1. (Finite State Machine Design; 15 points)
   (a) (7 points) Design a finite state machine to accept a binary string
       which does not contain three consecutive identical symbols. So,
       0010011110101 will be accepted whereas 1110 will be rejected.
   (b) (8 points) Design a finite state machine whose input alphabet is the
       roman alphabet, A; it accepts any string which contains the substring
       “rare” (a substring is a consecutive sequence of characters).

2. (Reasoning about Finite State Machines; 20 points) This question is about
   designing a finite state transducer and proving its correctness.
   (a) (4 points) Design a finite state transducer that outputs every symbol
       of the input string which is at an even numbered position (where the
       first symbol is at position 0). Thus, with input \( \epsilon \), the output is \( \epsilon \),
       with input string consisting of just symbol \( a \), the output is \( a \), and
       with input string being \( abcd \) the output is \( ac \).
   (b) (5 points) Define the function computed above, define the output
       string as a function of the input string. Start with \( f(\epsilon) = \epsilon \).
   (c) (6 points) Annotate the states of your machine with predicates in
       order to prove that the machine computes \( f \). Use symbols \( x \) and \( y \)
       for the input and output strings at a state.
   (d) (5 points) Using the annotation predicates, write the theorems that
       need to be proved to show that the machine computes \( f \). You don’t
       have to actually prove the theorems.

3. (Writing Recursive Programs; 30 points) Write Haskell programs for the
   following problems. You may have to code additional helper functions.
   (a) (6 points) The result of a class test is kept as a non-empty list of
       pairs, (name, score). Write a function that computes the list of stu-
       dents whose scores are 90 or above, the list of students whose scores
       are 80 or above but below 90, and the list of remaining students.
       Here is a possible output.

       ```haskell
       Main> grade [("x",30),("y",92),("z",80),("p",90)]
       ([("y","p"),["z"],["x"]])
       ```
   (b) (6 points) Compute all suffixes of a given string. Here is a possible
       output.

       ```haskell
       Main> suffix "abc"
       ["abc","bc","c",""]
       ```
(c) (8 points) Given two non-empty lists \( L \) and \( R \) form their cartesian product; i.e., create a list of pairs with the first element from \( L \) and the second from \( R \), for all elements of \( L \) and \( R \). The order of elements in your output is immaterial. Here is a possible output.

Main> cart [2,1] [3,5]
[(2,3),(2,5),(1,3),(1,5)]

You may use concatenation of lists, ++, for this problem.

(d) (10 points) Consider strings consisting solely of left and right parentheses. Strings that are properly nested, such as \( () \), \( (((())))) \), and \( (((())))) \), are called balanced. Strings \( )() \), \( (()()) \), and \( (((())))) \) are unbalanced.

It is known that a string is balanced if and only if (1) at every point in the string the number of left paren ≥ the number of right paren counting from the left end of the string, and (2) at the very end, the number of left paren = the number of right paren. Below, the left paren count − the right paren count at every point is shown (as superscript) for two strings.

\[
\begin{array}{c|c}
\text{String} & \text{Left paren count - Right paren count} \\
\hline
() & 0 \text{ (balanced)} \\
(()) & 1 \text{ (balanced)} \\
(()()) & 2 \text{ (balanced)} \\
\end{array}
\]

Write a program that outputs True if the argument string is balanced, and False otherwise. Recall that a string is a list of characters. Your program should scan the string from left to right computing the left paren count − the right paren count incrementally.

4. (Properties of Recursive Programs; 10 points) The notes contain a program to right_rotate a list using the function rev. We can similarly write a function for left_rotate. I reproduce right_rotate from the notes and give the code for left_rotate. I abbreviate right_rotate and left_rotate by rr and lr, respectively.

\[
\begin{align*}
\text{rr} \; \text{[]} &= \; \text{[]} \\
\text{rr} \; \text{xs} &= y: (\text{rev} \; \text{ys}) \\
&\quad \text{where } y:ys = (\text{rev} \; xs) \\
\end{align*}
\]

\[
\begin{align*}
\text{lr} \; \text{[]} &= \; \text{[]} \\
\text{lr} \; (x:xs) &= xs ++ [x] \\
\end{align*}
\]

Prove that \( \text{lr} \) followed by \( \text{rr} \) has no effect, i.e., \( \text{rr}(\text{lr} \; \text{xs}) = \text{xs} \).

You will need the following facts for the proof

\[
\begin{align*}
\text{rev} \; [x] &= [x] \\
\text{rev}(\text{rev} \; \text{xs}) &= \text{xs} \\
\text{rev} \; (\text{xs} ++ \text{ys}) &= (\text{rev} \; \text{ys}) ++ (\text{rev} \; \text{xs}) \\
[x] ++ \text{xs} &= x:xs \\
\end{align*}
\]