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.

```haskell
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.

```haskell
data Choice a = ColumnA a \mid 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 \((x+1) \cdot (x+2)\), e.g. \(xPlusOnexPlusTwo 2\) yields 12. Try to write it “point free”, i.e. don’t introduce any variables or write any lambda expressions.

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

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

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

```hs
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 \textit{try} to add new mappings, if all of the appropriate information is present. You shouldn’t return anything if the information isn’t all there. All three functions should behave the same (morally speaking).

\textbf{maybeInsert} :: Ord \(k\) \(\Rightarrow\) Maybe \(k\) \(\Rightarrow\) Maybe \(a\) \(\Rightarrow\) Map \(k\) \(a\) \(\Rightarrow\) Maybe (Map \(k\) \(a\))

\textbf{eitherInsert} :: Ord \(k\) \(\Rightarrow\) Either \(e\) \(k\) \(\Rightarrow\) Either \(e\) \(a\) \(\Rightarrow\) Map \(k\) \(a\) \(\Rightarrow\) Either \(e\) (Map \(k\) \(a\))

Write the following function. The \(A\) is for \textit{Applicative}.

\textbf{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 // 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 \((\star\star) \:: \text{Applicative } f \Rightarrow f \ a \to f \ b \to f \ b\) (without using the built-in \((\star\star)\) of the type class itself). Note that \(\text{Nothing} \star\star \text{Just "little old me"} = \text{Nothing}\).

Write \((\star\lt) \:: \text{Applicative } f \Rightarrow f \ a \to f \ b \to f \ a\).

Why does Haskell include default definitions for \((\star\lt)\) and \((\star\star)\) 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 \(\text{ignore} :: \text{Applicative } f \Rightarrow f \ a \to f \ ()\). Throw away as little as possible.
3.5 Pair programming

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

3.6 Flip it and reverse it

Write a function \( (<<*>>) :: \text{Applicative}\ f \Rightarrow f\ a \rightarrow f\ (a \rightarrow b) \rightarrow f\ b \). Make sure your function works from left to right. Can you write it without writing any lambdas?
4 Alternative

The Alternative class is defined as follows:

\[
\text{class Applicative } f \Rightarrow \text{Alternative } f \text{ where} \\
\quad \text{empty :: } f \ a \\
\quad (\langle|\rangle) :: f \ a \to f \ a \to f \ a
\]

4.1 One way or another

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

\[
\begin{align*}
\text{empty } \langle|\rangle \ a & = a \\
\text{a } \langle|\rangle \ \text{empty} & = a \\
\text{Just } v \langle|\rangle \ a & = \text{Just } v \\
(a \langle|\rangle \ b) \langle|\rangle \ c & = a \langle|\rangle (b \langle|\rangle \ c) \\
\text{empty } \langle\ast\rangle \ a & = \text{empty}
\end{align*}
\]

4.2 Midnight watch

Define the function \(\text{guard} :: \text{Alternative } f \Rightarrow \text{Bool} \to f ()\)

4.3 Answering this question is required

Not really, it’s not for credit. Let me know if you come up with a funnier title.

Define the function \(\text{optional} :: \text{Alternative } f \Rightarrow f \ a \to f (\text{Maybe } a)\).

---

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