

Laboratorul 11

Amintiți-vă clasele Functor și Applicative, rulați și analizați 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 (++ " world") <*> Just "hello,"
pure (+) <*> Just 3 <*> Just 5
pure (+) <*> Just 3 <*> Nothing
(++ ) <$> ["ha","heh"] <*> ["?","!"]
-}
```

Exercitii

1. Se dă tipul de date

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

Să se scrie instanțe Functor și Applicative pentru tipul de date List.

```
instance Functor List where
    fmap = undefined
instance Applicative List where
    pure = undefined
    (<*>) = undefined
```

Exemple

```
f = Cons (+1) (Cons (*2) Nil)
v = Cons 1 (Cons 2 Nil)
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 funcțiile `noEmpty`, respectiv `noNegative` care validează un string, respectiv un întreg.

```
noEmpty :: String -> Maybe String
noEmpty = undefined
```

```
noNegative :: Int -> Maybe Int
noNegative = undefined
```

```
test21 = noEmpty "abc" == Just "abc"
test22 = noNegative (-5) == Nothing
test23 = noNegative 5 == Just 5
```

- b) Să se scrie o funcție care construiește un element de tip `Cow` verificând numele, vârsta și greutatea cu funcțiile de la a).

```
cowFromString :: String -> Int -> Int -> Maybe Cow
cowFromString = undefined
```

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

- c) Se scrie funcția de la b) folosind `fmap` și `<*>`.

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 funcție `validateLength` care validează lungimea unui șir (să fie mai mică decât numărul dat ca parametru).

```
validateLength :: Int -> String -> Maybe String
validateLength = undefined
```

```
test31 = validateLength 5 "abc" == Just "abc"
```

- b) Să se implementeze funcțiile `mkName` și `mkAddress` care transformă un șir de caractere într-un element din tipul de date asociat, validând stringul cu funcția `validateLength` (numele trebuie să aibă maxim 25 caractere iar adresa maxim 100).

```
mkName :: String -> Maybe Name
mkName = undefined
```

```
mkAddress :: String -> Maybe Address
mkAddress = undefined
```

```
test32 = mkName "Gigel" == Just (Name "Gigel")
test33 = mkAddress "Str Academiei" == Just (Address "Str Academiei")
```

- c) Să se implementeze funcția `mkPerson` care primește ca argument două șiruri de caractere și formează un element de tip `Person` dacă sunt validate condițiile, folosind funcțiile implementate mai sus.

```
mkPerson :: String -> String -> Maybe Person
mkPerson = undefined
```

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

- d) Să se implementeze funcțiile de la b) și c) folosind `fmap` și `<*>`.