```haskell
{-# LANGUAGE FlexibleInstances #-}
import Data.Monoid
import Data.Semigroup (Max (..), Min (..))
import Data.Foldable (foldMap, foldr)
import Data.Char (isUpper)
```

Laboratorul 11

Exerciții pentru Foldable

din HaskellBook

1. Implementați următoarele funcții folosind `foldMap` și/sau `foldr` din clasa `Foldable`, apoi testați-le cu mai multe tipuri care au instanță pentru `Foldable`

```haskell
elem :: (Foldable t, Eq a) => a -> t a -> Bool
elem = undefined

null :: (Foldable t) => t a -> Bool
null = undefined

length :: (Foldable t) => t a -> Int
length = undefined

toList :: (Foldable t) => t a -> [a]
toList = undefined
```

`fold` combină elementele unei structuri folosind structura de monoid a acestora.

```haskell
fold :: (Foldable t, Monoid m) => t m -> m
fold = undefined -- Hint: folosiți foldMap
```

2. Scrieți instanțe ale lui `Foldable` pentru următoarele tipuri

```haskell
data Constant a b = Constant b

data Two a b = Two a b

data Three a b c = Three a b c

data Three' a b = Three' a b b

data Four' a b = Four' a b b b

data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a) (GoatLord a) (GoatLord a)
```

3. Scrieți o funcție de filtrare pentru `Foldable`

```haskell
filterF
    :: ( Applicative f
       , Foldable t
       , Monoid (f a)
```

```
        )
     => (a -> Bool) -> t a -> f a
filterF = undefined -- Hint folosiți foldMap
```

Aveți mai jos cinci teste pentru funcția de mai sus. Explicați de ce este posibil ca pentru aceeași parte dreaptă a lui == să avem cinci lucruri diferite in partea stângă

```
unit_testFilterF1 = filterF Data.Char.isUpper "aNA aRe mEre" == "NARE"
unit_testFilterF2 = filterF Data.Char.isUpper "aNA aRe mEre" == First (Just 'N')
unit_testFilterF3 = filterF Data.Char.isUpper "aNA aRe mEre" == Min 'A'
unit_testFilterF4 = filterF Data.Char.isUpper "aNA aRe mEre" == Max 'R'
unit_testFilterF5 = filterF Data.Char.isUpper "aNA aRe mEre" == Last (Just 'E')
```

Exerciții pentru Functor

Scrieți instanțe ale clasei `Functor` pentru tipurile de date descrise mai jos.

```
newtype Identity a = Identity a

data Pair a = Pair a a

-- scrieți instanță de Functor pentru tipul Two de mai sus

-- scrieți instanță de Functor pentru tipul Three de mai sus

-- scrieți instanță de Functor pentru tipul Three' de mai sus

data Four a b c d = Four a b c d

data Four'' a b = Four'' a a a b

-- scrieți o instanță de Functor penru tipul Constant de mai sus

data Quant a b = Finance | Desk a | Bloor b

data K a b = K a

newtype Flip f a b = Flip (f b a) deriving (Eq, Show)
  -- pentru Flip nu trebuie să faceți instanță

instance Functor (Flip K a) where
  fmap = undefined
```

S-ar putea să fie nevoie să adăugați unele constrângeri la definirea instanțelor

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

```haskell
data Parappa f g a = DaWrappa (f a) (g a)

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

data Notorious g o a t = Notorious (g o) (g a) (g t)

-- scrieți o instanță de Functor pentru tipul GoatLord de mai sus

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