Exercise 1
Let consider the following type definition representing a tree (what we saw so far was simply a binary tree)

```
data Tree a = EmptyTree | Node a [(Tree a)] deriving Show
```

Knowing that [] is a functor, make this type an instance of functor and applicative functor.

Exercise 2

Determine if a valid Functor can be written for the datatype provided.
1. data Bool = False | True
2. data BoolAndSomethingElse a = False' a | True' a
3. data BoolAndMaybeSomethingElse a = Falsish | Truish a
4. Use the kinds to guide you on this one, don’t get too hung up on the details.

    newtype Mu f = InF { outF :: f (Mu f) }

5. Again, follow the kinds and ignore the unfamiliar parts

    import GHC.Arr
    data D = D (Array Word Word) Int Int

Exercise 3
Rearrange the arguments to the type constructor of the datatype so the Functor instance works.
1.

```
data Sum a b = First a | Second b
instance Functor (Sum e) where
fmap f (First a) = First (f a)
fmap f (Second b) = Second b
```

2.

```
data Company a b c = DeepBlue a c | Something b
instance Functor (Company e e') where
fmap f (Something b) = Something (f b)
fmap _ (DeepBlue a c) = DeepBlue a c
```

3.

```
data More a b = L a b a | R b a b deriving (Eq, Show)
instance Functor (More x) where
fmap f (L a b a') = L (f a) b (f a')
fmap f (R b a b') = R b (f a) b'
```

Keeping in mind that it should result in a Functor that does the following:

```
Prelude> fmap (+1) (L 1 2 3)
L 2 2 4
Prelude> fmap (+1) (R 1 2 3)
R 1 3 3
```

Exercise 4
Write Functor instances for the following datatypes.
1. `data Quant a b = Finance | Desk a | Bloor b`
2. No, it’s not interesting by itself.

```
data K a b = K a
```

3.

```
 {-# LANGUAGE FlexibleInstances #-}
newtype Flip f a b = Flip (f b a) deriving (Eq, Show)
newtype K a b = K a
instance Functor (Flip K a) where
fmap = undefined
```

4.

```
data EvilGoateeConst a b =
GoatyConst b
```

-- You thought you'd escaped the goats
-- by now didn't you? Nope.

No, it doesn't do anything interesting. No magic here or
in the previous exercise. If it works, you succeeded.

5. Do you need something extra to make the instance work?

```
data LiftItOut f a = LiftItOut (f a)
```

6. `data Parappa f g a = DaWrappa (f a) (g a)`

7. Don't ask for more typeclass instances than you need. You
can let GHC tell you what to do.

```
data IgnoreOne f g a b = IgnoringSomething (f a) (g b)
```

8. `data Notorious g o a t = Notorious (g o) (g a) (g t)`

9. You'll need to use recursion.

```
data List a = Nil | Cons a (List a)
```

10. A tree of goats forms a Goat-Lord, fearsome poly-creature.

```
data GoatLord a = NoGoat | OneGoat a | MoreGoats
(GoatLord a) (GoatLord a) (GoatLord a)
-- A VERITABLE HYDRA OF GOATS
```

11. You'll use an extra functor for this one, although your solution might
do it monomorphically without using fmap. Keep in mind that you will
probably not be able to validate this one in the usual manner. Do your
best to make it work.

```
data TalkToMe a = Halt | Print String a | Read (String ->
a)
```

Exercise 5

Given a type that has an instance of Applicative, specialize the types of
the methods. Test your specialization in the REPL. One way to do this is
to bind aliases of the typeclass methods to more concrete types that
have the type we told you to fill in.

1. -- Type
    []
-- Methods

```
pure :: a -> ? a
(<*>) :: ? (a -> b) -> ? a -> ? b
```

2. -- Type
IO
-- Methods

```
pure :: a -> ? a
(<*>) :: ? (a -> b) -> ? a -> ? b
```

3. -- Type

```
(,) a
```

-- Methods

```
pure :: a -> ? a
(<*>) :: ? (a -> b) -> ? a -> ? b
```

4. -- Type

```
(->) e
```

-- Methods

```
pure :: a -> ? a
(<*>) :: ? (a -> b) -> ? a -> ? b
```

Exercise 6

Write instances for the following datatypes. Confused?
Write out what the type should be. Use the checkers library to validate the instances.

1. data Pair a = Pair a a deriving Show

2. This should look familiar.

```haskell
data Two a b = Two a b
```

3. `data Three a b c = Three a b c`
4. `data Three' a b = Three' a b b`
5. `data Four a b c d = Four a b c d`
6. `data Four' a b = Four' a a a b`

Exercise 7

1. Ordered pairs (a, b) are data of type (,) a b. For instance, (1, 2) is data of type (,) Int Int. Make the type ((,) a) an instance of Functor and Applicative functor classes.
2. **Association Lists / Functors composition**

   Association lists are lists of key-value pairs, and have long been used in functional programming languages to represent finite functions. In Haskell, we can represent association lists very cleanly:

```haskell
type Assoc a b = [(a,b)]
```

Or, perhaps more suggestively for our current purposes,

```haskell
type Assoc a b = [] ((,) a b)
```

If this were ordinary code, we'd be tempted to write

```haskell
type Assoc a b = [] ((,) a b)  = ([] . (,) a) b
```

and then reduce to

```haskell
type Assoc a = [] . (,) a
```

This sort of thing is possible (albeit, not with this syntax), but our point here isn't to repurpose our code rewriting techniques as type rewriting techniques, but instead to note that Assoc a can be thought of as involving the composition of two functors, [] and (,) a. This is the more significant observation. Compositions of homomorphisms are homomorphisms, and therefore compositions of functors must also be functors. The question is how to take advantage of this?

Our goal is to make Assoc a (where Assoc a b is a type alias for [(a,b)]) into a functor, much as (->) a is a functor (remembering now that Assoc is often used to represent finite functions). One problem is that we can't turn a type *synonym* into a functor, only a *type*, optionally applied to type variables. Another problem is that the outer type constructor [] is already a Functor, and types can belong to type classes in only one way.

A first approach to this solution would be to wrap our aliased type, and define

```
newtype Assoc a b = Assoc { getPairs :: [(a,b)] }
```

Make Assoc a an instance of Functor

```
instance Functor (Assoc a) where
   fmap f assoc = ??
```

This works well. An alternative might be to stick with the original definition of Assoc as a type alias, and introduce a new name for the fmap function associated with the composed functors. Give the definition of amap below.,

```
amap :: (b -> c) -> Assoc a b -> Assoc a c
amap f ps = ??
```

Make Assoc a an instance of applicative functor

This is easy enough, and after a while intuitive enough, that Haskell programmers typically don't wrap composed functor instances, nor do they introduce new names associated with viewing a composed functor instance as a functor directly. Instead, they just use (fmap . fmap) to lift a function through two functors.

The really striking thing is that this generalizes! Let's consider a type that involves a three-level composition of functors, e.g.,

```
type AssocMap a b c = [(a,b->c)]
```

This is a type alias that composes the functors [], (,) a, and (->) b, applying the result to c. If we want to produce a function that does a remapping three levels down:

```
mmap :: (c -> d) -> AssocMap a b c -> AssocMap a b d
mmap f assocmap = ??
```

Make AssocMap a b an instance of Functor and applicative functor

Hint: We'll eventually derive mmap = fmap . fmap . fmap! So we can just use fmap . fmap . fmap.

http://cmsc-16100.cs.uchicago.edu/2021-autumn/Lectures/09/functors.php