DM580 – EXERCISE SHEET #7

(1) Recall the type of natural numbers.

```
data Nat = Zero
         | Succ Nat
```

   (a) Write a function `multiply :: Nat -> Nat -> Nat` that multiplies two natural numbers. (You may use the function from Lecture 7 for adding natural numbers as a helper, or write your own.)

   (b) Write a function `exponentiate :: Nat -> Nat -> Nat` such that `exponentiate n m` computes n^m for all natural numbers n, m.

(2) Write a function `extract :: [Maybe a] -> [a]` which extracts the value of type `a` out of each non-`Nothing` element in a list.

(3) Write a function `altmap :: (a -> b) -> (a -> b) -> [a] -> [b]` that alternates between applying two functions to a list, e.g.,

```
altmap (+10) (+100) [1,2,3,4,5] == [11,102,13,104,15]
```

(4) Consider the type of binary trees.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

   (a) Write a function `treemap :: (a -> b) -> Tree a -> Tree b` that applies the given function to the data inside each leaf in a tree.

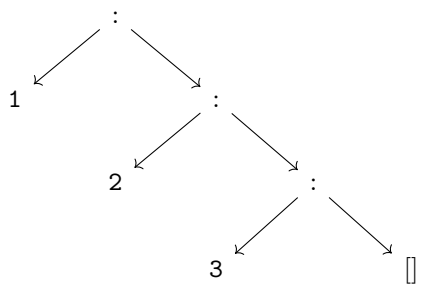
   (b) Argue that `treemap id == id`, where `id :: a -> a` is the identity function (defined in the prelude as `id x = x`)

   (c) Argue that `(treemap f) . (treemap g) == treemap (f . g)` for all possible functions `g :: b -> c` and `f :: a -> b`.

   (d) Write a function `treeconcat :: Tree (Tree a) -> Tree a` that flattens a tree with more trees as their leaves into a tree.

   (e) Write a function `treeapply :: Tree (a -> b) -> Tree a -> Tree b` that returns the tree with data of type `b` at its leaves produced by replacing each leaf in the function tree by the function at the leaf mapped over the tree with data of type `a`.

(5) A list can be seen as a particular kind of tree, it's *syntax tree*. For example, the list `[1,2,3]` is syntactic sugar for `1:(2:(3:[]))`, which corresponds to the tree



Write a function `list2tree :: [a] -> Tree (Maybe a)` that represents a list as a binary tree.