${
m CS131}$ Applicative worksheet Due at the beginning of class on Tuesday, October 3rd

Name:		
CAS ID (e.g., abc0123	34@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 <i>help</i> 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 IATEX 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.)

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 :: Ord $k \Rightarrow$ Maybe $k \rightarrow$ Maybe a \rightarrow Map $k a \rightarrow$ Maybe (Map k a)

eitherInsert :: Ord $k \Rightarrow$ Either e $k \Rightarrow$ Either e a \Rightarrow Map k = Either e (Map k =)

Write the following function. The A is for Applicative.

insertA :: Applicative f, Ord $k \Rightarrow f k \Rightarrow f a \Rightarrow Map k a \Rightarrow f (Map k a)$

3 Generalizing

3.1	How art thou a king	: // B	ut by	fair	sequence	and	succession?
-----	---------------------	--------	-------	------	----------	-----	-------------

Write a function sequence A :: Applicative f => [f a] -> f [a].

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

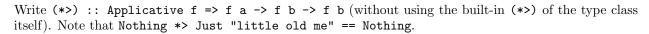
3.2 One-upping

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	T_0	the	left.	to	the	left
$\boldsymbol{\sigma}$	\mathbf{T}	$\mathbf{u}_{\mathbf{I}\mathbf{I}\mathbf{C}}$	TCTU,	\mathbf{u}	$\mathbf{u}_{\mathbf{I}\mathbf{I}\mathbf{C}}$	1010



Write (<*) :: Applicative $f \Rightarrow f a \rightarrow f b \rightarrow f a$.

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 $f \Rightarrow f$ a $\rightarrow f$ (). Throw away as little as possible.



Write a function (>*<) :: Applicative $f \Rightarrow f a \rightarrow f b \rightarrow f (a,b)$.

3.6 Flip it and reverse it

Write a function (<***>) :: 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?

4 Alternative

The Alternative class is defined as follows: 1

```
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:

4.2 Midnight watch

Define the function guard :: Alternative $f \Rightarrow Bool \rightarrow f$ ()

4.3 Answering this question is required

Define the function optional :: Alternative $f \Rightarrow f a \rightarrow f$ (Maybe a).

¹The 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.