Answer two questions from section A, and from section B answer question 4 and one of 5 and 6. Students are allowed to use course notes, books and calculators.
Section B: Functional Programming

4. (a) Consider the following data type describing expressions:

\[
type \text{VarName} = \text{String}
\]
\[
data \text{Exp} = \begin{array}{c}
\text{Val} \quad \text{Int} \\
| \quad \text{Var} \quad \text{VarName} \\
| \quad \text{Exp} \, :+: \, \text{Exp} \\
| \quad \text{Exp} \, :*:\, \text{Exp}
\end{array}
\]

Define a function \(\text{ops}\), which given an expression returns the total number of additions and multiplications which appear in the expression.

\[
\text{ops} :: \text{Exp} \rightarrow \text{Int}
\]

(b) Now consider the following data type describing integers, with the addition of positive infinity:

\[
data \text{Inf} = \begin{array}{c}
\text{Finite} \quad \text{Int} \\
| \quad \text{Infinite}
\end{array}
\]

Define infix addition \(\leftrightarrow\) and multiplication \(\leftrightarrow\) operators on the datatype \(\text{Inf}\). Recall that adding any number to infinity will result in infinity. Similarly, multiplying any number by infinity will also return infinity. Make \(\text{Inf}\) an instance of \text{Show}. Finite numbers will just be shown without any further adornment (\text{Finite} 7 will be shown as 7), while infinite numbers will be shown as \text{<inf>}.  

(c) Now consider the following datatype defining programs in an imperative language:

\[
data \text{Prg} = \begin{array}{c}
\text{Skip} \\
| \quad \text{VarName} \, := \, \text{Exp} \\
| \quad \text{Prg} \, ::= \, \text{Prg}
\end{array}
\]
```haskell
| While Exp Prg
| For (VarName, Int, Int) Prg

Skip does nothing, while := is the assignment operator, ::: is sequential composition, While is a while loop and For is a for loop with the given variable looping with values starting from the first number till the second.

The worst-case complexity of a program is defined as follows:

- At worst, Skip takes 1 time unit to execute.
- At worst, v := e takes \( n \) time units to execute, where \( n \) is the number of operators in expression e.
- At worst, p ::: q takes the sum of the worst case of p and of q.
- At worst, For (v, n0, n1) p takes \( n \times w \) time steps, where \( n \) is the one more than the difference between n1 and n0, and \( w \) is the worst case complexity of p. You may assume that \( n2 \) is always bigger or equal to n1.
- Since while loops may never terminate, the worst case complexity of a while loop is \( \infty \).

Using the above definitions, define the function complexity, which given a program Prg, returns an Inf, equal to the worst case complexity of the program.

(d) What do the following functions do? What are their types?

```haskell
  - veni :: [a] -> [a]
  - veni (x:xs) |
    | x `elem` xs = veni xs
    | otherwise = x:veni xs

  - vedi :: Expr -> [a]
  - vedi (Val _) = []
  - vedi (Var v) = [v]
  - vedi (e :+: f) = veni (vedi e ++ vedi f)
  - vedi (e :*: f) = veni (vedi e ++ vedi f)

  - vici :: Exp -> [a]
  - vici Skip = []
  - vici (v := e) = veni (v:vedi e)
```

```
\[ \text{vici (p :> q)} = \text{veni (vici p ++ vici q)} \]
\[ \text{vici (While e p)} = \text{veni (vedi e ++ vici p)} \]
\[ \text{vici (For (v, _, _) p)} = \text{veni (v:vici p)} \]

Note that \text{elem x xs} returns whether \( x \) is an element of \( xs \):

\[ \text{elem :: Eq a => a -> [a] -> Bool} \]

5. (a) Define a typeclass \text{Counter}, which includes types \( a \) supporting 
\text{zero} (returning an object of type \( a \)), \text{isZero} (a function which 
takes an object of type \( a \) and returns true or false) and two func-
tions \text{decrement} and \text{increment} (both take an object of type \( a \) 
and return an object of type \( a \)).

(b) Make \text{Int} an instance of \text{Counter}.

(c) Addition is defined as follows:

\[
\begin{align*}
\text{add x y} & \mid \text{isZero x} = y \\
& \mid \text{otherwise} = \text{add} (\text{decrement} x) (\text{increment} x)
\end{align*}
\]

Give the full type of \text{add}.

(d) Another way to encode a counter is a string. A number \( n \) would 
be encoded as \( n \) copies of a symbol \( x \). Make strings an instance 
of \text{Counter}.

6. (a) Using a list comprehension, write a function which calculates the 
multiples of a given number in increasing order.

\[\text{multiples :: Int -> [Int]}\]
(b) Using pattern-matching, write a function which, given two lists of integers (both in increasing order) calculates their intersection.

\[
\text{intersectSorted} :: \text{[Int]} \rightarrow \text{[Int]} \rightarrow \text{[Int]}
\]

(c) Hence or otherwise, write a function to calculate the common multiples of two given integers.

\[
\text{commonMultiples} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{[Int]}
\]

(d) Hence or otherwise, write a function to calculate the least common multiple of two given integers.

\[
\text{lcm} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

[15 marks]