```haskell
-- Laboratorul 11

-- Amintiti-vă clasele Functor si Applicative, rulati si analizati următoarele
-- exemple.

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

-- class Functor f => Applicative f where
-- pure :: a -> f a
-- (<*>) :: f (a -> b) -> f a -> f b

-- Just length <*> Just "world"
-- ------------------
-- Just 5

-- Just (++" world") <*> Just "hello,"
-- ------------------
-- Just "hello, world"

-- pure (+) <*> Just 3 <*> Just 5
-- -------------------
-- Explicatie: Just (+) -> Just((+) 3 5)
-- Just 8

-- pure (+) <*> Just 3 <*> Nothing
-- -------------------
-- Nothing

-- (++) <$> ["ha","heh"] <*> ["?","!"]
-- --------------------
-- ["ha?","ha!","heh?","heh!"]

-- Exercitii

-- 1. Se dă tipul de date

data List a = Nil
            | Cons a (List a)
            deriving (Eq, Show)

-- Să se scrie instante Functor si Applicative pentru tipul de date List.

instance Functor List where
    fmap :: (a -> b) -> List a -> List b
```

```
    fmap f Nil = Nil
    fmap f (Cons a l) = Cons (f a) Nil

concatOnList :: List a -> List a -> List a
concatOnList l Nil = l
concatOnList Nil l = l
concatOnList (Cons l x) y = Cons l (concatOnList x y)

instance Applicative List where
    pure :: a -> List a
    pure a = Cons a Nil
    (<*>) :: List (a -> b) -> List a -> List b
    f <*> Nil = Nil
    Nil <*> f = Nil
    Cons f l <*> (Cons x y) = Cons (f x) (concatOnList (fmap f y) (l <*> Cons x
y))

-- Exemple

f :: List (Integer -> Integer)
f = Cons (+1) (Cons (*2) Nil)

v :: List Integer
v = Cons 1 (Cons 2 Nil)

test1 :: Bool
test1 = (f <*> v) == Cons 2 (Cons 3 (Cons 2 (Cons 4 Nil)))

-- 2. Se dă tipul de date

data Cow = Cow {name :: String, age :: Int, weight :: Int}
        deriving (Eq, Show)

-- a) Să se scrie functiile noEmpty, respectiv noNegative care valideaza un
-- string, respectiv un intreg.

noEmpty :: String -> Maybe String
noEmpty str = if not (null str) then Just str else Nothing

noNegative :: Int -> Maybe Int
noNegative x = if x < 0 then Nothing else Just x

test21 :: Bool
test21 = noEmpty "abc" == Just "abc"
```

```haskell
test22 :: Bool
test22 = noNegative (-5) == Nothing

test23 :: Bool
test23 = noNegative 5 == Just 5

-- b) Sa se scrie o functie care construieste un element de tip Cow verificând
-- numele, varsta si greutatea cu functiile de la a).

cowFromString :: String -> Int -> Int -> Maybe Cow
cowFromString nume varsta greutate = if (noEmpty nume == Just nume) &&
                                       (noNegative varsta == Just varsta) &&
                                       (noNegative greutate == Just greutate)
                                    then Just Cow {name = nume, age = varsta,
weight = greutate}
                                    else Nothing

test24 :: Bool
test24 = cowFromString "Milka" 5 100 == Just (Cow {name = "Milka", age = 5,
weight = 100})

-- c) Se se scrie functia de la b) folosind fmap si <*>.

cowFromString2 :: String -> Int -> Int -> Maybe Cow
cowFromString2 nume varsta greutate = Cow <$> noEmpty nume <*> noNegative varsta
<*> noNegative greutate

-- 3. Se dau următoarele tipuri de date:

newtype Name = Name String deriving (Eq, Show)
newtype Address = Address String deriving (Eq, Show)
data Person = Person Name Address
                deriving (Eq, Show)

-- a) Să se implementeze o functie validateLength care validează lungimea unui
-- sir (sa fie mai mică decât numărul dat ca parametru).

validateLength :: Int -> String -> Maybe String
validateLength nr str = if length str < nr then Just str else Nothing

test31 :: Bool
test31 = validateLength 5 "abc" == Just "abc"

-- b) Să se implementeze functiile mkName si mkAddress care transformă un sir de
-- caractere într-un element din tipul de date asociat, validând stringul cu
```

```haskell
-- functia validateLength (numele trebuie sa aiba maxim 25 caractere iar adresa
-- maxim 100).

mkName :: String -> Maybe Name
mkName nume = if validateLength 26 nume == Just nume then Just (Name nume) else
Nothing

mkAddress :: String -> Maybe Address
mkAddress adresa = if validateLength 26 adresa == Just adresa then Just (Address
adresa) else Nothing

test32 :: Bool
test32 = mkName "Gigel" == Just (Name "Gigel")

test33 :: Bool
test33 = mkAddress "Str Academiei" == Just (Address "Str Academiei")

-- c) Să se implementeze functia mkPerson care primeste ca argument două siruri
-- de caractere si formeaza un element de tip Person daca sunt validate,
-- conditiile folosind functiile implementate mai sus.

mkPerson :: String -> String -> Maybe Person
mkPerson nume adresa = if mkName nume == Just (Name nume) && mkAddress adresa ==
Just (Address adresa )
                       then Just (Person (Name nume) (Address adresa))
                       else Nothing

test34 :: Bool
test34 = mkPerson "Gigel" "Str Academiei" == Just (Person (Name "Gigel") (Address
"Str Academiei"))

-- d) Să se implementeze functiile de la b) si c) folosind fmap si <*>.

mkPerson2 :: String -> String -> Maybe Person
mkPerson2 nume adresa = Person <$> Just (Name nume) <*> Just (Address adresa)
```