```
import Data.List
import Test.QuickCheck


-- For general grading criteria, see the last two pages of the PDF



-- Question 1
myscanl :: (b -> a -> b) -> b -> [a] -> [b]
myscanl f y []     = y : []
myscanl f y (x:xs) = y : myscanl f (f y x) xs

prop_myscanl :: Integer -> [Integer] -> Bool
prop_myscanl x xs = myscanl (-) x xs == scanl (-) x xs

-- Note:
-- the type must be generic (not limited to numbers only)

-- Question 2

cycle' :: [a] -> [a]
cycle' xs = [ x | y<-[1..], x <- xs]

prop_cycle :: Small Int -> NonEmptyList Integer -> Bool
prop_cycle (Small n) (NonEmpty xs) = take n (cycle xs) == take n
(cycle' xs)

-- Question 3

find' :: (a -> Bool) -> [a] -> Maybe a
find' f = foldr (\x y -> if f x then (Just x) else y) Nothing

prop_find xs = find (<0) xs == find' (<0) xs

-- Question 4

zipFiles :: FilePath -> FilePath -> IO [(String,String)]
zipFiles f1 f2 = zip <$> (lines <$> readFile f1) <*> (lines <$>
readFile f2)
```

```haskell
-- Question 5

maximum' :: Ord a => [a] -> a
maximum' = head . reverse . sort

prop_maximum :: NonEmptyList Integer -> Bool
prop_maximum (NonEmpty xs)= maximum xs == maximum' xs

-- Question 6

mzip :: [Maybe a] -> [Maybe b] -> [(a,b)]
mzip _            []          = []
mzip []           _           = []
mzip (Just x:xs)  (Just y:ys) = (x,y) : mzip xs ys
mzip (x:xs) (y:ys)            = mzip xs ys

-- Question 7
-- a

data Tree a = Node a (Tree a) (Tree a)
            | Leaf
            deriving (Show, Eq, Ord)

-- b
treeins :: Ord a => a -> Tree a -> Tree a
treeins x (Node a t1 t2) | x < a     = Node a (treeins x t1) t2
                         | otherwise = Node a t1 (treeins x t2)
treeins x Leaf = Node x Leaf Leaf


-- c
list2tree :: Ord a => [a] -> Tree a
list2tree xs = foldr treeins Leaf xs

-- d
tree2list Leaf = []
tree2list (Node x t1 t2) = tree2list t1 ++ [x] ++ tree2list t2

-- e
listsort :: Ord a => [a] -> [a]
listsort = tree2list . list2tree
```

```
prop_listsort :: [Int] -> Bool
prop_listsort xs = listsort xs == sort xs

-- Question 8
-- a

data Stream a = Stream a (Stream a)

-- b

toList (Stream x xs) = x : toList xs

-- c
instance Functor Stream where
  fmap f (Stream x xs) = Stream (f x) (fmap f xs)

-- d

-- No: as a consequence of the Functor laws, Functors are uniquely
defined

-- (on the exam, we expect compact answers)

-- e

instance Applicative Stream where
  pure x = Stream x (pure x)
  (Stream f fs) <*> (Stream x xs) = Stream (f x) (fs <*> xs)

-- f

zipstream :: Stream a -> Stream b -> Stream (a,b)
zipstream x y  = (,) <$> x <*> y


-- Question 9

ones = 1 : ones

r = 1 : zipWith (+) r ones
```