Programming Paradigms
Practical session, Week 6

Department of Informatics
University of Beira Interior

Chapter 8 (continued): Type classes

1. (From previous exercise sheet) Complete the following instance declarations:

   ```
   instance Eq a => Eq (Maybe a) where
   ...
   ```

   and

   ```
   instance Eq a => Eq [a] where
   ...
   ```

2. Consider the following datatype:

   ```
   -- Type of non-empty lists
   data NonEmptyList a = Singular a | NECons a (NonEmptyList a)
   ```

   (a) Provide the instance of `Show` for the datatype of non-empty lists.

   (b) Provide the instance of `Eq` for the datatype of non-empty lists.

   (c) Provide the instance of `Ord` for the datatype of non-empty lists.

   (d) Provide the instance of `Functor` for the datatypes of non-empty lists.

3. Provide an instance of `Num` for lists, so that the operations in `Num` are lifted to lists. For example,

   ```
   [1,2] + [3,4] = [4,6]
   ```

   You can define `fromInteger i` as the list which contains `i` transformed into the appropriate type (use `fromInteger` on the input).

4. Define a class EmptyTestable t that defines a function

   ```
   isEmpty :: t -> Bool
   ```

   Define instances for Ints, Integers, lists, and pairs:

   - For integers, consider zero empty and non-zero non-empty.
   - For lists, consider the empty list empty (e.g. [] is empty).
   - A pair is empty if it is made of two empty values.

Chapter 15 slides: Lazy Evaluation

1. Identify the redexes in the following expressions, and determine whether each redex is innermost, outermost, neither, or both:

   (a) `1 + (2*3)`

   (b) `(1+2) * (2+3)`

   (c) `fst (1+2,2+3)`

   (d) `(\x ->1 + x) (2*3)`

2. Show why outermost evaluation is preferable to innermost for the purposes of evaluating the expression

   ```
   fst (1+2,2+3)
   ```

3. Given the definition

   ```
   mult = \x -> (\y -> x * y)
   ```

   show how the evaluation of `mult 3 4` can be broken down into four separate steps.

4. Using a list comprehension, define an expression

   ```
   fibs :: [Integer]
   ```

   that generates the infinite sequence of Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... using the following simple procedure:

   - the first two numbers are 0 and 1;
   - the next is the sum of the previous two;
   - return to the second step.

   Hint: make use of the library functions `zip` and `tail`. Note that numbers in the Fibonacci sequence quickly become large, hence the use of the type Integer of arbitrary-precision integers above.