> The objective of this exam is to test your understanding of weeks 9 and 10 of the CIS 194 Spring 2013 course (functors and applicative functors).

Name: Andrés Martínez

The `Maybe` type encapsulates an optional value:

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

The `Functor` type class is used for types that can be mapped over:

```
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)
```

Instances of `Functor` should satisfy the following laws:

- `fmap id == id`
- `fmap (g . f) == fmap g . fmap f`

`(<$>)` is an infix synonym for `fmap`:

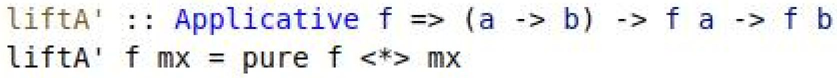
```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

An applicative functor is a functor with application:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

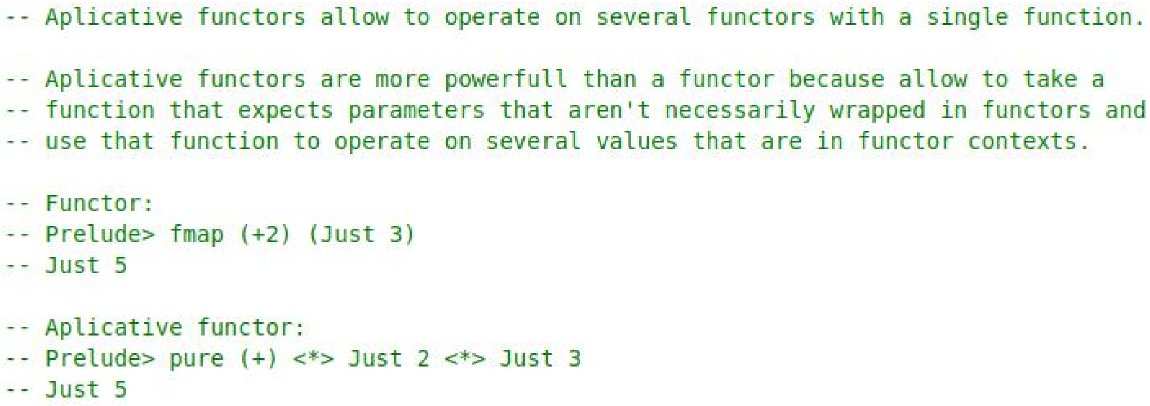
```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _  = Nothing
  Just f  <*> mx = fmap f mx
```

1. (1 point) Every `Applicative` is also a `Functor`, so we can define `fmap` in terms of `pure` and `(<*>)`. Try it:



```
liftA' :: Applicative f => (a -> b) -> f a -> f b
liftA' f mx = pure f <*> mx
```

2. (1 point) What's so special about applicative functors? Why is an applicative functor more powerful than a functor? If you want, you can use the `Maybe` type as an example.



```
-- Aplicative functors allow to operate on several functors with a single function.

-- Aplicative functors are more powerfull than a functor because allow to take a
-- function that expects parameters that aren't necessarily wrapped in functors and
-- use that function to operate on several values that are in functor contexts.

-- Functor:
-- Prelude> fmap (+2) (Just 3)
-- Just 5

-- Aplicative functor:
-- Prelude> pure (+) <*> Just 2 <*> Just 3
-- Just 5
```

3. (1 point) We can represent a book as an author and a title:

```
data Book = Book String String deriving Show
```

Consider the following expressions:

```
maybeAuthor1, maybeAuthor2 :: Maybe String
maybeAuthor1 = Just "Charles Dickens"
maybeAuthor2 = Nothing

maybeTitle1, maybeTitle2 :: Maybe String
maybeTitle1 = Nothing
maybeTitle2 = Just "David Copperfield"
```

Can you match the following expressions with their results?

(a) `ghci> Book <$> maybeAuthor1 <*> maybeTitle1`

(a) 2

(b) `ghci> Book <$> maybeAuthor1 <*> maybeTitle2`

(b) 3

(c) `ghci> Book <$> maybeAuthor2 <*> maybeTitle1`

(c) 2

(d) `ghci> Book <$> maybeAuthor2 <*> maybeTitle2`

(d) 2

For each expression, choose one of the following options:

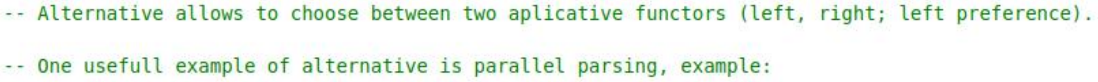
1. `Just (Book "David Copperfield" "Charles Dickens")`
2. `Nothing`
3. `Just (Book "Charles Dickens" "David Copperfield")`
4. `Book (Just "Charles Dickens") (Just "David Copperfield")`

4. (1 point (bonus)) An `Alternative` is a monoid on applicative functors:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

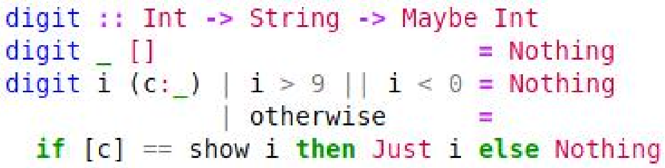
```
instance Alternative Maybe where
  empty = Nothing
  Nothing <|> mx = mx
  mx      <|> _  = mx
```

What's so special about `Alternative`? Why is an `Alternative` useful? If you want, you can use the `Maybe` type as an example.

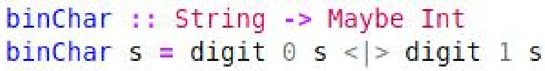


```
-- Alternative allows to choose between two aplicative functors (left, right; left preference).

-- One usefull example of alternative is parallel parsing, example:
```



```
digit :: Int -> String -> Maybe Int
digit _ []                      = Nothing
digit i (c:_) | i > 9 || i < 0 = Nothing
              | otherwise      =
  if [c] == show i then Just i else Nothing
```



```
binChar :: String -> Maybe Int
binChar s = digit 0 s <|> digit 1 s
```