Programare funcțională

Introducere în programarea funcțională folosind Haskell
C11- Seria 24

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Recap

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

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

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

Notația do pentru monade

```haskell
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
```

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

```haskell
binding' :: IO ()
binding' =
  getLine >>= putStrLn

binding :: IO ()
binding = do
  name <- getLine
  putStrLn name
```

Instanta de Monade pentru liste

Functiile din clasa **Monad** specializate pentru liste:

```
(>>=)  :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
         -- [ys | x <- xs, ys <- f x]
```

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven xs = do
    x <- xs
    if even x
       then [x*x, x*x]
       else [x*x]
*C10> twiceWhenEven [1,2,3,4]
*> [1,4,4,9,16,16]
```

Monade (cont.)

Monada Maybe (a funcțiilor parțiale)

```haskell
data Maybe a = Nothing | Just a

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a

instance Monad Maybe where
  return = Just

  Just va >>= f   = f va
  Nothing >>= _   = Nothing
```

Monada Maybe – exemplu

```haskell
radical :: Float -> Maybe Float
radical x
    | x >= 0 = return (sqrt x)
    | x < 0  = Nothing

-- a * x^2 + b * x + c = 0
solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return (negate c / b)
solEq2 a b c = do
  rDelta <- radical (b * b - 4 * a * c)
  return ((negate b + rDelta) / (2 * a))
```

Monada Maybe – exemplu

```haskell
-- a * x^2 + b * x + c = 0
solEq2All :: Float -> Float -> Float -> Maybe [Float]
solEq2All 0 0 0 = return [0]
solEq2All 0 0 c = Nothing
solEq2All 0 b c = return [negate c / b]
solEq2All a b c = do
        rDelta <- radical (b * b - 4 * a * c)
        let s1 = (negate b + rDelta) / (2 * a)
        let s2 = (negate b - rDelta) / (2 * a)
        return [s1,s2]
```

Monada Either (a excepțiilor)

```haskell
data Either err a = Left err | Right a

(>>=) :: Either err a -> (a -> Either err b) ->
          Either err b
return :: a -> Either err a

instance Monad (Either err) where
     return = Right

     Right va >>= f = f va
          err >>= _ = err
  -- Left verr >>= _ = Left verr
```

Monada Either – exemplu

```
radical :: Float -> Either String Float
radical x
   | x >= 0 = return (sqrt x)
   | x < 0  = Left "radical: argument negativ"

-- a * x^2 + b * x + c = 0
solEq2 :: Float -> Float -> Float -> Either String Float
solEq2 0 0 0 = return 0
solEq2 0 0 c = Left "ecuatie: fara solutie"
solEq2 0 b c = return (negate c / b)
solEq2 a b c = do
   rDelta <- radical (b * b - 4 * a * c)
   return ((negate b + rDelta) / (2 * a))
```

Monada Writer (variantă simplificată)

```haskell
newtype Writer log a = Writer {runWriter :: (a, log)}
-- a este parametru de tip

tell :: log -> Writer log ()
tell msg = Writer ((), msg)

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

Monada Writer (varianta lungă)

```
class Semigroup a where
  (<>) :: a -> a -> a

class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)

newtype Writer log a = Writer {runWriter :: (a, log)}

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= f = let (va, log1) = runWriter ma
                 (vb, log2) = runWriter (f va)
             in Writer (vb, log1 `mappend` log2)
```

Monada Writer - Exemplu logging

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

tell :: log -> Writer log ()
tell msg = Writer ((), msg)

logIncrement :: Int -> Writer String Int
logIncrement x = do
   tell ("increment: " ++ show x ++ "\n")
   return (x + 1)

logIncrement2 :: Int -> Writer String Int
logIncrement2 x = do
   y <- logIncrement x
   logIncrement y

*C11> runWriter (logIncrement2 13)
(15,"increment: 13\nincrement: 14\n")
```

Monada Reader (stare nemodificabilă)

```haskell
newtype Reader env a = Reader {runReader :: env -> a}
-- runReader :: Reader env a -> env -> a

ask :: Reader env env
ask = Reader id

instance Monad (Reader env) where
  return = Reader const
  -- return x = Reader (\_ -> x)

  ma >>= k = Reader f
     where
     f env = let va = runReader ma env
                in runReader (k va) env
```

Monada Reader - exemplu

```haskell
tom :: Reader String String
tom = do
  env <- ask -- gives the environment (here a String)
  return (env ++ " This is Tom.")

jerry :: Reader String String
jerry = do
  env <- ask
  return (env ++ " This is Jerry.")

tomAndJerry :: Reader String String
tomAndJerry = do
    t <- tom
    j <- jerry
    return (t ++ "\n" ++ j)

runJerryRun :: String
runJerryRun = runReader tomAndJerry "Who is this?"
```

Pe data viitoare!