# CT331 Assignment 3

Programming Paradigms

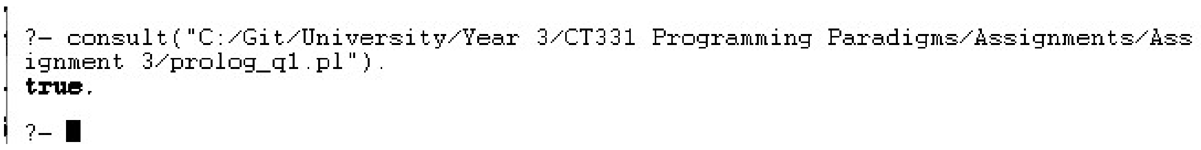
Maxwell Maia 21236277

## Question 1

1.

```
takes(tom, ct331).
takes(mary, ct331).
takes(joe, ct331).
takes(tom, ct345).
takes(mary, ct345).
instructs(bob, ct331).
instructs(ann, ct345).

teaches(Instructor, Student) :-
   instructs(Instructor, Course),
   takes(Student, Course).
```

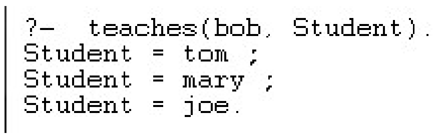
```
?- consult("C:/Git/University/Year 3/CT331 Programming
Paradigms/Assignments/Assignment 3/prolog_q1.pl").
```



```
?- consult("C:/Git/University/Year 3/CT331 Programming Paradigms/Assignments/Ass
ignment 3/prolog_q1.pl").
true.

?- 
```

2.

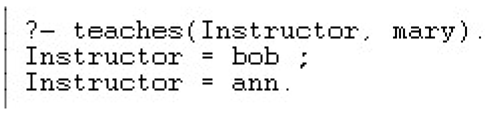
teaches(bob, Student).



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

3.

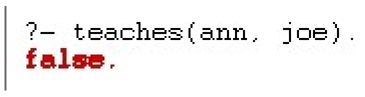
teaches(Instructor, mary).



```
?- teaches(Instructor, mary).
Instructor = bob ;
Instructor = ann.
```

4.

Result of query is false.



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

The teaches rule says that if an instructor instructs a course, and a student takes that same course, then the instructor teaches that student.

In the query: teaches(ann, joe).

The following are initialized:

Instructor: ann

Student: joe

```
teaches(ann, joe) :-
   instructs(ann, Course),
   takes(joe, Course).
```

Prolog finds the fact: instructs(ann, ct345).

And then initialises Course: ct345

Prolog finds the fact takes(joe, ct331).

This does not match the Course: ct345

So prolog skips this line.

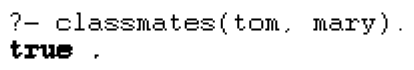
There are no other facts in the database of the nature: takes(joe, X).

Therefore, Prolog cannot prove that takes(joe, ct345). Therefore, Prolog cannot prove that teaches(ann, joe). with the data in the database. So it says false.

5.

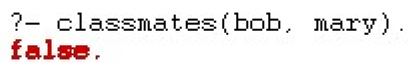
```
teaches(Instructor, Student) :-
   instructs(Instructor, Course),
   takes(Student, Course).

classmates(Student1, Student2) :-
   takes(Student1, Course),
   takes(Student2, Course).
```

Queries:



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

This is correct. Tom is a classmate of Mary.



```
?- classmates(bob, mary).
false.
```

This is correct. Bob is not a classmate of Mary since Bob does not take a course.

Searching for all pairs of classmates results in:

Duplicate pairs:

e.g.

Student1 = tom,

Student2 = mary ;

Student1 = mary,

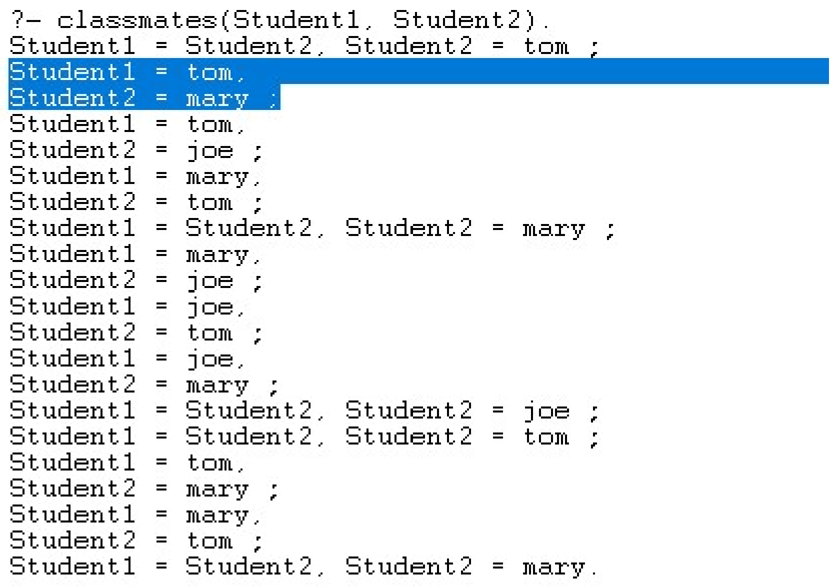
Student2 = tom ;

Students that are classmates with themselves:

e.g.

Student1 = Student2, Student2 = tom ;

Pairs of students that are classmates (correct answer).



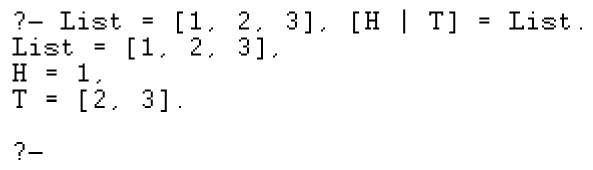
```
?- classmates(Student1, Student2).
Student1 = Student2, Student2 = tom ;
Student1 = tom,
Student2 = mary ;
Student1 = tom,
Student2 = joe ;
Student1 = mary,
Student2 = tom ;
Student1 = Student2, Student2 = mary ;
Student1 = mary,
Student2 = joe ;
Student1 = joe,
Student2 = tom ;
Student1 = joe,
Student2 = mary ;
Student1 = Student2, Student2 = joe ;
Student1 = Student2, Student2 = tom ;
Student1 = tom,
Student2 = mary ;
Student1 = mary,
Student2 = tom ;
Student1 = Student2, Student2 = mary.
```

A pair of classmates are highlighted.

## Question 2

1.

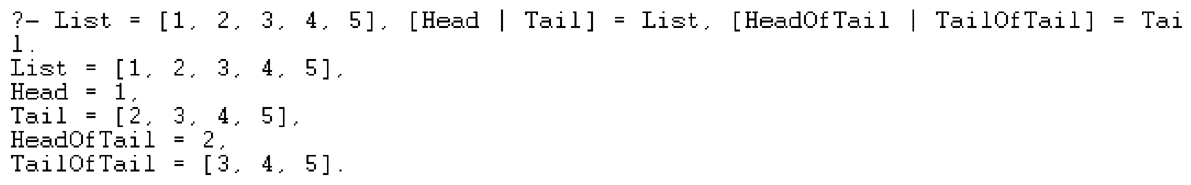
List = [1, 2, 3], [H | T] = List.



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

?-
```

2.

List = [1, 2, 3, 4, 5], [Head | Tail] = List, [HeadOfTail | TailOfTail] = Tail.

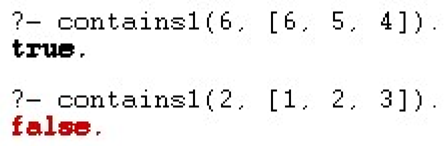


```
?- List = [1, 2, 3, 4, 5], [Head | Tail] = List, [HeadOfTail | TailOfTail] = Tai
l.
List = [1, 2, 3, 4, 5],
Head = 1,
Tail = [2, 3, 4, 5],
HeadOfTail = 2,
TailOfTail = [3, 4, 5].
```

3.

contains1(Element, [Element | _]).

Test queries:

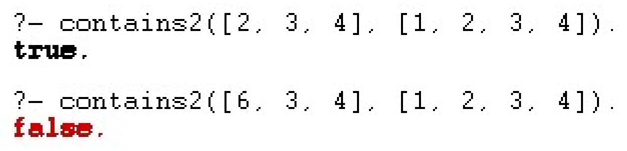


```
?- contains1(6, [6, 5, 4]).
true.

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

4.

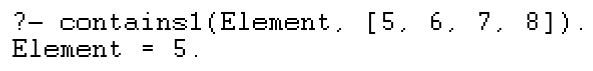
contains2(List, [_ | List]).

Test queries:



```
?- contains2([2, 3, 4], [1, 2, 3, 4]).
true.

?- contains2([6, 3, 4], [1, 2, 3, 4]).
false.
```

5.



```
?- contains1(Element, [5, 6, 7, 8]).
Element = 5.
```

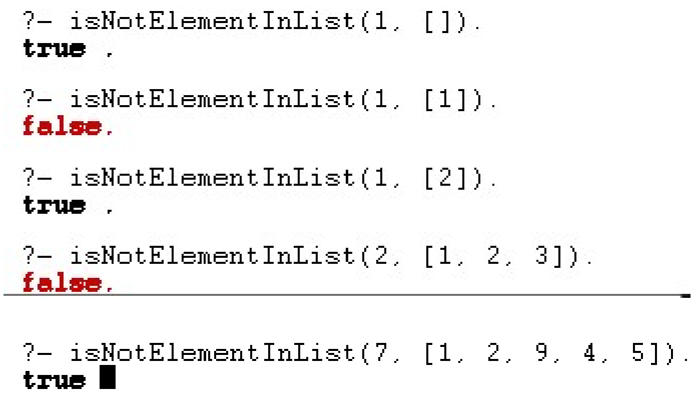
## Question 3

Base case: If the list is empty, the element is not in the list.

Recursive step: Check if the element is not equal to the head of the list, intersect that result with the result from recursively checking the rest of the list.

```
isNotElementInList(_, []).

isNotElementInList(El, [Head | Tail]) :-
    El \= Head,
    isNotElementInList(El, Tail).
```

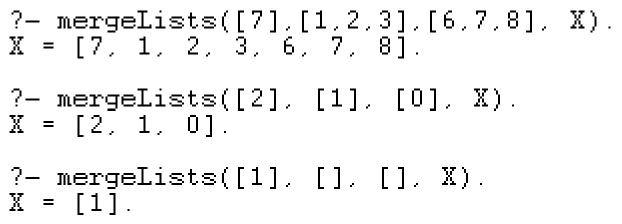
Test queries:



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

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

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

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

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

## Question 4

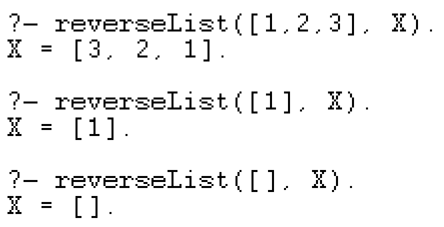
```
mergeLists([], List2, List3, Merged) :-
   mergeListsHelper(List2, List3, Merged).
mergeLists([H|T], List2, List3, [H|MergedTail]) :-
   mergeLists(T, List2, List3, MergedTail).

mergeListsHelper([], L, L).
mergeListsHelper([H|T], List2, [H|MergedTail]) :-
   mergeListsHelper(T, List2, MergedTail).
```



```
?- 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

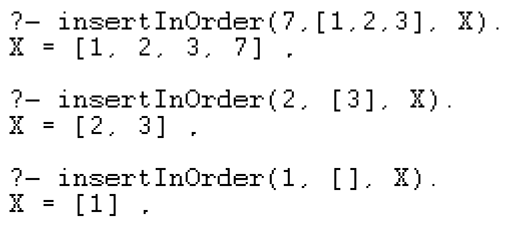
```
reverseList(L, R):-
reverse2(L, [], R).
reverse2([], R, R).
reverse2([H|T], Temp, R) :-
reverse2(T, [H|Temp], R).
```



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

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

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

## Question 6

```
insertInOrder(El, [], [El]).
insertInOrder(El, [H|T], [El, H|T]) :- El =< H.
insertInOrder(El, [H|T], [H|NewTail]) :-
   El > H,
   insertInOrder(El, T, NewTail).
```



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

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

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