```haskell
import           Test.QuickCheck hiding (Failure, Success)

semigroupAssoc :: (Eq m, Semigroup m) => m -> m -> m -> Bool
semigroupAssoc a b c = (a <> (b <> c)) == ((a <> b) <> c)

monoidLeftIdentity   :: (Eq m, Monoid m) => m -> Bool
monoidLeftIdentity a = (mempty <> a) == a

monoidRightIdentity   :: (Eq m, Monoid m) => m -> Bool
monoidRightIdentity a = (a <> mempty) == a
```

Laboratorul 10

Exerciții pentru Semigroup și Monoid din HaskellBook

Setup

Given a datatype, implement the Semigroup and Monoid instances. Add Semigroup/Monoid constraints to type variables where needed.

Note: We're not always going to derive every instance you may want or need in the datatypes we provide for exercises. We expect you to know what you need and to take care of it yourself by this point.

Validate all of your instances with QuickCheck. Since the only law is associativity, that's the only property you need to reuse.

Example 1 - Trivial

```haskell
-- Example 1 - Trivial

data Trivial = Trivial
  deriving (Eq, Show)

instance Semigroup Trivial where
  _ <> _ = Trivial

instance Monoid Trivial where
  mempty  = Trivial

instance Arbitrary Trivial where
  arbitrary = return Trivial

type TrivAssoc = Trivial -> Trivial -> Trivial -> Bool
type TrivId    = Trivial -> Bool
```

```haskell
testTrivial :: IO ()
testTrivial
  = do
    quickCheck (semigroupAssoc :: TrivAssoc)
    quickCheck (monoidLeftIdentity :: TrivId)
    quickCheck (monoidRightIdentity :: TrivId)
```

Exercise 2 - Identity

```haskell
-- Exercise 2 - Identity

newtype Identity a = Identity a
  deriving (Eq, Show)

instance Semigroup a => Semigroup (Identity a) where
  Identity x <> Identity y = undefined

instance (Semigroup a, Monoid a) => Monoid (Identity a) where
  mempty = undefined

instance Arbitrary a => Arbitrary (Identity a) where
  arbitrary = Identity <$> arbitrary
```

Exercise 3 - Pair

```haskell
-- Exercise 3 - Pair

data Two a b = Two a b
  deriving (Eq, Show)

instance (Arbitrary a, Arbitrary b) => Arbitrary (Two a b) where
  arbitrary = Two <$> arbitrary <*> arbitrary
```

Exercise 4 - Triple

```haskell
-- Exercise 4 - Triple

data Three a b c = Three a b c
  deriving (Eq, Show)

instance ( Arbitrary a
         , Arbitrary b
         , Arbitrary c
         ) => Arbitrary (Three a b c) where
  arbitrary = Three <$> arbitrary <*> arbitrary <*> arbitrary
```

Exercise 5 - Boolean conjunction

```
-- Exercise 5 - Boolean conjunction

newtype BoolConj = BoolConj Bool
  deriving (Eq, Show)

instance Arbitrary BoolConj where
  arbitrary = BoolConj <$> arbitrary
```

Exercise 6 - Boolean disjunction

```
-- Exercise 6 - Boolean disjunction

newtype BoolDisj = BoolDisj Bool
  deriving (Eq, Show)

instance Arbitrary BoolDisj where
  arbitrary = BoolDisj <$> arbitrary
```

Exercise 7 - Or

```
-- Exercise 7 - Or

data Or a b = Fst a | Snd b
  deriving (Eq, Show)

instance (Arbitrary a, Arbitrary b) => Arbitrary (Or a b) where
  arbitrary = oneof [Fst <$> arbitrary, Snd <$> arbitrary]
```

The Semigroup for Or should have the following behavior. We can think of it as having a "sticky" Snd value, whereby it'll hold onto the first Snd value when and if one is passed as an argument.

```
Prelude> Fst 1 <> Snd 2
Snd 2
Prelude> Fst 1 <> Fst 2
Fst 2
Prelude> Snd 1 <> Fst 2
Snd 1
Prelude> Snd 1 <> Snd 2
Snd 1
```

Exercise 8 - Lifting Monoid to Functions

```
-- Exercise 8 - Lifting Monoid to Functions
```

```haskell
newtype Combine a b = Combine { unCombine :: a -> b }

instance (CoArbitrary a, Arbitrary b) => Arbitrary (Combine a b) where
  arbitrary = Combine <$> arbitrary
```

What it should do:

```
Prelude> f = Combine $ \n -> Sum (n + 1)
Prelude> g = Combine $ \n -> Sum (n - 1)
Prelude> unCombine (f <> g) $ 0
Sum {getSum = 0}
Prelude> unCombine (f <> g) $ 1
Sum {getSum = 2}
Prelude> unCombine (f <> f) $ 1
Sum {getSum = 4}
Prelude> unCombine (g <> f) $ 1
Sum {getSum = 2}
```

Hint: This function will eventually be applied to a single value of type a. But you'll have multiple functions that can produce a value of type b. How do we combine multiple values so we have a single b? This one will probably be tricky! Remember that the type of the value inside of Combine is that of a function.