Laboratorul 13 - Programare Logică și Funcțională

Seria 36

Mai 2024

In acest laborator vom invata despre **Monade**.

Pentru o familiarizare, puteti parcurge articolul lui Dan Piponi, *You Could Have Invented Monads! (And Maybe You Already Have.)*, pe care il gasiti aici:
`http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html`.

Fie urmatorul cod in Haskell (pe care il veti pune intr-un fisier):

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

asoc :: (Int -> Maybe Int) -> (Int -> Maybe Int) -> (Int -> Maybe Int) -> Int -> Bool
asoc f g h x = undefined

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

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

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

Exercitiul 1. Intelegeti functionarea operatiilor monadice (`>>=` si `return`). Inspectati si tipul lor de date! (spoiler mai jos)

```
-- spoiler
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a

> return 3 :: Maybe Int
Just 3

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

Aminititi-va de operatia `>>`, studiata in laboratorul cu `IO()`. Aceasta se exprima astfel:

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

> (Just 3) >> Nothing
Nothing

> (Just 3) >> (Just 6)
Just 6
```

Exercitiul 2. Aveti definit operatorul `<=<` de compunere a functiilor imbogatite. Creati singuri exemple prin care sa intelegeti functionarea acestui operator.

Exercitiul 3. Definiti proprietatea de asociativitate, adica sa respecte

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

Exercitiul 4. Uitati-va la functiile `pos` si `foo`.

- Intelegeti ce face functia `foo`;
- Definiti functia `foo` utilizand `do`-notation.

Despre `do`-notation:

$e >>= \backslash x- > rest$	$do\ x < -e\ ;\ rest$
$e >>= \backslash\_ > rest$	$do\ x\ ;\ rest$
$e >> rest$	$do\ x\ ;\ rest$

De exemplu

```
e1 >>= \x1 -> e2 >> e3
```

se traduce in

```
do
    x1 <- e1
    e2
    e3
```

iar

```
e1 >>= \x1 ->
e2 >>= \x2 ->
e3 >>= \_ ->
e4 >>= \x4 ->
e5
```

se traduce in

```
do
    x1 <- e1
    x2 <- e2
    e3
    x4 <- e4
    e5
```

Exercitiul 5. Vrem sa definim functia `addM :: Maybe Int -> Maybe Int -> Maybe Int` care aduna doua valori de tip `Maybe Int`. Implementati aceasta functie utilizand ambele tipuri de scriere, dar utilizand si o varianta nemonadica.

Exemple de functionare:

```
> addM (Just 4) (Just 3)
Just 7

> addM (Just 4) Nothing
Nothing

> addM Nothing Nothing
Nothing
```

Fie urmatorul nou continut de fisier:

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

instance  Monad WriterS where
  return va = Writer (va, "")
  ma >>= k = let (va, log1) = runWriter ma
                 (vb, log2) = runWriter (k va)
             in  Writer (vb, log1 ++ log2)


instance  Applicative WriterS where
  pure = return
```

```
  mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)

instance  Functor WriterS where
  fmap f ma = pure f <*> ma

tell :: String -> WriterS ()
tell log = Writer ((), log)

logIncrement :: Int -> WriterS Int
logIncrement x = undefined

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

Acest fisier contine definitia monadei `Writer String`.

Exercitiul 6. Definiti functiile `logIncrement` si `logIncrement2` astfel incat functionarea sa fie in felul urmator:

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

Exercitiul 7. Modificati definitia monadei astfel incat sa produca lista mesajelor logate, si nu concatenarea lor.

```
> runWriter $ logIncrement N 2 4
(6, ["increment:2", "increment:3", "increment:4", "increment:5"])
```