CS131 Applicative worksheet
Due at the beginning of class on Tuesday, February 20th

Name: ____________________________________________

CAS ID (e.g., abc01234@pomona.edu): ________________________

I encourage you to collaborate. Please record your collaborations below.

Most solutions can be written in a single-line. Some solutions may take as many as four or five lines, but any more and you’re off the scent.

Feel free to use Prelude definitions that help... but don’t make the question trivial.

Each question is worth one point.

Please turn in your work as a double-sided printout of this sheet, not on separate paper. If you would rather typeset your work, I can give you the \LaTeX... but you’ll learn more by writing it by hand.

Collaborators: ____________________________________________

________________________________________________________________________
1. Instances

1.1 One of each, please

Write Functor and Applicative instances for the following datatype. Write the instantiated type of each instance function, even though Haskell doesn’t want it.

data Pair a = Pair a a

1.2 One or the other

Write Functor and Applicative instances for the following datatype. Write the instantiated type of each instance function, even though Haskell doesn’t want it.

data Choice a = ColumnA a | ColumnB a

Justify your choice for pure.
1.3 Reader

instance Functor ((->) r) where
    -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap = (.)

Write an Applicative instance for (->) r. Write the instantiated type of each instance function, even though Haskell doesn’t want it. (It’s common to write such types as comments, like above.)

1.4 Reading is fundamental

Write a function xPlusOnexPlusTwo :: Num a => a -> a which computes \((x+1)(x+2)\), e.g. \(xPlusOnexPlusTwo 2\) yields 12. To receive full credit, do not introduce any variables.

Hint: try using readers!
2 Abstracting out the essence

2.1 Not if you called them “stench blossoms”

Write a function that takes a first name and a last name and tries to join them into a full name. We'll do it first for Maybe and Either, then in general. For example, \texttt{maybeName (Just "Dr.") (Just "Dave")} should yield \texttt{Just "Dr. Dave"}, but \texttt{maybeName (Just "Madonna") Nothing} should yield \texttt{Nothing}.

\texttt{maybeName :: Maybe String \rightarrow Maybe String \rightarrow Maybe String}

\texttt{eitherName :: Either e String \rightarrow Either e String \rightarrow Either e String}

\texttt{nameA :: Applicative f \Rightarrow f String \rightarrow f String \rightarrow f String}
2.2 Are phonebooks even a thing anymore?

import qualified Data.Map as Map
import Data.Map (Map)

Given a key, a value, and a map, `Map.insert` will add a new mapping. Write the following two functions using pattern matching which try to add new mappings, if all of the appropriate information is present.

maybeInsert :: Ord k => Maybe k -> Maybe a -> Map k a -> Maybe (Map k a)

eitherInsert :: Ord k => Either e k -> Either e a -> Map k a -> Either e (Map k a)

Write the following function. The `A` is for Applicative.

insertA :: Applicative f, Ord k => f k -> f a -> Map k a -> f (Map k a)
3 Generalizing

3.1 How art thou a king // But by fair sequence and succession?
Write a function sequenceA :: Applicative f => [f a] -> f [a].

Go take a look at the Traversable type class in the Prelude.

3.2 One-upping
Look at Control.Applicative: there are functions liftA, liftA2, and liftA3. Look at the type of liftA...
what other names does this function have?

Implement liftA2 and liftA3.

Write the type of liftA4 and implement it.
3.3 To the left, to the left

Write (⋆⋆) :: Applicative f => f a -> f b -> f b (without using the built-in (⋆⋆) of the type class itself). Note that Nothing ⋆⋆ Just "little old me" == Nothing.

Write (<⋆) :: Applicative f => f a -> f b -> f a.

Why does Haskell include default definitions for (<⋆) and (⋆⋆) in the Applicative type class itself, as opposed to defining these functions outside the type class?

3.4 I’m not listening

Write a function ignore :: Applicative f => f a -> f (). Throw away as little as possible.
3.5  Pair programming

Write a function $(>\ll) :: \text{Applicative } f \Rightarrow f \ a \rightarrow f \ b \rightarrow f \ (a,b)$. 

3.6  Flip it and reverse it

Write a function $(<\ll> :: \text{Applicative } f \Rightarrow f \ a \rightarrow f \ (a \rightarrow b) \rightarrow f \ b$. Make sure your function works from left to right. Can you write it without writing any lambdas?
The `Alternative` class is defined as follows:

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

### 4.1 One way or another

Define an instance for `Alternative Maybe`. The following should all hold:

- `empty <|> a == a`
- `a <|> empty == a`
- `Just v <|> a == Just v`
- `(a <|> b) <|> c == a <|> (b <|> c)`
- `empty <*> a == empty`

### 4.2 Midnight watch

Define the function `guard :: Alternative f => Bool -> f ()`

### 4.3 Answering this question is required

Define the function `optional :: Alternative f => f a -> f (Maybe a)`.

---

1. The actual `Alternative` type class has two other functions, `some, many :: f a -> f [a]`. We’ll use them for parsers, but they don’t make much sense in this setting, so we’ll leave them out.