Consider the following datatype representing mathematical expressions:

```
data Exp = Exp :+: Exp
    | Exp :*: Exp
    | Exp :-: Exp
    | Neg Exp
    | Val Int
    | Var String
```

The operators represent addition, multiplication, subtraction and negation respectively. Val is used for numeric values, while Var is used for variables.

a) Write a function which, given a variable name and an expression, returns whether the expression contains only instances of the given variable.

b) An expression is said to be linear in a variable x if it contains no instance of x multiplied by itself. Write a program, which given an expression and a variable name, returns whether the expression is linear in the given variable.

c) Define a type class supporting the constant values zero and one, a unary operator neg, and two binary operators conj and disj. Make Bool an instance of the class interpreting zero as false, one as true, neg as not and conj and disj as conjunction and disjunction respectively. Make Exp an instance of the class interpreting zero and one as the constants 0 and 1, neg as negation, conj as multiplication and disj as addition.
2. Consider the following datatype representing 3-ary trees:

```haskell
data Tree3 a
    = Node a (Tree3 a, Tree3 a, Tree3 a)
    | Null
```

a) Write a function which counts the number of nodes in a 3-ary tree.
b) Write a function which, given a 3-ary tree, and two values x and y, replaces all the data in nodes containing value x with value y. Give the type of the function.
c) Give the type of the following function:

```haskell
switch Null = Null
switch (Node v (tl, tm, tr)) = Node v (tr, switch tm, tl)
```
d) Write a function, which given two 3-ary trees, returns whether their structure matches (in other words, the data may be different, but not the positions of nodes and nulls).

Section B

3. Write clauses for the following list handling rules:
   a. `insert(Element,OldList,NewList)` which inserts `Element` in the `OldList` to produce the `NewList`.
      e.g. `insert(8,[1,2,3],NL)` binds NL to `[8,1,2,3]`
          [10]
   b. `right(List,Element)` which binds `Element` to the rightmost member within the `List`.
      e.g. `right([riccardo, doris, vanessa, roberta],R)` binds R to ‘roberta’
          [15]

4. The **Cartesian Product** of two sets is the set containing all the tuples produced from one element of one set and second element from the other set. Consider the sets $A = \{a, b, c\}$ and $B = \{1, 2, 3\}$
   
   The cartesian product of A and B = $A \times B$
   
   = \{(a,1), (a,2), (a,3), (b,1), (b,2), (b,3), (c,1), (c,2), (c,3) \}

   Write a Prolog program in which it is possible to issue goals to display all the tuples produced from the Cartesian Product of any two sets, including the empty set.
   e.g. `cartesian_product ([a,b,c],[1,2,3])` would display:

<table>
<thead>
<tr>
<th>a</th>
<th>a 2</th>
<th>a 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b 1</td>
<td>b 2</td>
<td>b 3</td>
</tr>
<tr>
<td>c 1</td>
<td>c 2</td>
<td>c 3</td>
</tr>
</tbody>
</table>
5. Write recursive clauses for the following predicates:

(a) **deduct(List_of_Integers, Number, Result)** which binds **Result** to the value of the **Number** after that each element within the **List_of_Integers** are deducted from it.
   e.g. `deduct([3,2,5], 20, Ans)` binds Ans to 10.  

(b) **display_items(List_of_Items)** which displays a sequence of the items within the **List_of_Items** from left to right.
   e.g. `display_items([max, $, 2, lca648])` displays `max $ 2 lca648`  

(c) **push(Item, List, Stack)** which binds **Stack** to **List** with the **Item** entered at its head.
   e.g. `push(a, [b,c], NewStack)` binds NewStack to `[a,b,c]`.  