```haskell
-- Laboratorul 10

-- FUNCTORI

-- class Functor f where
-- fmap :: ( a -> b ) -> f a -> f b

-- Scrieti instante ale clasei Functor pentru tipurile de date descrise mai jos.

newtype Identity a = Identity a
instance Functor Identity where
    fmap :: (a -> b) -> Identity a -> Identity b
    fmap f (Identity a) = Identity (f a)

data Pair a = Pair a a
instance Functor Pair where
    fmap :: (a -> b) -> Pair a -> Pair b
    fmap f (Pair a1 a2) = Pair (f a1) (f a2)

data Constant a b = Constant b
instance Functor (Constant a) where
    fmap :: forall k (a1 :: k) a2 b.(a2 -> b) -> Constant a1 a2 -> Constant a1 b
    fmap f (Constant b) = Constant (f b)

data Two a b = Two a b
instance Functor (Two a) where
    fmap :: (a2 -> b) -> Two a1 a2 -> Two a1 b
    fmap f (Two a b) = Two a (f b)

data Three a b c = Three a b c
instance Functor (Three a b) where
    fmap :: (a2 -> b2) -> Three a1 b1 a2 -> Three a1 b1 b2
    fmap f (Three a b c) = Three a b (f c)

data Three' a b = Three' a b b
instance Functor (Three' a) where
    fmap :: (a2 -> b) -> Three' a1 a2 -> Three' a1 b
    fmap f (Three' a b1 b2) = Three' a (f b1) (f b2)

data Four a b c d = Four a b c d
instance Functor (Four a b c) where
    fmap :: (a2 -> b2) -> Four a1 b1 c a2 -> Four a1 b1 c b2
    fmap f (Four a b c d) = Four a b c (f d)

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

```haskell
instance Functor (Four'' b) where
    fmap :: (a -> b2) -> Four'' b1 a -> Four'' b1 b2
    fmap f (Four'' a1 a2 a3 b) = Four'' a1 a2 a3 (f b)

data Quant a b = Finance | Desk a | Bloor b
instance Functor (Quant a) where
    fmap :: (a2 -> b) -> Quant a1 a2 -> Quant a1 b
    fmap f Finance  = Finance
    fmap f (Desk a) = Desk a
    fmap f (Bloor a) = Bloor (f a)

-- S-ar putea să fie nevoie să adăugati unele constrângeri la definirea
-- instantelor

data LiftItOut f a = LiftItOut (f a)
instance Functor f => Functor (LiftItOut f) where
    fmap :: Functor f => (a -> b) -> LiftItOut f a -> LiftItOut f b
    fmap f (LiftItOut fa) = LiftItOut (fmap f fa)

data Parappa f g a = DaWrappa (f a) (g a)
instance (Functor f, Functor g) => Functor (Parappa f g) where
    fmap :: (Functor f, Functor g) => (a -> b) -> Parappa f g a -> Parappa f g b
    fmap f (DaWrappa fa ga) = DaWrappa (fmap f fa) (fmap f ga)

data IgnoreOne f g a b = IgnoringSomething (f a) (g b)
instance Functor g => Functor (IgnoreOne f g a) where
    fmap :: forall k (g :: * -> *) (f :: k -> *) (a :: k) a1 b.Functor g =>(a1 ->
b) -> IgnoreOne f g a a1 -> IgnoreOne f g a b
    fmap f (IgnoringSomething fa ga) = IgnoringSomething fa (fmap f ga)

data Notorious g o a t = Notorious (g o) (g a) (g t)
instance Functor g => Functor (Notorious g o a) where
    fmap :: Functor g => (a1 -> b) -> Notorious g o a a1 -> Notorious g o a b
    fmap f (Notorious go ga gt) = Notorious go ga (fmap f gt)

data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a) (GoatLord a)
(GoatLord a)
instance Functor GoatLord where
    fmap :: (a -> b) -> GoatLord a -> GoatLord b
    fmap f NoGoat = NoGoat
    fmap f (OneGoat a) = OneGoat (f a)
    fmap f (MoreGoats a1 a2 a3) = MoreGoats (fmap f a1) (fmap f a2) (fmap f a3)

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

```haskell
fmap :: (a -> b) -> TalkToMe a -> TalkToMe b
fmap f Halt = Halt
fmap f (Print str a) = Print str (f a)
fmap f (Read fstr) = Read (f . fstr)
```