Programare declarativă[1]

Intrare/Ieșire

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Comenzi în Haskell

Despre intenție și acțiune

[1] S. Peyton-Jones, Tackling the Awkward Squad: ...

- [1] A purely functional program implements a function; it has no side effect.
- [1] Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, ...

Exemplu

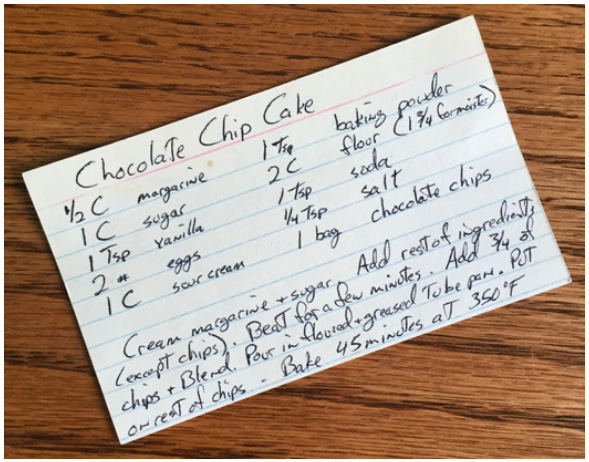
```
putChar :: Char -> IO ()
>     putChar '!'
```

reprezintă o comandă care, dacă va fi executată, va afișa un semn de exclamare.

Mind-Body Problem - Rețetă vs Prăjitură

http://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

c :: Cake

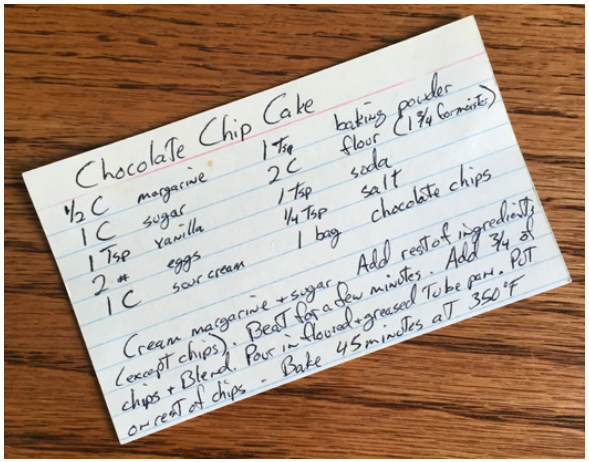


r :: Recipe Cake

Mind-Body Problem - Rețetă vs Prăjitură

http://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

c :: Cake



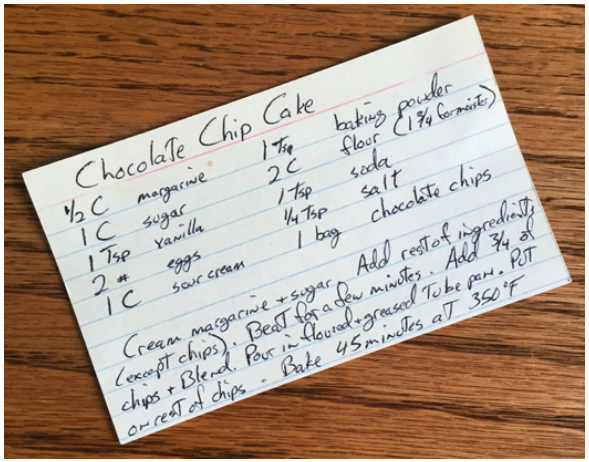
r :: Recipe Cake

IO este o rețetă care produce o valoare de tip **a**.

Mind-Body Problem - Rețetă vs Prăjitură

http://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

c :: Cake



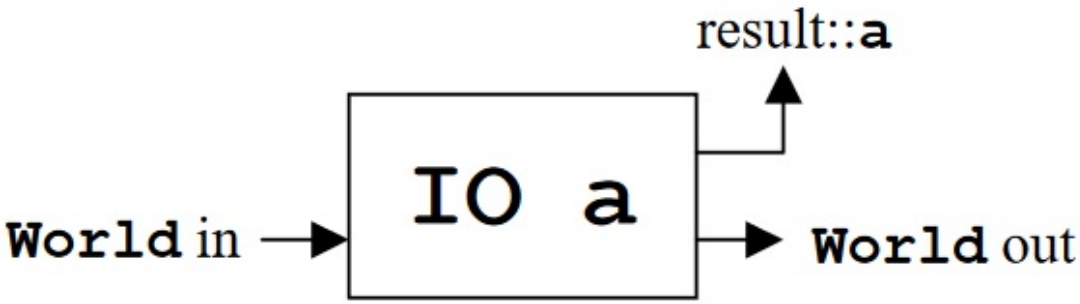
r :: Recipe Cake

IO este o rețetă care produce o valoare de tip **a**.

Motorul care citeste si executa instructiunile IO se numeste Haskell Runtime System (RTS). Acest sistem reprezinta legatura dintre programul scris si mediul în care va fi executat, împreuna cu toate efectele si particularitatile acestuia.

Comenzi în Haskell

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



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

Combină două comenzi!

```
(>>) :: IO () -> IO () -> IO ()
putChar :: Char -> IO ()
```

Exemplu

```
putChar '?' >> putChar '!'
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare .

Afișează un șir de caractere

```
putStr :: String -> IO ()
putStr []     = done
putStr (x:xs) = putChar x >> putStr xs
```

Observație:

```
done :: IO ()
```

reprezintă o comandă care, dacă va fi executată, nu va face nimic.

Exemplu

```
putStr "?!" == putChar '?' >> (putChar '!' >> done)
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare.

putStr folosind funcționale

```
putStr      :: String -> IO ()
putStr      = foldr (>>) done . map putChar
```

Afișează și treci pe rândul următor

```
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

putStr folosind funcționale

```
putStr      :: String -> IO ()
putStr      = foldr (>>) done . map putChar
```

Afișează și treci pe rândul următor

```
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

(**IO** (), (>>), done) e monoid

```
m >> done           =   m
done >> m           =   m
(m >> n) >> o       =   m >> (n >> o)
```

Când sunt executate comenzile?

main

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
?!
```

Funcția executată este **main**

Validitatea raționamentelor

Raționamentele substitutive sunt valabile

În Haskell

Expresii

```
(1+2) * (1+2)
```

este echivalentă cu expresia

```
let x = 1+2 in x * x
```

și se evaluează amândouă la 9

Comenzi

```
putStr "HA!" >> putStr "HA!"
```

este echivalentă cu

```
let m = putStr "HA!" in m >> m
```

si amândouă afișează "HA!HA!".

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

Referential transparency

orice expresie poate fi înlocuită cu valoare ei

```
addExclamation :: String -> String
addExclamation s = s ++ "!"

main = putStrLn (addExclamation "Hello")
Prelude> main
Hello!

main = putStrLn ("Hello" ++ "!")
Prelude> main
Hello!
```

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Nu putem înlocui **getLine** cu valoarea ei!

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Nu putem înlocui **getLine** cu valoarea ei!

Soluția: **getLine** are tipul **IO String**

Comenzi cu valori

Comenzi cu valori

- **IO** () corespunde comenzilor care nu produc rezultate
  - () este tipul unitate care conține doar valoarea ()

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

Comenzi cu valori

- **IO** () corespunde comenzilor care nu produc rezultate
  - () este tipul unitate care conține doar valoarea ()

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

Exemplu: citește un caracter

- **IO Char** corespunde comenzilor care produc rezultate de tip **Char**

```
getChar :: IO Char
```

- Dacă „șirul de intrare” conține "abc"
- atunci **getChar** produce:
  - 'a'
  - șirul rămas de intrare "bc"

Produu o valoare fără să faci nimic!

```
return :: a -> IO a
```

Asemănatoar cu done, nu face nimic, dar produce o valoare.

Exemplu

```
return ""
```

- Dacă „șirul de intrare” conține "abc"
- atunci **return** "" produce:
  - valoarea ""
  - șirul (neschimbat) de intrare "abc"

Combinarea comenzilor cu valori

Operatorul de legare / bind

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Exemplu

```
getChar >>= \x -> putChar (toUpper x)
```

- Dacă „șirul de intrare” conține "abc"
- atunci comanda de mai sus, atunci când se execută, produce:
  - ieșirea "A"
  - șirul rămas de intrare "bc"

Operatorul de legare / bind

Mai multe detalii

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- Dacă fiind o comandă care produce o valoare de tip a
  m :: **IO** a
- Data fiind o funcție care pentru o valoare de tip a se evaluează la o comandă de tip b
  k :: a -> **IO** b
- Atunci
  m >>= k :: **IO** b
  este comanda care, dacă se va executa:
  - Mai întâi efectuează m, obținând valoarea x de tip a
  - Apoi efectuează comanda k x obținând o valoare y de tip b
  - Produce y ca rezultat al comenzii

Citește o linie!

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

Exemplu

Dat fiind șirul de intrare "abc\ndef", getLine produce șirul "abc" și șirul rămas de intrare e "def"

Comenzile sunt cazuri speciale de comenzi cu valori

done e caz special de return

```
done        :: IO ()
done        = return ()
```

>> e caz special de >>=

```
(>>)        :: IO () -> IO () -> IO ()
m >> n      = m >>= \ () -> n
```

Operatorul de legare e similar cu **let**

Operatorul **let**

```
let x = m in n
```

let ca aplicație de funcții

```
(\ x -> n) m
```

Operatorul de legare

```
m >>= \ x -> n
```

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo

main :: IO ()
main = echo
```

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo

main :: IO ()
main = echo
```

Test

```
$ runghc Echo.hs
One line
ONE LINE
And, another line!
AND, ANOTHER LINE!
```

Notația do

Citirea unei linii în notație „do”

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

Echivalent cu:

```
getLine :: IO String
getLine = do {
            x <- getChar;
            if x == '\n' then
              return []
            else do {
              xs <- getLine;
              return (x:xs)
            }
          }
```

Echo în notația „do”

```
echo :: IO ()
echo = getLine >>= \line ->
         if line == "" then
           return ()
         else
           putStrLn (map toUpper line) >>
           echo
```

Echivalent cu

```
echo :: IO ()
echo = do {
           line <- getLine;
           if line == "" then
             return ()
           else do {
             putStrLn (map toUpper line);
             echo
           }
         }
```

Notația „do” în general

- Fiecare linie x <– e; ... devine e >>= \x –> ...
- Fiecare linie e; ... devine e >> ...

De exemplu

```
do { x1 <– e1;
     x2 <– e2;
     e3;
     x4 <– e4;
     e5;
     e6 }
```

e echivalent cu

```
e1  >>= \x1 –>
e2  >>= \x2 –>
e3  >>
e4  >>= \x4 –>
e5  >>
e6
```

Clasa de tipuri **Monad**

```
class 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 comenzilor 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 comenzilor

În Haskell, monada este o clasă de tipuri!

Kinds (tipuri de tipuri)

Observăm că m în definiția de mai sus este un constructor de tip.

Kinds (tipuri de tipuri)

Observăm că m în definiția de mai sus este un constructor de tip.

În Haskell, valorile sunt clasificate cu ajutorul *tipurilor*:

```
Prelude> :t "as"
"as" :: [Char]
```

Constructorii de tipuri sunt la rândul lor clasificați în *kind*-uri:

```
Prelude> :k Char
*     -- constructor de tip fara argumente
Prelude> :k []
[] :: * -> *   -- constructor de tip cu un argument
```

Constructorii de tip pot fi și ei grupați în clase.

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.

Ce este o monadă?

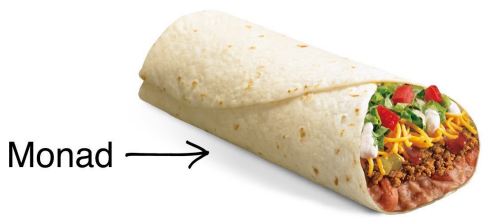
Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.
- "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.

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.
- "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.
- O monadă este un burrito. `https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/`



`https://twitter.com/monadburritos`

Funcții "îmbogățite" și programarea cu efecte

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

  știind x, obținem direct y

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

  știind x, obținem direct y

- Funcție îmbogățită: $x \mapsto$

  știind x, putem să extragem y și producem un efect

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

  știind x, obținem direct y

- Funcție îmbogățită: $x \mapsto$ 

  știind x, putem să extragem y și producem un efect

Referințe:

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

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

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$ 

Exemple

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$

Exemple

- Folosind tipul **Maybe** a

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

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$

Exemple

- Folosind tipul **Maybe** a

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

- Folosind un tip care are ca efect modificarea unei stari

```
newtype State st a = State { runState :: st -> (a, st)}
rnd :: State Word32 Word32
rnd = State f
  where f seed = (seed', seed')
        seed' = cMULTIPLIER * seed + cINCREMENT
```

Exemple: bind (>>=)

- monada **Maybe**

```
> (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
```

Exemple: bind (>>=)

- monada listelor

```
> f = (\x -> if (x>=0) then [sqrt x,-sqrt x)] else [])

> [4,8] >>= f
```

Exemple: bind (>>=)

- monada listelor

```
> f = (\x -> if (x>=0) then [sqrt x,-sqrt x)] else [])

> [4,8] >>= f
[2.0,-2.0,2.8284271247461903,-2.8284271247461903]

> [4,8] >>= f >>= f
```

Exemple: bind (>>=)

- monada listelor

```
> f = (\x -> if (x>=0) then [sqrt x,-sqrt x)] else [])

> [4,8] >>= f
[2.0,-2.0,2.8284271247461903,-2.8284271247461903]

> [4,8] >>= f >>= f
[1.4142135623730951,-1.4142135623730951,
1.6817928305074292,-1.6817928305074292]
```

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
```

Exemple de efecte laterale

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

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 listelor (a funcțiilor nedeterministe)

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

Rezultatul funcției e lista tuturor valorilor posibile.

Monada listelor (a funcțiilor nedeterministe)

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

Rezultatul funcției e 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)
```

Functor / Applicative / **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
```

Clasa **Monad** este o extensie a clasei Applicative!

Functor: efecte laterale

Functor

Schimbă rezultatul: efectele laterale rămân aceleași

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

Exemplu — liste

Dată fiind o funcție `f :: a -> b` și o listă *la* de elemente de tip a, vreau să să obțin o lista de elemente de tip b transformând fiecare element din *la* folosind funcția f.

```
instance Functor [] where
  fmap = map
```

Functor: efecte laterale

Functor

Schimbă rezultatul: efectele laterale rămân aceleași

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

Instanță pentru tipul optiune fmap :: (a -> b) -> **Maybe** a -> **Maybe** b

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Exemple

```
Main> fmap (*2) [1..3]
[2,4,6]
Main> fmap (*2) (Just 200)
Just 400
Main> fmap (*2) Nothing
Nothing
Main> fmap (*2) (+100) 4
208
Main> fmap (*2) (Right 6)
Right 12
Main> fmap (*2) (Left 135)
Left 135
Main> fmap (show . (*2) . read) getLine >>= putStrLn
123
246
```

Problemă

- Folosind fmap putem transforma o funcție `h :: a -> b` într-o funcție între computații cu efecte `fmap h :: m a -> m b`
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente E.g., cum trecem de la `h :: a -> b -> c` la `h' :: m a -> m b -> m c`?
- Putem încerca să folosim fmap
- dar, deoarece `h :: a -> (b -> c)` obtinem
  `fmap h :: m a -> m (b -> c)`
- Putem aplica `fmap h` la o valoare `ca :: m a` și obținem
  `fmap h ca :: m (b -> c)`

Problemă

Cum transformăm un obiect din `m (b -> c)` într-o funcție `m b -> m c`?

Clasa de tipuri Applicative

Definiție

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

- Orice instanță a lui Applicative trebuie să fie instanță a lui **Functor**

Instanță pentru tipul opțiune

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f  <*> x = fmap f x
```

Clasa de tipuri Applicative

Instanță pentru tipul opțiune

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f  <*> x = fmap f x
```

```
> pure "Hey" :: Maybe String
Just "Hey"
> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
```

Tipul listă (computație nedeterministă)

Instanță pentru tipul computațiilor nedeterministe (liste)

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

```
Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"]["Hello world","Hello happiness","Goodbye
    world","Goodbye happiness"]
Main> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Functor și Applicative pot fi 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)
```

Pe săptămâna viitoare!