# Programare declarativă

Ioana Leuștean, Ana Cristina Țurlea

ioana@fmi.unibuc.ro, ana.turlea@fmi.unibuc.ro

# Informații examen:

- Va avea loc în data de 23 ianuarie.
- Va fi pe calculatoare. Programarea pe săli va fi afișată pe moodle inainte de examen.
- Valorează 7 puncte din nota finala (1 pct oficiu și 2 pct testul de laborator).
- Condiția de promovabilitate: Nota finală cel puțin 5 ($5 > 4.99$).
- Acoperă **toată** materia.
- Materiale ajutătoare **doar cursul tipărit și legat**.

# Exerciții

În acest curs vom prezenta exerciții rezolvate:

- definierea funcției `insertAt` folosind `foldr`
- definirea unui interpretor folosind o combinație

  între monadele `Writer` și `Maybe`
- exerciții inspirate de jocul de șah

# `insertAt` folosind `foldr`

# `insertAt` folosind `foldr`

- Folosind proprietatea de universalitate a funcției `foldr`, definiți funcția

```
insertAt :: a -> [a] -> Int -> [a]

insertAt
    :: a
    -- ^elementul de inserat
    -> [a]
    -- ^lista în care se inserează
    -> Int
    -- ^poziția înainte de care se inserează
    -> [a]
```

care inserează un element într-o listă într-o poziție dată.

- Prima poziție din listă este 1
- Dacă poziția <= 1, se va insera pe prima poziție
- Dacă poziția > lungimea listei, se va adăuga la sfârșit.

# Exemple

```
> insertAt 1 [2,3,4] (-4)
[1,2,3,4]
> insertAt 1 [2,3,4] 0
[1,2,3,4]
> insertAt 1 [2,3,4] 1
[1,2,3,4]
> insertAt 1 [2,3,4] 2
[2,1,3,4]
> insertAt 1 [2,3,4] 3
[2,3,1,4]
> insertAt 1 [2,3,4] 4
[2,3,4,1]
> insertAt 1 [2,3,4] 5
[2,3,4,1]
> insertAt 1 [2,3,4] 100
[2,3,4,1]
```

# `foldr` - Proprietatea de universalitate

Dacă `g` este o funcție recursivă definită pe liste finite, `f` este o funcție,

iar `i` este un element astfel încât

```
g []     = i
g (x:xs) = f x (g xs)
```

atunci

```
g = foldr f i
```

*Pentru mai multe detalii citiți Cursul 3!*

# insertAt - o soluție

```
insertAt e xs n
  | n <= 1     = e:xs
  | null xs    = [e]
  | otherwise = (head xs) : insertAt e (tail xs)  (n-1)
```

**Atenție!**

Această soluție, deși este corectă, nu ne ajută în rezolvarea problemei.

# insertAt - cu recursie pe liste

Pentru a aplica proprietatea de universalitate, trebuie să definim operația de inserare prin recursie pe o listă.

```
insertAt e xs n = g xs e n

g []  e _      = [e]
g (x:xs) e n
  | n <=1      = e:x:xs
  | otherwise = x : (g xs e (n-1))
```

# Determinarea elementului `i` și a funcției `f`

Din

```
g []  e _ = [e]
```

deducem

```
 g [] =\e n -> [e]
```

deci `i = \e n -> [e]`

Astfel, `i :: a -> Int -> [a]` , iar funcția f va avea tipul

```
f :: a -> (a -> Int -> [a]) -> (a -> Int -> [a])
```

Va fi mai convenabil (cand o definim) sa ne gandim la `f` ca

```
f :: a -> (a -> Int -> [a]) -> a -> Int -> [a]
```

# Atenție la tipuri

```
i :: a
  -- ^Elementul de inserat
  -> Int
  -- ^pozitia de inserat (minus lungimea listei)
   -> [a]

f :: a
  -- ^elementul curent din listă
  -> (a -> Int -> [a])
  -- ^funcția care știe să insereze în restul listei
  -> a
  -- ^elementul de inserat
  -> Int
  -- ^poziția de inserat (minus lungimea deja parcursă)
  -> [a]
```

# Determinarea funcției f

Deoarece `f x (g xs) = g (x:xs)` obținem

```
f x (g xs) e n =  if (n <=1)  then  e:x:xs
                  else  x : (g xs e (n-1))
```

Notăm `u = g xs` și înlocuim în definiția lui `f` :

```
f x u e n =  if (n <=1)  then  e:x:xs
             else  x : (u e (n-1))
```

# Determinarea funcției f

Am obținut

```
f x u e n =  if (n <=1)  then  e:x:xs
                       else  x : (u e (n-1))
```

unde `u = g xs`.

**Atenție!**

În definiție apare `xs` , care nu poate fi definit folosind parametrii lui `f`.

Trebuie să găsim o definiție a lui `f` care depinde numai de parametrii săi!

# Determinarea funcției f

Observăm că `x:xs = g xs x 1`, deci avem

```
f x (g xs) e n =  if (n <=1)  then  e:(g xs x 1)
                  else  x : (g xs e (n-1))
```

și, notând `u = g xs`

```
f x u e n =  if (n <=1)  then  e:(u x 1)
             else  x : (u e (n-1))
```

# Determinarea funcției f

Am obținut

```
f x u e n =  if (n <=1)  then  e:(u x 1)
                       else  x : (u e (n-1))
```

adică

```
f x u = \e n -> if (n <=1)  then  e:(u x 1)
                           else  x : (u e (n-1))
i = \e n -> [e]
```

**Definiția lui** `insertAt` **cu** `foldr`

```
insertAt e xs n = (foldr f i) e n
```

# Definirea unui interpretor

# Limbajul

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
          | Out Term
  deriving (Show)
```

# Exemplu de program

```
pgm :: Term

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

# Monada Writer

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

instance  Monad Writer  where
  return a = Writer (a, "")
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
             in  Writer (b, log1 ++ log2)

instance Applicative Writer  where
  pure = return
  mf <*> ma = do { f <- mf; a <- ma; return (f a) }

instance Functor Writer  where
fmap f ma = pure f <*> ma
```

# Interpretorul folosind monada `Writer`

```
type M a = Writer a

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

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

type Environment = [(Name, Value)]

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

# Interpretorul folosind monada `Writer`

```haskell
interp :: Term -> Environment -> M Value
interp (Var x) env = get x env
interp (Con i) _ = return $ Num i
interp (t1 :+: t2) env = do
  v1 <- interp t1 env
  v2 <- interp t2 env
  add v1 v2

get :: Name -> Environment -> M Value
get x env = case [v | (y,v) <- env , x == y] of
  (v:_) -> return v
  _     -> return Wrong

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

# Interpretorul folosind monada `Writer`

```
interp (Lam x e) env =
  return $ Fun $ \ v -> interp e ((x,v):env)
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
  apply f v

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

# Interpretorul folosind monada `Writer`

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

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

# Exemplu

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

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

> test pgm
"Output: 10; 11; \nValue: 42"

pgmW4 = App (Var "y") (Lam "y" (Out (Con 3)))

> test pgmW
"Output: \nValue: <wrong>"
```

# Problemă

În continuare vom modifica programul astfel încât, în cazul apariției unei erori, se va întoarce rezultatul `Nothing` fără a afișa output-ul acumulat până în acel moment.

Pentru aceasta vom înlocui monada `Writer` cu o nouă monadă care combină `Writer` cu `Maybe`.

Definim

```
newtype MaybeWriter a = MW {getvalue :: Maybe (a,String)}
```

# Exercițiul 1

- Faceți `MaybeWriter` instanță a clasei `Monad`, astfel încât cazurile de eroare să întoarcă numai valoarea `Nothing`, ignorand output-ul acumulat.

```
newtype MaybeWriter a = MW {getvalue :: Maybe (a,String)}

instance  Monad (MaybeWriter ) where
  return x = MW $ Just (x, "")
  ma >>= f =
      case a of
        Nothing -> MW Nothing
        Just (x,w) ->
            case getvalue (f x) of
              Nothing -> MW Nothing
              Just (y,v) -> MW $ Just (y, w++v)
    where a = getvalue ma
```

# Exercițiul 2

- În interpretor modificăm următoarele definiții:

```
type M a = MaybeWriter a

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

Precizați ce modificări trebuie făcute pentru a obține un interpretor cu valori în `MaybeWriter` astfel încât toate cazurile de eroare să întoarcă `Nothing`.

# Exercițiul 2

```haskell
showM :: Show a => M a -> String
showM ma =
    case a of
      Nothing -> "Nothing"
      Just (x,w) ->
          "Output: " ++ w ++ "\nValue: " ++ show x
  where a = getvalue ma
```

# Exercițiul 2

```
get :: Name -> Environment -> M Value
get x env =
    case [v | (y,v) <- env , x == y] of
      (v:_) -> return v
      _     -> MW Nothing

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

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

# Exercițiul 2

```haskell
tellMW :: String -> MaybeWriter ()
tellMW  ceva = MW $ Just ( ( ) , ceva)


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

# Exemplu

```
pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Out (Con 10)) :+:  (Out (Con 11)))
> test pgm
"Output: 10; 11; \nValue: 42"


pgmW = App (Lam "y" (Out (Con 3))) (Var "y")

> test pgmW
"Nothing"
```

# Exercițiul 3

Modificăm tipul de date `Term` prin adăugarea operației `:/:` care va fi interpretată ca `div`.

```
data Term = ... | Term :/: Term
```

În modulul definit la Exercițiul 2, în care interpretarea termenilor se face în monada `MaybeWriter Value`, completati definiția funcției `interp` adăugând semantica operației `:/:`, considerând ca eroare împărțirea la `0`.

# Exercițiul 3

```
data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Term :/: Term
          | Lam Name Term
          | App Term Term
          | Out Term
  deriving (Show)
```

# Exercițiul 3

```
interp (t1 :/: t2) env
  = do
    v1 <- interp t1 env
    v2 <- interp t2 env
    imparte v1 v2

imparte :: Value -> Value -> M Value
imparte (Num i) (Num j)
  | j == 0     = MW Nothing
  | otherwise = return (Num $ i `div` j)
imparte _ _ = MW Nothing
```

# Exemplu

```
pgm2 = App
     (Lam "x" ((Var "x") :+: (Var "x")))
     ((Con 10) :/:  (Out (Con 2)))
```

```
> test pgm2
"Output: 2; \nValue: 10"
```

```
pgmW2 = App
     (Lam "x" ((Var "x") :+: (Var "x")))
     ((Con 10) :/:  (Out (Con 0)))
```

```
> test pgmW2
"Nothing"
```

# Jocul de șah

# Tabla de șah

```
import          Data.List (lookup)
import          Data.Char (chr, ord)
```

Problemele din acest exercițiu se petrec pe o tablă de șah. O poziție pe tabla de șah este reprezentată ca o pereche `(coloana, linie)`, unde `coloana` este o literă mică între 'a' și 'h', iar `linie` este o cifră între `1` și `8`.

Căsuța din stânga-jos a tablei de șah are poziția `('a', 1)` iar cea din dreapta-sus are poziția `('h', 8)`.

```
type Linie = Int
type Coloana = Char
type Pozitie = (Coloana, Linie)
```

# Tabla de șah

```
type Linie = Int
type Coloana = Char
type Pozitie = (Coloana, Linie)
```

O mutare (validă) a unei piese de șah este dată de o pereche `(dcol, dlin)` reprezentând deplasamentul pe coloană, respectiv linie, indus de mutare.

Un deplasament de `x` pe coloană (linie) înseamnă o mutare cu `x` căsuțe spre dreapta (în sus). Un deplasament negativ indică mutarea în direcția opusă, i.e. stânga (jos).

De exemplu, mutarea `(1, -2)` reprezintă mutarea (validă pentru un cal) a unei căsuțe la dreapta și două căsuțe in jos.

```
type DeltaLinie = Int
type DeltaColoana = Int
type Mutare = (DeltaColoana,DeltaLinie)
```

# Exercițiul 1: Efectuarea unei mutări

Implementați o funcție `mutaDacaValid` care dată fiind o poziție `p` și o mutare `m`, întoarce poziția obținută după efectuarea mutării piesei din poziția `p` folosind mutarea descrisă de `m`. Dacă mutarea nu este posibilă, se va întoarce vechea poziție. Exemple:

```
ex1t1, ex1t2, ex1t3 :: Bool
ex1t1 = mutaDacaValid ('e', 5) (1, -2) == ('f', 3)
      -- 'f' este o casuță la dreapta lui 'e',
      -- linia 3 e cu 2 sub 5
ex1t2 = mutaDacaValid ('b', 5) (-2, 1) == ('b', 5)
      -- mutând 2 căsuțe la stânga am ieșit de pe tablă
ex1t3 = mutaDacaValid ('e', 2) (1, -2) == ('e', 2)
      -- mutând 2 căsuțe în jos am ieșit de pe tablă
```

# Exercițiul 1: Efectuarea unei mutări (cont)

```haskell
mutaDacaValid :: Pozitie -> Mutare -> Pozitie
mutaDacaValid (c, l) (dc, dl)
  | c' `intre` ('a', 'h') && l' `intre` (1, 8) = (c', l')
  | otherwise = (c, l)
  where
    l' = l + dl
    c' = toEnum (fromEnum c + dc)
    x `intre` (b, e) = x >= b && x <= e

-- fromEnum 'c' == 99
-- toEnum (fromEnum 'a' + 3) :: Char  == 'd'
```

# Joc și desfășurarea lui

Definiția de mai jos reprezintă lista mutărilor valide pentru un cal. Mutările valide pentru un cal sunt în forma literei L: 2 căsuțe într-o direcție și 1 căsuță într-o direcție perpendiculară.

```
mutariPosibile :: [Mutare]
mutariPosibile = [(-2,-1),(-2,1),(2,-1),(2,1),(-1,-2),(1,
```

În continuare vom reprezenta mai succint o mutare a unui cal ca un indice (de la 0 la 7) în lista de mutări posibile.

```
type IndexMutare = Int
```

Un "joc" este o secvență de sărituri ale calului, dată ca o listă de indici de mutare.

```
type Joc = [IndexMutare]
exJoc :: Joc
exJoc = [0,3,2,7]
```

# Exercițiul 2: simularea unui joc

Desfășurarea unui joc este definită ca lista de poziții indusă de un joc, executând mutările în ordine, pornind de la o poziție inițială.

```
type DesfasurareJoc = [Pozitie]
```

Implementați o funcție `joaca` care pentru o poziție inițială `p` și o secvență de indici `joc` produce desfășurarea jocului corespunzător, adică lista de poziții care sunt atinse începând cu `p` și efectuând în ordine mutările corespunzătoare din `mutariPosibile` descrise de indicii din `joc`.

Dacă indicele nu reprezintă o poziție din `mutariPosibile`, sau dacă mutarea este invalidă (ar ajunge în afara tablei), atunci acesta este ignorat.

# Exercițiul 2: simularea unui joc (cont)

Exemple:

```
ex2t1, ex2t2, ex2t3 :: Bool
ex2t1 = joaca ('e',5) [0,3,2,7]
      == [('e',5),('c',4),('e',5),('g',4),('h',6)]
ex2t2 = joaca ('e',5) [0,3,9,2,7]
      == [('e',5),('c',4),('e',5),('g',4),('h',6)]
 --  9 nu e un index valid in mutariPosibile
ex2t3 = joaca ('a',8) [0,3,2,7]
      == [('a',8),('c',7)]
 --  doar mutarea dată de indicele 2 poate fi efectuată
joaca :: Pozitie -> Joc -> DesfasurareJoc
joaca p [] = [p]
joaca p (i : is) | i < 0 || i > 7 || p' == p = joaca p is
                 | otherwise = p : joaca p' is
  where p' = mutaDacaValid p (mutariPosibile !! i)
```

# Traseul calului pe tabla de șah. Arbore de joc.

Un joc puțin mai interesant este acela în care vrem să ne asigurăm ca nu vizităm aceeași căsuță de mai multe ori. Pentru aceasta, va trebui să ținem minte căsuțele pe care l-am vizitat deja.

Pentru a explora desfășurarea unui joc putem folosi un arbore de joc, în care un nod constă dintr-o poziție și are ca subarbori evoluții posibile ale jocului pornind din acea poziție.

```
data ArboreJoc = Nod Pozitie [ArboreJoc]
  deriving (Show, Eq)
```

# Traseul calului pe tabla de șah. Arbore de joc.(cont)

```haskell
data ArboreJoc = Nod Pozitie [ArboreJoc]
  deriving (Show, Eq)
```

Deoarece arborele de joc va fi foarte mare, vom folosi următoarea funcție pentru a obține arborele doar până la o adâncime dată.

```haskell
parcurge :: Int -> ArboreJoc -> ArboreJoc
parcurge adancime (Nod p as)
  | adancime <= 0 = Nod p []
  | otherwise     = Nod p (map (parcurge (adancime - 1)) a
```

# Monada `Writer` specializată la `Joc`

```haskell
newtype JocWriter a = Writer { runWriter :: (a, Joc) }

instance Monad JocWriter where
  return a = Writer (a, [])
  ma >>= k =    let (x, jocM) = runWriter ma
                    (y, jocK) = runWriter (k x)
                in  Writer (y, jocM ++ jocK)

instance Functor JocWriter where
  fmap f ma = ma >>= return . f

instance Applicative JocWriter where
  pure = return
  mf <*> ma = mf >>= (<$> ma)
```

# Monada `Writer` specializată la `Joc`

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

`scrie` ia ca argument **un singur** indice de mutare și "scrie la ieșire" jocul constând doar din acel indice.

```
scrie :: IndexMutare -> JocWriter ()
scrie i = Writer ((), [i])
```

# Exercitiul 4: Simulare joc cu obținerea mutărilor valide

Cerințele sunt aproximativ aceleași ca la exercițiul 2, cu diferența că

- o mutare care duce spre o poziție deja vizitată devine invalidă
- daca mutarea este validă, indexul ei trebuie scris folosind monada `Writer`

Implementați o funcție `joacaBine` care pentru o poziție inițială `p` și o secvență de indici `joc` produce desfășurarea jocului corespunzător, adică lista de poziții care sunt atinse începând cu `p` și efectuând în ordine mutările corespunzătoare din `mutariPosibile` descrise de indicii din `joc`. Efectul lateral al funcției este calcularea listei indicilor de mutare valizi.

Dacă indicele nu reprezintă o poziție din `mutariPosibile`, sau dacă mutarea este invalidă (ar ajunge în afara tablei **sau pe o poziție deja vizitată**), atunci acesta este ignorat.

# Exercitiul 4: Simulare joc cu obținerea mutărilor valide (cont)

Exemple:

```
ex4t1, ex4t2, ex4t3 :: Bool
ex4t1 = runWriter (joacaBine ('e',5) [0,3,2,7])
      ==  ( [('e',5),('c',4),('e',3),('f',5)], [0,2,7])
 -- mutarea 3 nu mai e valida pentru ca revine la o
 -- pozitie veche
ex4t2 = runWriter (joacaBine ('e',5) [0,3,9,2,7])
      ==  ( [('e',5),('c',4),('e',3),('f',5)], [0,2,7])
 -- indicele 9 e in afara tabelei de mutari valide
ex4t3 = runWriter (joacaBine ('a',8) [0,3,2,7])
      == ([('a',8),('c',7)], [2])
 -- doar mutarea dată de indicele 2 poate fi efectuată
```

# Exercitiul 4: Simulare joc cu obținerea mutărilor valide (cont)

Sugestie: folosiți o funcție auxiliară care ține minte pozițiile deja generate. Puteți folosi funcțiile definite mai sus.

```
joacaBine :: Pozitie -> Joc -> JocWriter DesfasurareJoc
joacaBine p joc = go [] p joc
  where
    go _ p [] = return [p]
    go ps p (i:is)
      | i < 0 || i > 7 || p' `elem` (p:ps) = go ps p is
      | otherwise = do
                      scrie i
                      game <- (go (p:ps) p' is)
                      return (p : game)
      where
        p' = mutaDacaValid p (mutariPosibile !! i)
```

Succes la examen!