The objective of this exam is to test your understanding of weeks 11 and 12 of the CIS 194 Spring 2013 course (applicative functors and monads).

Name: Andrés Martínez

1. (1 point (bonus)) In week 9, we studied functors, which allow us to map a function over some structure:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

In weeks 10 and 11, we studied applicative functors, which allow us to map a function contained over some structure over some structure:

```
class Functor f => Applicative f where
  pure  :: a -> f a

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

```
instance Applicative Maybe where
  pure = Just

  Nothing <*> _  = Nothing
  Just f  <*> mx = fmap f mx
```

And, in week 12, we studied monads:

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

```
instance Monad Maybe where
  Nothing >>= _ = Nothing
  Just x  >>= k = k x
```

Monads are more powerful than functors and applicative functors because they allow us to remove a level of structure using the `join` function:

```
ghci> join Nothing
Nothing
ghci> join (Just Nothing)
Nothing
```

```
ghci> join (Just (Just 1))
Just 1
ghci> join [[1,2],[],[3,4,5]]
[1,2,3,4,5]
```

Define `join`:

```haskell
join' :: Monad m => m (m a) -> m a
join' mmx = do
    m <- mmx
    id m
```

(Hint: Use `id` and remember that `m` is a `Monad`.)

2. (1 point) The bind operator, `(>>=)`, sequentially composes two actions, passing any value produced by the first as an argument to the second. Write `(>>=)` in terms of `fmap` and `join`:

```haskell
(>>>==) :: Monad m => m a -> (a -> m b) -> m b
mx >>>== k = join' (fmap k mx)
```

3. (1 point) Consider the following function:

```
mapA :: Applicative f => (a -> f b) -> [a] -> f [b]
mapA _ []     = pure []
mapA f (x:xs) = (:) <$> f x <*> mapA f xs
```

What is the result of `mapA Just [1,2,3]`?

A. `[1,2,3]`

B. `Just [1,2,3]`

C. `[Just 1,Just 2,Just 3]`

D. `Nothing`

4. (2 points) Consider the following function:

```
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA = mapA id
```

(a) What is the result of `sequenceA [Just 1,Nothing,Just 3]`?

A. `[1,3]`

B. `Just [1,3]`

C. `[Just 1,Just 3]`

D. `Nothing`

(b) What is the result of `sequenceA [Just 1,Just 2,Just 3]`?

A. `[1,2,3]`

B. `Just [1,2,3]`

C. `[Just 1,Just 2,Just 3]`

D. `Nothing`

5. (1 point) Consider the following functions:

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)

replicateA n fx = sequenceA (replicate n fx)
```

What is the type of `replicateA`?

A. `Functor f => Int -> f a -> f [a]`

B. `Int -> f a -> f [a]`

C. `Applicative f => Int -> f a -> f [a]`

D. `Applicative f => Int -> f a -> [f a]`