Laboratorul 10: Monade - Introducere

În acest laborator vom folosi noțiunile din cursurile 9 și 10.

Monada `Maybe`

Lucrați în fișierul `mMaybe.hs`, care conține definiția monadei `Maybe`. Definiția este comentată deoarece monada `Maybe` este definită în GHC.Base

0. Înțelegeți funcționarea operațiilor monadice (»=) și `return`

```
 return 3 :: Maybe Int
Just 3
(Just 3)  >>= (\ x -> if (x>0) then Just (x*x) else Nothing)
Just 9
```

Uneori vom folosi operația derivată (»)

```
ma >> mb = ma >>= \_ -> mb
```

```
(Just 3)  >> Nothing
 Nothing
(Just 3)  >> (Just 6)
 Just 6
```

1. Definiți operatorul de compunere a funcțiilor îmbogățite

```
(<=<) :: (a -> Maybe b) -> (c -> Maybe a) -> c -> Maybe b
f <=< g = (\ x -> g x >>= f)
```

1.1 Creați singuri exemple prin care să înțelegeți funcționarea acestui operator.

1.2 Definiți proprietatea

```
 asoc :: (Int -> Maybe Int) -> (Int -> Maybe Int) -> (Int -> Maybe Int) -> Int -> Bool
```

care pentru trei funcții date verifică asociativitatea operației (<=<):

```
  h <=< (g <=< f) $ x = (h <=< g) <=< f $ x
```

Verificați proprietatea pentru funcții particulare folosind `QuickCheck`.

2. Definim

```haskell
pos :: Int -> Bool
pos  x = if (x>=0) then True else False

foo :: Maybe Int ->  Maybe Bool
foo  mx =  mx  >>= (\x -> Just (pos x))
```

2.1 Înțelegeți ce face funcția `foo`.

2.2 Citiți notația do din cursul 9. Definiți funcția `foo` folosind notația `do`.

3. Vrem să definim o funcție care adună două valori de tip `Maybe Int`

```haskell
addM :: Maybe Int -> Maybe Int -> Maybe Int
addM mx my = undefined
```

Exemplu de funcționare:

```haskell
addM (Just 4)  (Just 3)
Just 7
addM (Just 4)  Nothing
Nothing
addM Nothing  Nothing
Nothing
```

3.1 Definiți `addM` prin orice metodă (de exemplu, folosind șabloane).

3.2 Definiți `addM` folosind operații monadice și notația `do`.

3.3 Scrieti un test care verifica egalitatea dintre cele doua functii de mai sus si rulati cu `QuickCheck`.

Notația `do` și secvențiere

4. Să se treacă în notația `do` urmatoarele funcții:

```haskell
cartesian_product xs ys = xs >>= ( \x -> (ys >>= \y-> return (x,y)))

prod f xs ys = [f x y | x <- xs, y<-ys]

myGetLine :: IO String
myGetLine = getChar >>= \x ->
      if x == '\n' then
          return []
      else
          myGetLine >>= \xs -> return (x:xs)

prelNo noin =  sqrt noin
ioNumber = do
     noin  <- readLn :: IO Float
     putStrLn $ "Intrare\n" ++ (show noin)
     let  noout = prelNo noin
     putStrLn $ "Iesire"
     print noout
```