1. Given the following set of clauses:

   eats(alf, cats).
eats(mary, cheese).
eats(mary, bread).

List all answers generated for the following queries
   a. ?- eats(mary,cheese).
   b. ?- eats(mary,cats).
   c. ?- eats(mary,X).
   d. ?- eats(X,cats).
   e. ?- eats(X,alf).
   f. ?- eats(X,Y).

2. Given the following set of clauses:

   travel(X) :- on_vacation(X), has_money(X).
   on_vacation(mary).
   on_vacation(peter).
   has_money(peter).

   a. List all answers generated for ?- on_vacation(X).
   b. List all answers generated for ?- travel(X).
   c. Draw the Prolog search tree for travel(X).
   d. Draw the Prolog clause tree for travel(peter).

3. Given the following set of clauses:

   foo([X], X).
   foo ([_|T],X) :- foo (T,X).

   a. ?- foo([1],1).
   b. ?- foo([3],1).
   c. ?- foo([1,2,3],1).
   d. ?- foo([1,2,3],3).
   e. ?- foo([1,2,3],X).
   f. ?- foo([X,2,3],1).
   g. ?- foo([1,2,X],1).
   h. ?- foo([1,2|X],1).
4. Given a set of facts of form \( \text{parent}(\text{name}1, \text{name}2) \) where (\text{name}1 is the parent of \text{name}2):

   a. Define a predicate sibling(\text{X}, \text{Y}) which holds iff \text{X} and \text{Y} are siblings.
   b. Define a predicate cousin(\text{X}, \text{Y}) which holds iff \text{X} and \text{Y} are cousins.
   c. Define a predicate grandchild (\text{X}, \text{Y}) which holds iff \text{X} is a grandchild of \text{Y}.
   d. Define a predicate descendent(\text{X}, \text{Y}) which holds iff \text{X} is a descendent of \text{Y}.

5. Consider the following genealogical tree (and its graphical representation):

<table>
<thead>
<tr>
<th>Genealogical Tree</th>
<th>Graphic Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{parent}(a,b).</td>
<td>a</td>
</tr>
<tr>
<td>\text{parent}(a,c).</td>
<td>/ \</td>
</tr>
<tr>
<td>\text{parent}(b,d).</td>
<td>b c</td>
</tr>
<tr>
<td>\text{parent}(b,e).</td>
<td>/ \ 1</td>
</tr>
<tr>
<td>\text{parent}(c,f).</td>
<td>d e f</td>
</tr>
</tbody>
</table>

   List all answers generated by your definitions for the following queries:

   a. \( ?- \text{ sibling}(\text{X}, \text{Y}). \)
   b. \( ?- \text{ cousin}(\text{X}, \text{Y}). \)
   c. \( ?- \text{ grandchild}(\text{X}, \text{Y}). \)
   d. \( ?- \text{ descendent}(\text{X}, \text{Y}). \)

6. Given the following set of clauses:

   \[
   \begin{align*}
   \text{jedi}(\text{luke}). \\
   \text{jedi}(\text{yoda}). \\
   \text{sith}(\text{vader}). \\
   \text{sith}(\text{maul}). \\
   \text{fight}(\text{X}, \text{Y}) & : - \text{jedi}(\text{X}), \text{sith}(\text{Y}). \\
   \text{fight}(\text{X}, \text{Y}) & : - \text{sith}(\text{X}), \text{X} \neq \text{Y}, \text{sith}(\text{Y}). \\
   \text{fight}(\text{X}, \text{Y}) & : - \text{jedi}(\text{X}), !, \text{jedi}(\text{Y}).
   \end{align*}
   \]

   List all answers generated for the following queries

   a. \( ?- \text{ fight}(\text{luke}, \text{yoda}). \)
   b. \( ?- \text{ fight}(\text{luke}, \text{vader}). \)
   c. \( ?- \text{ fight}(\text{vader}, \text{yoda}). \)
   d. \( ?- \text{ fight}(\text{vader}, \text{maul}). \)
   e. \( ?- \text{ fight}(\text{luke}, \text{X}). \)
   f. \( ?- \text{ fight}(\text{vader}, \text{X}). \)
   g. \( ?- \text{ fight}(\text{X}, \text{yoda}). \)
   h. \( ?- \text{ fight}(\text{X}, \text{maul}). \)
   i. \( ?- \text{ fight}(\text{X}, \text{Y}). \)
7. Given the following set of clauses, what is the output for foo([3,1,2,0],R), if any?

<table>
<thead>
<tr>
<th>Part</th>
<th>Code</th>
<th>Answer</th>
</tr>
</thead>
</table>
| A    | foo([H|_], H).  
      | foo([_|T],X) :- foo(T,X). |        |
| B    | foo([_|T],X) :- foo(T,X).  
      | foo([H|_], H). |        |
| C    | foo([H|_], H) :- H > 1.  
      | foo([_|T],X) :- foo(T,X). |        |
| D    | foo([_|T],X) :- foo(T,X).  
      | foo([H|_], H) :- H > 1. |        |
| E    | foo([H|_], H) :- H > 1, !.  
      | foo([_|T],X) :- foo(T,X). |        |
| F    | foo([_|T],X) :- foo(T,X).  
      | foo([H|_], H) :- H > 1, !. |        |
| G    | foo([H|_], H).  
      | foo([_|T],X) :- X > 1, foo(T,X). |        |
| H    | foo([_|T],X) :- X > 1, foo(T,X).  
      | foo([H|_], H). |        |
| I    | foo([H|_], H).  
      | foo([_|T],X) :- foo(T,X), X > 1. |        |
| J    | foo([_|T],X) :- foo(T,X), X > 1.  
      | foo([H|_], H). |        |
| K    | foo([H|_], H).  
      | foo([_|T],X) :- foo(T,X), !, X > 1. |        |
| L    | foo([_|T],X) :- foo(T,X), !, X > 1.  
      | foo([H|_], H). |        |

8. Define a predicate reverse(L,K) which holds if and only if the list K is the reverse of the list L.

9. Define a predicate add_up_list(L,X) which, given a list of integers L, returns a list of integers in which each element is the sum of all the elements in L up to the same position. Example:

```
?- add_up_list([1,2,3,4],X).
X = [1,3,6,10].
```
10. Consider the following Prolog predicate definition

```prolog
remove_at(X,[X|Xs],1,Xs).
remove_at(X,[Y|Xs],K,[Y|Ys]) :- K1 is K - 1, remove_at(X,Xs,K1,Ys).
```

It works for queries like

?- remove_at(X,[a,b,c,d],2,R).
X = b
R = [a,c,d].

However, it throws an exception for queries like

?- remove_at(c,[a,b,c,d],V,R).
ERROR: remove_at/4: Arguments are not sufficiently instantiated

Modify the predicate definition to make it work for the above query.

11. Write the prolog predicate `flatten(L,R)` that flattens a list of lists in L to a single list R.

The equivalent OCaml function is given by

```ocaml
let rec flatten l = match l with
| [] | [[]] -> []
| [ ]::t -> flatten t
| [h]::t -> h::flatten t
| ((h1::t1)::t) -> h1::flatten(t1::t);;
```