1. Part One

1.1. Simple Algebraic Data Types

Challeges:

1. Showing isomorphism between (Maybe a) and (Either () a):

```
maybeToEither :: Maybe a -> Either () a
maybeToEither Nothing = Left ()
maybeToEither (Some v) = Right v

eitherToMaybe :: Either () a -> Maybe a
eitherToMaybe (Left ()) = Nothing
eitherToMaybe (Right v) = Some v
```

1. Showing that $a + a = 2 \times a$

i show that by isomorphism between (Either a a) and (Bool,a)

```
f :: Either a a -> (Bool, a)
f (Right a) = (True, a)
f (Left a) = (False, a)

g :: (Bool, a) -> Either a a
g (True, a) = Right a
g (False, a) = Left a
```

1.2. Functors

Functor laws :

- preserves identity : $Fid\ a = id_{Fa} Fa$
- preserves composition: $F(g \circ f) = Fg \circ Ff$

Challenges:

2. proving functor laws on reader functor

```
f :: a -> b
g :: b -> c

fmap (g . f) reader = (g . f) . reader

fmap g (fmap f reader) = fmap g (f . reader) = g . (f .
```

```
reader)

-- by assosiativity of compostion it works
```

4. proving functor law for lists

```
-- Identity preservation

id [] = []
id (x:xs) = x : id xs

fmap id [] = []
fmap id (x:xs) = (id x) : fmap id xs

-- composition preservation

fmap (g . f) [] = []
fmap (g . f) (x:xs) = (g.f) x : fmap (g . f) xs

fmap g (fmap f []) = fmap g [] = []
fmap g (fmap f (x:xs)) = fmap g (f x : fmap f xs)
                       = g (f x) : fmap g (fmap f xs)
                       = (g . f) x : fmap (g . f) xs
```

1.3. Functoriality

Challenges:

2. showing that Maybe is isomorphic to :

```
type Maybe' a = Either (Const () a) (Identity a)

--- First ->
f :: Maybe a -> Maybe' a
f Nothing = Left (Const ())
f (Just a) = Right (Identity a)

--- Second <-
g :: Maybe' a -> Maybe a
g (Left _) = Nothing
g (Right (Identity a)) = Just a
```

```haskell
--- Showing they are inverse of each other

---- first f . g
f . g (Left (Const ())) = f Nothing
                        = Nothing
f . g (Right (Identity a)) = f (Just a)
                           = Right (Identity a)


f . g = id

---- second g . f
g . f Nothing = g (Left (Const ()))
              = Nothing
g . f (Just a) = g (Right (Identity a))
               = Just a
g . f = id

---- Qed
```

5. Definition of bifunctors in a language other than haskell here i will use rust

```rust
trait Bifunctor {
  type A;
  type B;

  fn bimap(f:F,s:S) -> B {
    compose(first(f),second(s))
  }

  fn first(f:F) -> B {
    bimap(f,id)
  }

  fn second(s:S) -> B {
    bimap(id,s)
```

```
    }
}
```