Laboratorul 13: Monade

Pentru primele exerciții veți folosi fișierul `lab13.hs`, care conține și definiția monadei `Maybe`. Definiția este comentată deoarece monada `Maybe` este deja definită în `GHC.Base`.

Revedeți explicațiile din curs pentru operațiile monadice `>>=` și `return`, și citiți exemplul următor:

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

1. Citiți definițiile de mai jos și încercați să înțelegeți ce face funcția `fct`. Scrieți apoi o definiție pentru `fct` folosind notația `do`.

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

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

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

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

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

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

   b) Definiți `addM` folosind operații monadice și notația `do`.

3. Rescrieți următoarele funcții folosind notația `do`:

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

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

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

4. Rescrieți următoarea funcție folosind notația cu secvențiere:

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

5. Pentru următoarele exerciții veți folosi fișierul `mWriter.hs`, ce conține o definiție a monadei `Writer String` (modificată pentru a compila fără opțiuni suplimentare):

```haskell
newtype WriterS a = Writer { runWriter :: (a, String) }
```

a) Definiți funcțiile `logIncrement` și `logIncrement2` din curs și testați-le.

b) Definiți funcția `logIncrementN`, care generalizează `logIncrement2`, astfel:

```haskell
logIncrementN :: Int -> Int -> WriterS Int
logIncrement x n = undefined
```

```haskell
runWriter $ logIncrementN 2 4
(6,"increment:2\nincrement:3\nincrement:4\nincrement:5\n")
```

c) Modificați definiția monadei `WriterS` astfel încât să producă lista mesajelor de log și nu concatenarea lor. Pentru a evita posibile confuzii, lucrați în alt fișier. Definiți funcția `logIncrementN` în acest context.

```haskell
newtype WriterLS a = Writer {runWriter :: (a, [String])}
```

```haskell
runWriter $ logIncrementN 2 4
(6,["increment:2","increment:3","increment:4","increment:5"])
```

6. Definim tipul de date

```haskell
data Person = Person { name :: String, age :: Int }
```

a) Definiți funcțiile

```haskell
showPersonN :: Person -> String
showPersonA :: Person -> String
```

care afișează "frumos" numele și vârsta unei persoane, urmând modelul:

```
showPersonN $ Person "ada" 20
"NAME: ada"

showPersonA $ Person "ada" 20
"AGE: 20"
```

b) Folosind funcțiile definite pentru exercițiile 5.a) și 5.b), definiți funcția

```
showPerson :: Person -> String
```

care afișează "frumos" toate datele unei persoane, urmând modelul

```
showPerson $ Person "ada" 20
"(NAME: ada, AGE: 20)"
```

c) Folosind monada `Reader` (găsiți implementarea instanțelor în fișierul `lab13.hs`), definiți variante monadice pentru cele trei funcții definite anterior. Variantele monadice vor avea tipul:

```
mshowPersonN ::  Reader Person String
mshowPersonA ::  Reader Person String
mshowPerson ::  Reader Person String

runReader mshowPersonN  $ Person "ada" 20
"NAME:ada"

runReader mshowPersonA  $ Person "ada" 20
"AGE:20"

runReader mshowPerson  $ Person "ada" 20
"(NAME:ada,AGE:20)"
```