Curs 11

Cuprins

Monade

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

Cum interacționezi cu mediul extern, păstrând puritatea?

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

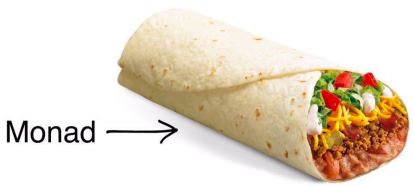
Cum interacționezi cu mediul extern, păstrând puritatea?
Se folosesc *monade*!

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este un burrito.

  https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/

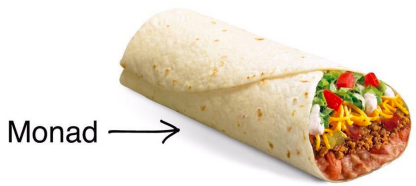


https://twitter.com/monadburritos

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este un burrito.
  https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/



https://twitter.com/monadburritos

- "All told, a monad in X is just a monoid in the category of endofunctors in X, with product x replaced by composition of endofunctors and unit set by the identity endofunctor."
  Saunders Mac Lane, Categories for the Working Mathematician, 1998.

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

- Funcţie îmbogăţită: $x \mapsto$

  ştiind x, putem să extragem y şi producem un efect

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

- Funcţie îmbogăţită: $x \mapsto$

ştiind x, putem să extragem y şi producem un efect

Referinţe:

https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/

https://bartoszmilewski.com/2016/11/30/monads-and-effects/

Funcţii îmbogăţite şi efecte

Funcţie îmbogăţită: $x \mapsto$ 

Exemplu

- Folosind tipul **Maybe** a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x)
```

Funcţii îmbogăţite şi efecte

Funcţie îmbogăţită: $x \mapsto$ 

Exemplu

- Folosind un tip care are ca efect un mesaj

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

f :: Int -> Writer String Int
f  x = if x < 0 then (Writer (-x, "negativ"))
                else (Writer (x, "pozitiv"))
```

Logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

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

Observații

- datele de tip Writer **log** a sunt definite folosind înregistrări
- o dată de tip Writer **log** a are una din formele
  Writer (va,vlog) sau Writer {runWriter = (va,vlog)}
  unde va :: a și vlog :: **log**
- runWriter este funcția proiecție:
  runWriter :: Writer **log** a –> (a, **log**)
  de exemplu runWriter (Writer (1,"msg")) = (1,"msg")

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Valoarea de tip b este transmisă direct funcţiei g.

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

  Valoarea de tip b este transmisă direct funcţiei g.

- Ce facem dacă

```
f :: a -> m b , g :: b -> m c
```

  unde m este un constructor de tip care îmbogăţeşte tipul?

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

  Valoarea de tip b este transmisă direct funcţiei g.

- Ce facem dacă

```
f :: a -> m b , g :: b -> m c
```

  unde m este un constructor de tip care îmbogăţeşte tipul?

  De exemplu,
  - m = **Maybe**
  - m = Writer **log**

- Atenţie! m trebuie să aibă un singur argument.

Compunerea funcţiilor

```
f :: a -> m b , g :: b -> m c
```

unde m este un constructor de tip care îmbogăţeşte tipul.

Vrem să definim o "compunere" pentru funcţii îmbogăţite

```
(<=<) :: (b -> m c) -> (a -> m b) -> a -> m c
```

Atunci când definim g <=< f trebuie să extragem valoarea întoarsă de f şi să o trimitem lui g.

Exemplu: logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

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

logIncrement :: Int -> Writer String Int
logIncrement x = Writer
    (x + 1, "Called increment with argument " ++ show
        x ++ "\n")
```

Problemă: Cum calculăm logIncrement (logIncrement x)?

Exemplu: logging în Haskell

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

logIncrement :: Int -> Writer String Int
logIncrement x = Writer
    (x + 1, "Called increment with argument " ++ show
        x ++ "\n")


logIncrement2 :: Int -> Writer String Int
logIncrement2 x = let (y, log1) = runWriter (
    logIncrement x)
                      (z, log2) = runWriter (
                          logIncrement y)
                  in  Writer (z, log1 ++ log2)
```

Cum compunem funcții cu efecte laterale

Problema generală

Dată fiind funcția f :: a -> m b și funcția g :: b -> m c, vreau să obțin o funcție g <=< f :: a -> m c care este „compunerea” lui g și f, propagând efectele laterale.

Exemplu

```
> logIncrement x = Writer (x + 1, "Called increment
    with argument " ++ show x ++ "\n")

> logIncrement <=< logIncrement $ 3
Writer {runWriter = (5,"Called increment with
    argument 3\nCalled increment with argument 4\n")}
```

Observaţie: Funcţia (<=<) este definită în Control.**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

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
- în Control.**Monad** sunt definite
  - f >=>g = \x -> f x >>= g
  - (<=<) = **flip** (>=>)

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
- în Control.**Monad** sunt definite
  - f >=>g = \x -> f x >>= g
  - (<=<) = **flip** (>=>)

Applicative va fi discutată mai târziu

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

În Haskell, monada este o clasă de tipuri!

Exemple: monada **Maybe**

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b

> (lookup 3 [(1,2), (3,4)]) >>= (\x -> if (x<0) then
    Nothing else (Just x))
Just 4

> (lookup 3 [(1,2), (3,-4)]) >>= (\x -> if (x<0) then
    Nothing else (Just x))
Nothing

> (lookup 3 [(1,2)]) >>= (\x -> if (x<0) then Nothing
    else (Just x))
Nothing
```

Proprietățile monadelor

Asociativitate şi element neutru

Operația <=< de compunere a funcţiilor îmbogăţite este asociativă şi are element neutru **return**

Proprietățile monadelor

Asociativitate şi element neutru

Operația <=< de compunere a funcţiilor îmbogăţite este asociativă şi are element neutru **return**

- Element neutru (la dreapta): g <=< **return** = g

```
(return x) >>= g = g x
```

- Element neutru (la stânga): **return** <=< g = g

```
x >>= return = x
```

- Associativitate: h <=< (g <=< f) = (h <=< g) <=< f

```
(f >>= g) >>= h = f >>= (\ x -> (g x >>= h))
```

Notația **do** pentru monade

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

Notația **do** pentru monade

Notaţia cu operatori	Notaţia **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

Notaţia cu operatori	Notaţia **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
```

Notația **do** pentru monade

De exemplu

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

devine

Notația **do** pentru monade

De exemplu

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

devine

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

Exemple de efecte laterale

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

Monada **Maybe**(a funcțiilor parțiale)

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

Monada **Maybe**(a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

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

Monada **Maybe**(a funcțiilor 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
```

Monada **Maybe**(a funcțiilor parțiale)

```
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)

```
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
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 Writer (variantă simplificată)

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

Monada Writer (variantă simplificată)

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

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 (variantă simplificată)

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

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)


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 = (<>)
```

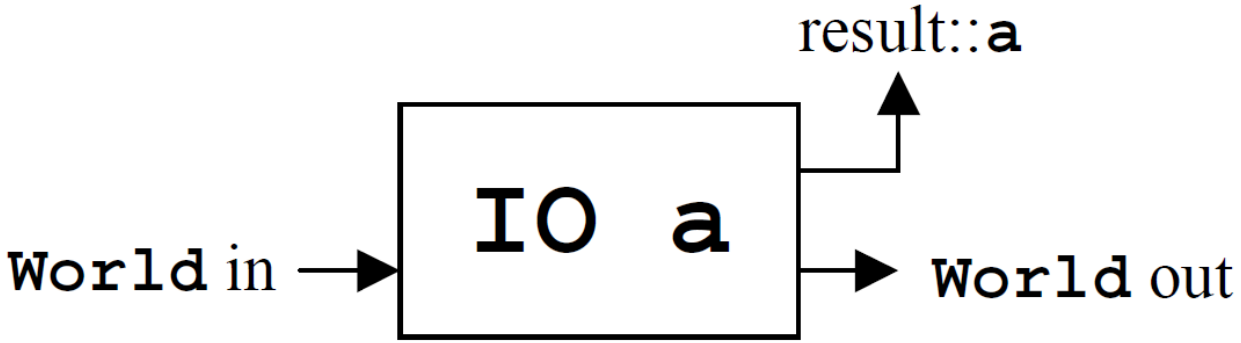
Monada Writer

```
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 **IO**

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Comenzi cu valori

- **IO** () corespunde comenzilor care nu produc rezultate

```
putChar :: Char -> IO ()
putStr  :: String -> IO ()
putStrLn :: String -> IO ()
```

- În general, **IO** a corespunde comenzilor care produc rezultate de tip a.

```
getChar :: IO Char
getLine  :: IO String
```

Compilarea programelor

Orice comandă **IO a** poate fi executată în interpretor, dar

Programele Haskell pot fi compilate

Fișierul scrie.hs:

```
main :: IO ()
main = putStrLn "?!"
```

```
08-io$ ghc scrie.hs
[1 of 1] Compiling Main  (scrie.hs, scrie.o)
Linking scrie.exe ...
08-io$ ./scrie
?!
```

Funcţia executată este **main**

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)
```

Interpretoare monadice

Sintaxă abstractă

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
```

Valori și medii de evaluare

```
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

instance Show Value where
 show (Num x) = show x
 show (Fun _) = "<function>"
 show Wrong   = "<wrong>"
```

Observații

- Vom interpreta termenii în valori ‘M Value‘, unde ‘M‘ este o monadă; variind monada, se obțin comportamente diferite;
- ‘Wrong‘ reprezintă o eroare, de exemplu adunarea unor valori care nu sunt numere sau aplicarea unui termen care nu e funcție.

Evaluare - variabile și valori

```
type Environment = [(Name, Value)]
```

Interpretarea termenilor în monada ‘M‘

```
interp :: Term -> Environment -> M Value
interp (Var x) env = lookupM x env
interp (Con i) _ = return $ Num i
interp (Lam x e) env = return $
  Fun $ \ v -> interp e ((x,v):env)

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> return Wrong
```

Evaluare - adunare

```
interp (t1 :+: t2) env = do
  v1 <- interp t1 env
  v2 <- interp t2 env
  add v1 v2
```

Interpretarea adunării în monada ‘M‘

```
add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = return Wrong
```

Evaluare - aplicarea funcțiilor

```
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
  apply f v
```

Interpretarea aplicării funcțiilor în monada 'M'

```
apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = return Wrong

-- k :: Value -> M Value
```

Testarea interpretorului

```
test :: Term -> String
test t = showM $ interp t []
```

unde

```
showM :: Show a => M a -> String
```

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Testarea interpretorului

```
test :: Term -> String
test t = showM $ interp t []
```

unde

```
showM :: Show a => M a -> String
```

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Exemplu de program

```
pgm :: Term
pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

test pgm  -- apelul pentru testare
```

Interpretor monadic

```
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

interp :: Term -> Environment -> M Value
```

În continuare vom înlocui monada M cu:

- Identity
- **Maybe**
- **Either String**
- Writer

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Pentru a particulariza interpretorul definim

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Pentru a particulariza interpretorul definim

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

Obținem interpretorul standard, asemănător celui discutat pentru limbajul Mini-Haskell.

Interpretare folosind monada ‘Identity‘

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

*Var0> test pgm
"42"
```

Interpretare în monada ‘Maybe‘ (opțiune)

```
data Maybe a = Nothing | Just a

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

Putem renunța la valoarea ‘Wrong‘, folosind monada ‘Maybe‘

```
type M a = Maybe a

showM :: Show a => M a -> String
showM (Just a) = show a
showM Nothing  = "<wrong>"
```

Interpretare în monada ‘Maybe‘

Putem acum înlocui rezultatele ‘Wrong‘ cu ‘Nothing‘

```
type M a = Maybe a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> Nothing

add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = Nothing

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = Nothing
```

Interpretare în monada ‘Either String‘

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

instance Monad (Either err) where
  return = Right
  Right a >>= k  = k a
  err     >>= _  = err
```

Putem nuanța erorile folosind monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a
```

Interpretare în monada ‘Either String‘

Putem acum înlocui rezultatele ‘Wrong‘ cu valori ‘Left‘

```
type M a = Either String a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> Left ("unbound variable " ++ x)

add :: Value -> Value -> M Value
add (Num i) (Num j) = return $ Num $ i + j
add v1 v2           = Left $
  "Expected numbers: " ++ show v1 ++ ", " ++ show v2

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply v _       = Left $
  "Expected function: " ++ show v
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

*Var2> test pgm
"Success: 42"
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

*Var2> test pgm
"Success: 42"

pgmE = App (Var "x") ((Con 10) :+:  (Con 11))

*Var2> test pgmE
"Error: unbound variable x"
```

Monada ‘Writer‘

Este folosită pentru a acumula (logging) informație produsă în timpul execuției.

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

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

Funcție ajutătoare

```
tell :: log -> Writer log ()
tell log = Writer ((), log)  -- produce mesajul
```

Interpretare în monada ‘Writer‘

Adăugarea unei instrucțiuni de afișare
data Term = ... | Out Term

```
type M a = Writer String a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ "  Value: " ++ show a
  where (a, w) = runWriter ma


interp (Out t) env = do
  v <- interp t env
  tell (show v ++ "; ")
  return v
```

- Out t se evaluează la valoarea lui t, cu efectul lateral de a adăuga valoarea la șirul de ieșire.

Interpretare în monada ‘Writer‘

```
data Term = ... | Out Term

type M a = Writer String a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ "  Value: " ++ show a
  where (a, w) = runWriter ma

pgmW = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Out (Con 10)) :+: (Out (Con 11)))

> test pgm
"Output: 10; 11; Value: 42"
```

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.

```
Prelude> [1,2,3] >>= \v -> [v+1]
[2,3,4]

Prelude> [1,2,3] >>= \v -> return (v+1)
[2,3,4]

Prelude> [4,9,25] >>= \x -> [(sqrt x), -(sqrt x)]
[2.0,-2.0,3.0,-3.0,5.0,-5.0]
```

Monada listelor (a rezultatelor nedeterministe)

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

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

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []
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)
```

Interpretare în monada listelor

Adăugarea unei instrucțiuni nedeterministe

```
data Term = ... | Amb Term Term | Fail

type M a = [a]

showM :: Show a => M a -> String
showM = show

interp Fail _ = []
interp (Amb t1 t2) env = interp t1 env ++ interp t2
    env

pgmN = (App (Lam "x" (Var "x" :+: Var "x"))
       (Amb (Con 1) (Con2)))
> test pgmN
"[2,4]"
```

Pe săptămâna viitoare!