Curs 1

2020-2021

Fundamentele Limbajelor de Programare

Cuprins

Haskell: Clasa de tipuri **Monad**

Clasa de tipuri **Monad**

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

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

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

Functor și Applicative definiți cu **return** și >>=

```
instance Monad M where
  return a = ...
  ma >>= k = ...

instance Applicative M where
  pure = return
  mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)
   -- mf >>= (\f -> ma >>= (\a -> return (f a)))

instance Functor F where          -- ma >>= \a -> return (f a)
  fmap f ma = pure f <*> ma       -- ma >>= (return . f)
```

Notația **do** pentru monade

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

Notația **do** pentru monade

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

De exemplu

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

devine

Notația **do** pentru monade

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

De exemplu

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

devine

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

Haskell: Monade standard

Exemple de efecte laterale

I/O	Monada **IO**
Parțialitate	Monada **Maybe**
Excepții	Monada **Either**
Nedeterminism	Monada [] (listă)
Logging	Monada Writer
Memorie read-only	Monada Reader
Stare	Monada State

Monada **Maybe** (a rezultatelor parțiale)

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

Monada **Maybe** (a rezultatelor parțiale)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just va  >>= k   = k va
  Nothing >>= _    = Nothing
```

Monada **Maybe** (a rezultatelor parțiale)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just va  >>= k   = k va
  Nothing >>= _    = Nothing

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

solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0          -- a * x^2 + b * x + c = 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 **Either** (a excepțiilor)

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

Monada **Either** (a excepțiilor)

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

instance Monad (Either err) where
  return = Right
  Right va >>= k  = k va
  err      >>= _  = err    -- Left verr >>=_ = Left verr
```

Monada **Either** (a excepțiilor)

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

instance Monad (Either err) where
  return = Right
  Right va >>= k  = k va
  err      >>= _  = err    -- Left verr >>=_ = Left verr

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

solEq2 :: Float -> Float -> Float -> Either String Float
solEq2 0 0 0 = return 0          -- a * x^2 + b * x + c = 0
solEq2 0 0 c = Left "Nu are solutii"
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 listelor (a rezultatelor nedeterministe)

```
instance Monad [] where
  return va = [va]
  ma >>= k = [vb | va <- ma, vb <- k va]
```

Rezultatul nedeterminist e dat de lista tuturor valorilor posibile.

Monada listelor (a rezultatelor nedeterministe)

```
instance Monad [] where
  return va = [va]
  ma >>= k = [vb | va <- ma, vb <- k va]
```

Rezultatul nedeterminist e dat de lista tuturor valorilor posibile.

```
radical :: Float -> [Float]
radical x | x >= 0 = [negate (sqrt x), sqrt x]
          | x < 0  = []

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []                    -- a * x^2 + b * x + c = 0
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)

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

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

Monada Writer (variantă simplificată)

```
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 >>= k =   let (va, log1) = runWriter ma
                   (vb, log2) = runWriter (k va)
               in Writer (vb, log1 ++ log2)
```

Monada Writer - Exemplu logging

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

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

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


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

Monada Writer (varianta lungă)

Clasa de tipuri Semigroup

O mulțime, cu o operație <> care ar trebui să fie asociativă

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

Clasa de tipuri Monoid

Un semigrup cu unitatea mempty. mappend este alias pentru <>.

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

Foarte multe tipuri sunt instanțe ale lui Monoid. Exemplul clasic: listele.

Monada Writer (varianta lungă)

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

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

Monada Reader (stare nemodificabilă)

```
newtype Reader env a = Reader { runReader :: env -> a }
-- inspecteaza starea curenta
ask :: Reader env env
ask = Reader id
-- modifica starea doar pentru computatia data
local :: (env -> env) -> Reader env a -> Reader env a
local f r = Reader (runReader r . f)
```

Monada Reader (stare nemodificabilă)

```
newtype Reader env a = Reader { runReader :: env -> a }
-- inspecteaza starea curenta
ask :: Reader env env
ask = Reader id
-- modifica starea doar pentru computatia data
local :: (env -> env) -> Reader env a -> Reader env a
local f r = Reader (runReader r . f)

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: mediu de evaluare

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

data Prop ::= Var String | Prop :&: Prop
type Env = [(String, Bool)]

var :: String -> Reader Env Bool
var x = do
          env <- ask
          fromMaybe False (lookup x env)

eval :: Prop -> Reader Env Bool
eval (Var x) = var x
eval (p1 :&: p2) = do
  b1 <- eval p1
  b2 <- eval p2
  return (b1 && b2)
```

Monada State

```
newtype State state a =
    State { runState :: state -> (a, state) }
```

Monada State

```
newtype State state a =
    State { runState :: state -> (a, state) }

instance Monad (State state) where
  return va = State (\s -> (va, s))
-- return va = State f where f s = (va, s)

  ma  >>= k =
      State $ \s -> let (va, news) = runState ma s
                    in runState (k va) news
-- ma :: State state a
-- k :: a -> State state b
-- s :: state
-- runState ma :: state -> (a, state)
-- (va, news) :: (a, state)  = runState ma s
-- k va :: State state b
-- runState (k va) news :: (b, state)
-- ma >>= k :: State state b
```

Monada State

```
newtype State state a =
    State { runState :: state -> (a, state) }

instance Monad (State state) where
  return va = State (\s -> (va, s))
  ma  >>= k =
      State $ \s -> let (va, news) = runState ma s
                    in runState (k va) news
```

Funcții ajutătoare:

```
get :: State state state          -- obtine starea curenta
get = State (\s -> (s, s))

set :: state -> State state () -- seteaza starea curenta
set s = State (const ((), s))

modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))    -- modifica starea
```

Monada State - exemplu "random"

```
newtype State state a = State{runState :: state->(a,state)}
get :: State state state
get = State (\s -> (s,s))
modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))
```

Monada State - exemplu "random"

```
newtype State state a = State{runState :: state->(a,state)}
get :: State state state
get = State (\s -> (s,s))
modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))

cMULTIPLIER, cINCREMENT :: Word32
cMULTIPLIER = 1664525 ; cINCREMENT = 1013904223

rnd, rnd2 :: State Word32 Word32
rnd = do modify (\seed -> cMULTIPLIER * seed + cINCREMENT)
         get
rnd2 = do r1 <- rnd
          r2 <- rnd
          return (r1 + r2)

Main> runState rnd2 0
(2210339985,1196435762)
```

Pe săptămâna viitoare!