Blockchain Lab: Cardano Track
Preliminary Test

1 Part I: Haskell basics

1. **Pattern Matching and Polymorphism**

   (a) For each pattern in the following function definitions, decide if they are well-typed. If yes, indicate the type of each sub-expression. Do the well-typed patterns cover every possible value of the corresponding type?

   Example:

   ```
   f :: (a,b) -> X
   f (a,b) = ...
   ```

   It's well typed. Subexpresions

   - `(a,b) :: (a,b)`
   - `a :: a`
   - `b :: b`

   The pattern covers every possible type.

   ```
   f :: (a,b) -> X
   f x = ...
   ```

   ```
   f :: (a,b) -> X
   f (x,y) = ...
   ```

   ```
   f :: [(a,b)] -> X
   f (a,b) = ...
   ```

   ```
   f :: [(a,b)] -> X
   f (x:xs) = ...
   ```

   ```
   f :: [(a,b)] -> X
   f ((x,y):((a,b):xs)) = ...
   ```

   ```
   f :: [(Int,a)] -> X
   f [(0,a)] = ...
   ```

   ```
   f :: [(Int,a)] -> X
   f ((x,1):xs) = ...
   ```

   ```
   f :: [(Int,a)] -> X
   f ((1,x):xs) = ...
   ```

   ```
   f :: (Int -> Int) -> Int -> X
   f a b = ...
   ```

   ```
   f :: (Int -> Int) -> Int -> X
   f a 3 = ...
   ```

   ```
   f :: (Int -> Int) -> Int -> X
   f 0 1 2 = ...
   ```

   ```
   f :: Int -> Int -> Int -> X
   f 0 1 2 = ...
   ```

   ```
   f :: Int -> Int -> Int -> X
   f 0 g = ...
   ```

   ```
   f :: a -> (a -> a) -> X
   f 0 g = ...
   ```

   (b) Give a definition (different to `undefined`) for each function declaration, if it's possible. There exists another different definition?

   ```
   f :: (a,b) -> b
   ```

   ```
   f :: (a,b) -> c
   ```

   ```
   f :: a -> b
   ```

   ```
   f :: (a -> b) -> a -> b
   ```

   ```
   f :: (a -> b) -> [a] -> [b]
   ```

   ```
   f :: (a -> b) -> a -> c
   ```

   ```
   f :: (a -> b) -> (b -> c) -> a -> c
   ```

   ```
   f :: (a -> b) -> (b -> c) -> [a] -> [c]
   ```

2. **Function definitions**

   Using library functions, define a function `optimizeList :: Eq a => [ a ] -> [ (Int, a) ]` that tries to optimize a list with many repetitions by compacting the same information into a new list. For example:

   ```
   > optimizeList  "aabbbcaadgggzzzzzzzzzzzzzzzzzzzzzzzzzz"
   [(2,'a'),(3,'b'),(1,'c'),(2,'a'),(1,'d'),(3,'g'),(25,'z')]
   ```

   Hint: Think about using `takeWhile`.

3. **Declaring and using types**

   (a) Consider the following type of binary trees:

   ```
   data Tree = Leaf Int | Node Tree Tree
   ```

Let us say that such a tree is balanced if the number of leaves in the left and right subtree of every node differs by at most one, with leaves themselves being trivially balanced. Define a function `balanced :: Tree -> Bool` that decides if a tree is balanced or not.

Hint: First define a function that returns the number of leaves in a tree.

(b) Define a function `balance :: [Int] -> Tree` that converts a non-empty list of integers into a balanced tree.

4. **Laziness**

Consider the previous data type and given the following functions

```
goLeft :: Tree -> Tree
goLeft (Leaf _) = error "leaf"
goLeft (Node t _) = t
```

```
flatten :: Tree -> [Int]
flatten (Leaf i)    = [i]
flatten (Node t t') = flatten t' ++ flatten t
```

Define a `Tree` value, let's call it `infLeftTree`, for which we can do an arbitrary number of `goLeft`s without throwing an error. Is it even possible? If yes, why does it work `head (flatten infLeftTree)`?

2 Part II: File System simulator

Lets consider a very simple le system simulator, where it's possible to represent directories and les. The design is explained in the `README` file.

Complete the `FileSystem.hs` module, replacing every `undefined` function. You could add auxiliary functions, but the datatypes and the module exports shouldn't be modied.