> The objective of this exam is to test your understanding of week 7 of the CIS 194 Spring 2013 course (folds and monoids).

Name: Andrés Martínez

1. (1 point) The `foldr` function, applied to a binary operator, a starting value, and a list, reduces the list using the binary operator, from right to left:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []     = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

The `filter` function, applied to a predicate and a list, returns the list of those elements that satisfy the predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _    []     = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs
```

Can you define `filter` in terms of `foldr`?

```
filter' :: Foldable t => (a -> Bool) -> t a -> [a]
filter' f xs = foldr (\y ys -> if f y then y:ys else ys) [] xs
```

2. The `foldl` function, applied to a binary operator, a starting value, and a list, reduces the list using the binary operator, from left to right.

(a) (1 point) Complete the definition of `foldl`:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     = acc
foldl f acc (x:xs) =
```

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f acc [] = acc
foldl' f acc (x:xs) = foldl f (f acc x) xs
```

(b) (1 point (bonus)) What is the difference between the following expressions?

- `foldr (+) 0 [1..5]`
- `foldl (+) 0 [1..5]`

```
foldr (\x y -> concat ["(",x,"+",y,")"]) "0" (map show [1..5])
"(1+(2+(3+(4+(5+0)))))"

foldl (\x y -> concat ["(",x,"+",y,")"]) "0" (map show [1..5])
"(((((0+1)+2)+3)+4)+5)"
```

3. (1 point) A monoid is a type with an associative binary operation that has an identity:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Instances of `Monoid` should satisfy the following laws:

- `mappend mempty x = x`
- `mappend x mempty = x`
- `mappend x (mappend y z) = mappend (mappend x y) z`

Remember the `Maybe` type?

```
data Maybe a = Nothing | Just a
```

Define an instance of `Monoid` for `Maybe a`:

```
instance Monoid a => Monoid (Maybe a) where
 mempty :: maybe a
 mempty = Nothing

 mappend :: Maybe a -> Maybe a -> Maybe a
 mappend x Nothing = x
 mappend Nothing x = x
```