Programming I: Functional Programming in Haskell
Solutions to Unassessed Exercises

Set 5: User-defined data types

1. ```
   toNum :: Maybe a -> Int
   toNum
     = maybe 0 (const 1)
   ```

2. ```
   filterMaybe :: (a -> Bool) -> Maybe a -> Maybe a
   filterMaybe p m@(Just x)
     | p x = m
   filterMaybe p _
     = Nothing
   ```

3. ```
   data Shape = Triangle Float Float Float |
              Square Float |
                Circle Float

   area :: Shape -> Float
   area (Triangle a b c)
     = triangleArea a b c
   area (Square d)
     = d * d
   area (Circle r)
     = pi * r ^ 2

   triangleArea a b c
     = sqrt (s * (s - a) * (s - b) * (s - c))
     where
       s = (a + b + c) / 2
   ```

4. ```
   data Shape = Triangle Float Float Float |
                Square Float |
                Circle Float |
              Polygon [Vertex]

   type Vertex = (Float, Float)

   area :: Shape -> Float
   ... -- As above
   area (Polygon vs)
     = polygonArea vs

   polygonArea :: [Vertex] -> Float
   polygonArea (v1 : v2 : v3 : vs)
     = triangleArea a b c + polyarea (v1 : v3 : vs)
   ```

```
  where
    a = dist v1 v2
    b = dist v2 v3
    c = dist v3 v1
    dist (x, y) (x', y')
      = sqrt ((x - x') ^ 2 + (y - y') ^ 2)
polyarea vs
  = 0
```

5.
```
type Date = (Int, Int, Int)

age :: Date -> Date -> Int
age (d, m, y) (d', m', y')
  | (m, d) <= (m', d') = y' - y
  | otherwise          = y' - y - 1
```

6.
```
flatten t
  = flatten' t []
  where
    flatten' Empty xs
      = xs
    flatten' (Node t1 x t2) xs
      = flatten' t1 (x : flatten' t2 xs)
```

7.
```
makeTrees :: Int -> [Tree]
makeTrees n
  | n == 0    = [Leaf]
  | otherwise = [Node l r | (t, t') <- map makeTrees' [0..n-1],
                            l <- t, r <- t']
  where
    makeTrees' k = (makeTrees k, makeTrees (n - k - 1))
```

Note: this recomputes the same list of trees many times. An alternative is to remember them in a list, `ts` say:

```
makeTrees n
  = makeTrees' n [[Leaf]]
  where
    makeTrees' k ts
      | k == 0    = head ts
      | otherwise = makesTrees' (k - 1) (ts' : ts)
      where
        ts' = [Node l r | (t, t') <- zip ts (reverse ts),
                          l <- t, r <- t']
```

`ts` contains the list of all lists of trees of sizes `n-k .. 0`. The `zip` call pairs up trees whose combined size is `n-k`. This saves a lot of repeated computation at the expense of a rather large data structure.

8.
```
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show
```

(a)
```
build :: [a] -> Tree a
build []
  = error "empty tree"
build [x]
```

```haskell
    = Leaf x
  build xs
    = Node (build left) (build right)
    where
      (left, right) = splitAt (length xs `div` 2) xs
```

(b)
```haskell
ends :: Tree a -> [a]
ends (Leaf a)
  = [a]
ends (Node left right)
  = ends left ++ ends right
```

(c)
```haskell
swap :: Tree a -> Tree a
swap (Leaf a)
  = Leaf a
swap (Node left right)
  = Node (swap right) (swap left)
```

`ends (swap (build xs))` is the same as `reverse xs`, i.e. `ends . swap . build` $\equiv$ `reverse`.

9.
```haskell
data Tree a b = Node a (Tree a b) (Tree a b) |
                Leaf b |
                Empty
```

(a)
```haskell
mapT :: (a -> a) -> (b -> b) -> Tree b a -> Tree b a
mapT f g Empty
  = Empty
mapT f g (Leaf x)
  = Leaf (f x)
mapT f g (Node x t1 t2)
  = Node (g x) (mapT f g t1) (mapT f g t2)
```

(b)
```haskell
foldT :: (a -> b) -> (c -> b -> b -> b) -> b -> Tree c a -> b
foldT leafFun nodeFun b Empty
  = b
foldT leafFun nodeFun b (Leaf x)
  = leafFun x
foldT leafFun nodeFun b (Node x t1 t2)
  = nodeFun x (foldInto t1) (foldInto t2)
  where
    foldInto = foldT leafFun nodeFun b
```

(c) In the following we'll use anonymous functions (lambda expressions) but you can just as well use named functions, e.g. in a `where` clause.

i.
```haskell
countLeaves :: Tree a b -> Int
countLeaves
  = foldT (const 1) (\x -> (+)) 0
```
Note that the prelude function, `const x y = x`, throws away its second argument. Note also that the lambda expression above will be given three arguments by `foldT`; the first isn't needed as we're only counting leaves, but the second and third need to be added together. It looks a bit odd, but it has exactly the same meaning as:
```haskell
(\x -> \left -> \right -> left + right)
```

ii.
```haskell
sum :: Tree Int Int -> Int
sum
  = foldT id (\n -> \x -> \y -> n + x + y) 0
```

iii. `flattenLR :: Tree a a -> [a]`

```
flattenLR
  = foldT (\l -> [l]) (\n -> \x -> \y -> x ++ [n] ++ y) []
```

iv. `flattenRL :: Tree a a -> [a]`

```
flattenRL
  = foldT (\l -> [l]) (\n -> \x -> \y -> y ++ [n] ++ x) []
```

v. `eval :: Tree (Int -> Int -> Int) Int -> Int`

```
eval
  = foldT id id 0
```