,Y) :- instructs(X,Z), takes(Y,Z).
```

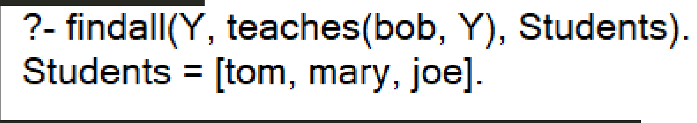


```
?- teaches(ann, mary).
true.

?- teaches(ann, joe).
false.
```



```
?- teaches(bob, Y).
Y = tom ;
Y = mary ;
Y = joe.
```

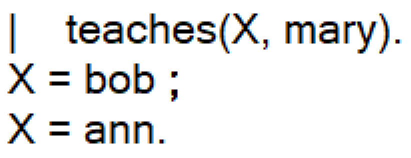
```
%# 2. Write a prolog query that uses the 'teaches'
rule to show all students instructed by bob.

% use ; until false is displayed to show all results
teaches(bob, Y).
```



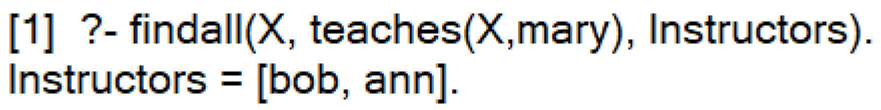
```
?- findall(Y, teaches(bob, Y), Students).
Students = [tom, mary, joe].
```

```
Note: you can use this query: ' findall(Y, teaches(bob, Y), Students). ' to
avoid having to press ; for each result. Not sure if this is against the spirit
of the question.
```

```
%3. Write a prolog query that uses the 'teaches' rule to show all instructors
that instruct mary.


teaches(X, mary). % use ; to show all results



/* Note: again you can do ' findall(X, teaches(X, mary), Lecturers). ' but not
sure if this is in the spirit of the assignment */
```

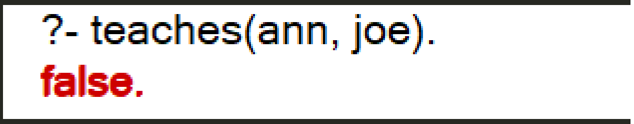


```
|    teaches(X, mary).
X = bob ;
X = ann.
```



```
[1]  ?- findall(X, teaches(X,mary), Instructors).
Instructors = [bob, ann].
```

```
%4. What is the result of the query:
teaches(ann, joe).

false

%Why is this the case?
```



```
?- teaches(ann, joe).
false.
```

The rule is filled as such:
'teaches(ann,joe) :- instructs(ann,ct345), takes(joe, ct345).'

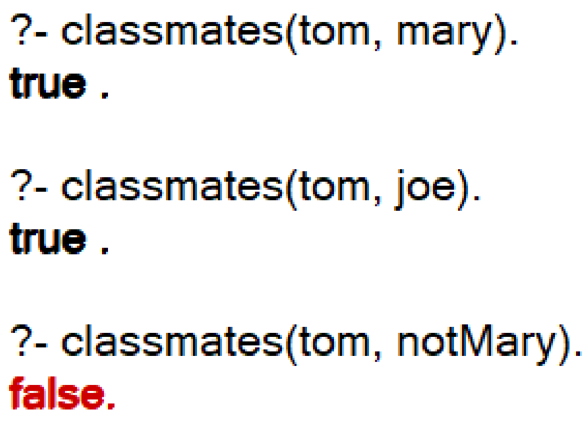
Where the rule fills out instructs(ann,ct345) is filled out (and only this as this is the only instructs fact).
The rule takes(joe, ct345) fails though as there is no fact to fill the query.
Thus, the query fails as the conjunction fails.

```
/*5. Write a prolog rule called 'classmates' that returns true if two students
take the same course. Demonstrate with suitable queries that this rule works as
described. [1 mark each]
*/



classmates(X,Y) :- takes(X, Z), takes(Y, Z).
```

Tom and Mary are true as both fulfil the conjunction
takes(X, Z), takes(Y, Z). with either module ct331 or ct345.

Tom and Joe are true as both fulfil the conjunction with the module ct331 as both take that in the takes(X, Z) fact.

Tom and notMary fail though as there is no takes fact with notMary in it, sharing a common module with Tom.

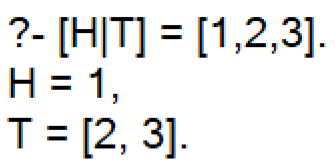


```
?- classmates(tom, mary).
true .

?- classmates(tom, joe).
true .

?- classmates(tom, notMary).
false.
```

Question 2

1. Using the “=” sign and the prolog list syntax to explicitly unify variables with parts of a list, write a prolog query that displays the head and tail of the list [1,2,3].

[H|T] = [1,2,3].

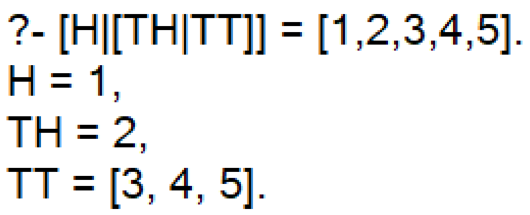


```
?- [H|T] = [1,2,3].
H = 1,
T = [2, 3].
```

---

2. Similarly, use a nested list to display the head of the list, the head of the tail of the list and the tail of the tail of the list [1,2,3,4,5].

[H|[TH|TT]] = [1,2,3,4,5].

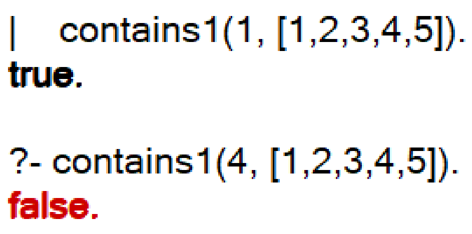


```
?- [H|[TH|TT]] = [1,2,3,4,5].
H = 1,
TH = 2,
TT = [3, 4, 5].
```

---

3. Write a prolog rule 'contains1' that returns true if a given element is the first element of a given list

contains1(X, [X|_]).

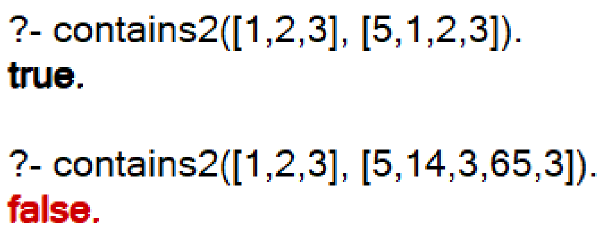


```
|    contains1(1, [1,2,3,4,5]).
true.

?- contains1(4, [1,2,3,4,5]).
false.
```

---

4. Write a prolog rule 'contains2' that returns true if a given list is the same as the tail of another given list.

contains2(Y, [_|T]) :- Y = T.

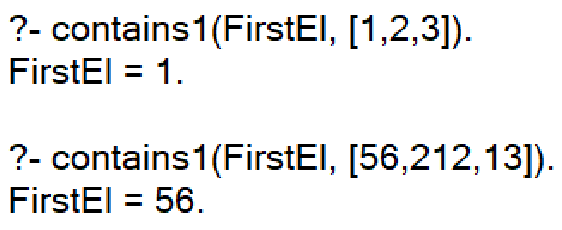


```
?- contains2([1,2,3], [5,1,2,3]).
true.

?- contains2([1,2,3], [5,14,3,65,3]).
false.
```

---

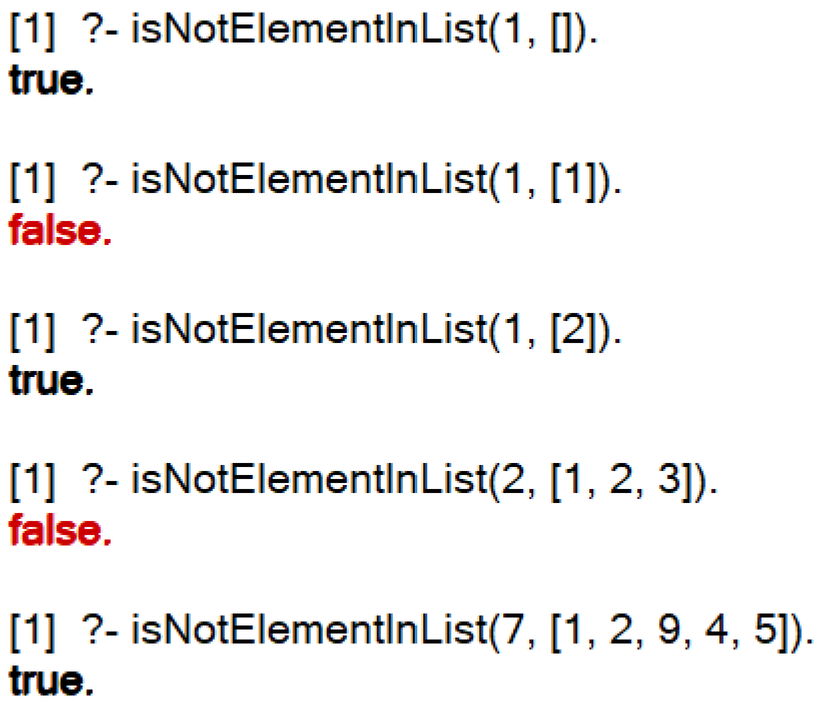
5. Write a prolog query using 'contains1' to display the first element of a given list.

contains1(FirstEl, [1,2,3]).

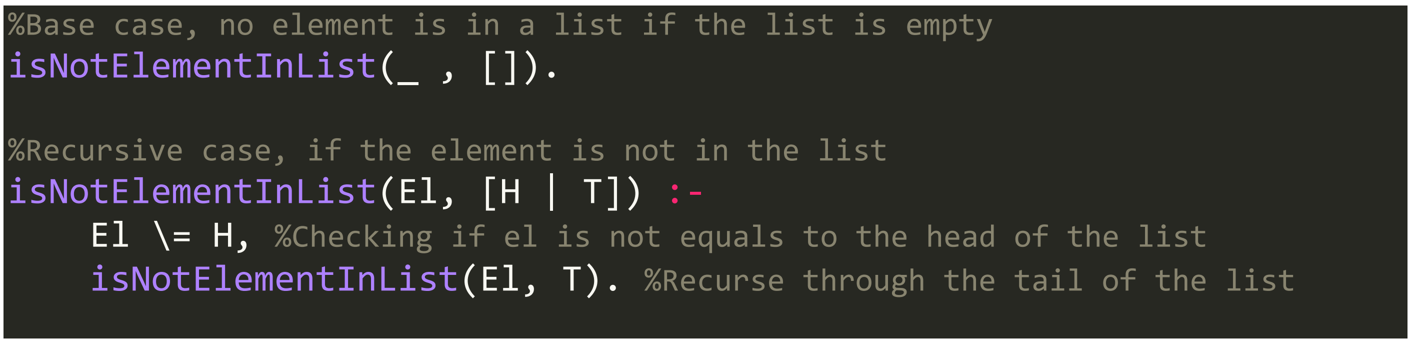


```
?- contains1(FirstEl, [1,2,3]).
FirstEl = 1.

?- contains1(FirstEl, [56,212,13]).
FirstEl = 56.
```

Question 3



```
[1] ?- isNotElementInList(1, []).
true.

[1] ?- isNotElementInList(1, [1]).
false.

[1] ?- isNotElementInList(1, [2]).
true.

[1] ?- isNotElementInList(2, [1, 2, 3]).
false.

[1] ?- isNotElementInList(7, [1, 2, 9, 4, 5]).
true.
```



```
%Base case, no element is in a list if the list is empty
isNotElementInList(_ , []).

%Recursive case, if the element is not in the list
isNotElementInList(El, [H | T]) :-
    El \= H, %Checking if el is not equals to the head of the list
    isNotElementInList(El, T). %Recurse through the tail of the list
```

I have a base case that deals with the case of an empty list or when the recursive rule has reached the end of the list.

The recursive rule takes in the element, and then checks the element to the head of the list, if it's not equals it recurses itself, sending the element and the remainder of the list being the tail.

If it encounters the end of the list it will return true, otherwise it will return fail to the query if it finds the element in the list.

Question 4

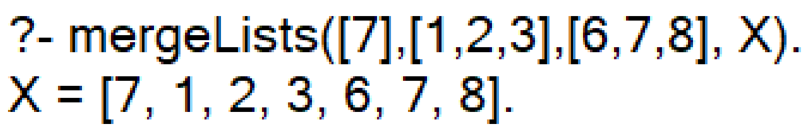
```prolog
/* Helper predicate: mergeLists/3
Merges two lists by merging the second and third lists and then merging the
first list with the result. */
mergeLists(List1, List2, List3, Merged) :-
    mergeLists(List2, List3, List23), % Merge the second and third lists.
    mergeLists(List1, List23, Merged). % Merge the first list with the result.

% Base case: Merging an empty list with another list results in the same list.
mergeLists([], List, List).

% Recursive rule: Merge two non-empty lists.
mergeLists([Head|Tail1], List2, [Head|ResultTail]) :-
    /*Take the head of the first list and then recursively merge the rest of
the first list with the second list. */
    mergeLists(Tail1, List2, ResultTail).
```

mergeLists/4 merges three lists: List1, List2, and List3. It does this by first merging List2 and List3 into List23, then merging List1 with List23 to get the final merged list Merged.

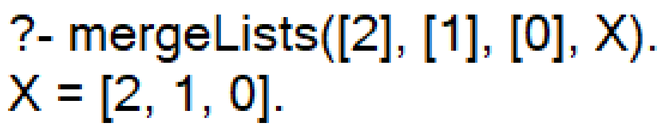
mergeLists/3 is a predicate that merges two lists. It has two base cases:

If the first list is empty ([]), the result is the second list.

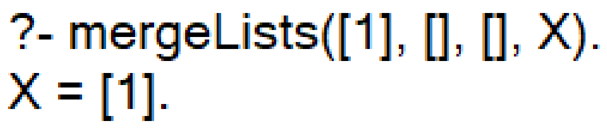
If the first list is not empty, it takes the head of the first list (the first element) and merges it with the result of merging the tail of the first list (the rest of the elements) with the second list. - This is a recursive process. The recursion continues until the first list is empty, at which point the second list is returned as the result.



```
?- mergeLists([7],[1,2,3],[6,7,8], X).
X = [7, 1, 2, 3, 6, 7, 8].
```



```
?- mergeLists([2], [1], [0], X).
X = [2, 1, 0].
```



```
?- mergeLists([1], [], [], X).
X = [1].
```

Question 5

```
% This is the helper function that will be called by the user.
reverseList(List, Reversed) :- reverseList(List, [], Reversed).

% Base case - reversing an empty list results in an empty list.
reverseList([], Reversed, Reversed).

/* Recursive case - add the head of the list to the accumulator and call
reverseList on the tail. */
reverseList([H|T], Acc, Reversed) :- reverseList(T, [H|Acc], Reversed).
```

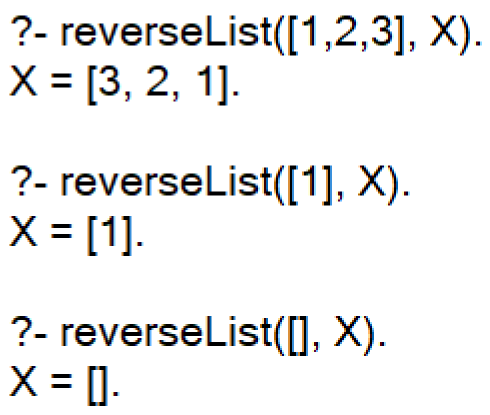
reverseList/2 is the predicate that users are expected to call. It takes two arguments: the list to be reversed (List) and the reversed list (Reversed). It calls the helper predicate reverseList/3 with an empty list as the initial value of the accumulator.

reverseList/3 is a helper predicate that uses an accumulator to hold the reversed list. It has two clauses:

1. The base case: when the list to be reversed is empty ([]), the accumulator holds the reversed list, so it's unified with Reversed.

2. The recursive case: when the list to be reversed is not empty, it splits the list into the head (H) and the tail (T), adds the head to the front of the accumulator, and calls reverseList/3 on the tail.

The accumulator is used to build up the result, and when the list to be transformed is empty, the accumulator holds the final result.



```
?- reverseList([1,2,3], X).
X = [3, 2, 1].

?- reverseList([1], X).
X = [1].

?- reverseList([], X).
X = [].
```

Question 6

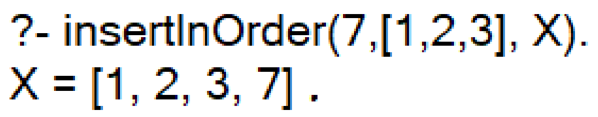
```prolog
/*Base case - list is empty, the element is placed in the return
list */
insertInOrder(X, [], [X]).

/*if the element is less than the head, it is placed at the start
of the list */
insertInOrder(X, [H|T], [X,H|T]) :-
    X =< H.

/*if the element is greater than the head, we recurse through the
rest of list - return a list having the head, and the return of the
recursive call as the tail */
insertInOrder(X, [H|T], [H|Rest]) :-
    X > H,
    insertInOrder(X, T, Rest). /*recursively go through the
remainder of the list, and return the list with the element
inserted in the correct position */
```

1. Base case: If the list is empty ([]), the element X is the only element in the list, so the result is a list containing just X.

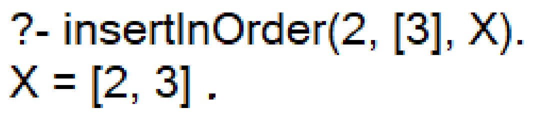
2. Recursive case 1: If the element X is less than or equal to the head of the list (H), X should be inserted before H. So, the result is a list with X as the first element, followed by the rest of the list ([H|T]).

3. Recursive case 2: If the element X is greater than the head of the list (H), X should be inserted somewhere in the tail of the list (T). So, the predicate calls itself recursively with X and T as arguments, and the result (Rest) is appended to H to form the final list.

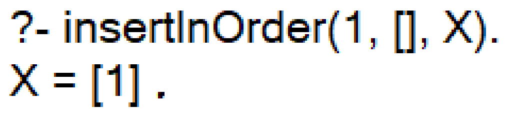
This predicate assumes that the input list is already sorted in ascending order. It maintains this order by inserting X at the correct position.



```
?- insertInOrder(7,[1,2,3], X).
X = [1, 2, 3, 7] .
```



```
?- insertInOrder(2, [3], X).
X = [2, 3] .
```



```
?- insertInOrder(1, [], X).
X = [1] .
```