CS131 Applicative worksheet

This isn't homework and is worth no credit. But it's good practice!

Name: _____

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

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

I encourage you to work these problems by hand, on the board. You'll learn the most, not to mention getting the best practice for exams.

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.

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.)

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, `maybeName (Just "Dr.") (Just "Dave")` should yield `Just "Dr. Dave"`, but `maybeName (Just "Madonna") Nothing` should yield `Nothing`.

```
maybeName :: Maybe String -> Maybe String -> Maybe String
```

```
eitherName :: Either e String -> Either e String -> Either e String
```

```
nameA :: Applicative f => f String -> f String -> 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 :: Maybe k -> Maybe a -> Map k a -> Maybe (Map k a)
```

```
eitherInsert :: Either e k -> Either e a -> Map k a -> Either e (Map k a)
```

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

```
insertA :: Applicative f => 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 `(<*) :: f a -> f b -> f a`.

Why does Haskell include `(<*)` and `(*>)` in the Applicative type class?

3.4 I'm not listening

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