Cursul 10 - MONADE

- Monade
  - Introducere
  - Tipul de data Maybe
  - Constructori de tip
  - Monada in Haskell
    - Exemplul 1 monada Maybe
    - Exemplul 2 monada []
  - Clasa Monad
  - Notatia do
  - Legi ale monadei
  - Studiu de caz

Introducere

- Monada – o cale de a structura calculele ca valori si secvente de calcule ce utilizeaza aceste valori
- Permit construirea de calcule folosind blocuri secventiale care, la randul lor pot fi secvente de calcul
- Monada determina modul in care o combinatie de calcule formeaza un nou calcul, scutindu-l pe programator de a scrie cod pentru astfel de combinatii
- O monada este vazuta ca o strategie pentru combinarea calculelor intr-un calcul complex

Tipul de data Maybe

- Tipul "calcul care poate esua sau poate returna o valoare":

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

```
mydiv:: Float -> Float -> Maybe Float
mydiv x 0 = Nothing
mydiv x y = Just (x/y)
```

- Tipul Maybe sugereaza o strategie de combinare a calculelor: daca un calcul compus consta dintr-un calcul B care depinde de rezultatul altui calcul A, atunci calculul combinat va produce `Nothing` daca A sau B produce `Nothing`; altfel, daca ambele se produc cu succes, calculul combinat va produce rezultatul lui B aplicat rezultatului lui A

Proprietati fundamentale

- Modularitate
  - calcule compuse din calcule simple
  - separarea strategiilor de combinare de calcule in sine
- Flexibilitate
  - Programele scrise cu monade sunt mai adaptabile decat cele fara monade
- Izolare
  - monadele permit crearea de structuri de calcul in stil imperativ(sistemul I/O de ex.) care raman izolate de corpul principal al programului functional

Constructori de tip

- Un constructor de tip este o definitie de tip parametrizat ce foloseste tipuri parametrizate
- In definitia tipului Maybe

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

Maybe este un constructor de tip iar **Nothing**, **Just** sunt constructori de date (de tipul Maybe)

- Se poate construi o data aplicand constructorul de data **Just** unei valori:

```
tara = Just "Romania"
```

- Se poate construi un tip aplicand constructorul de tip **Maybe** unui tip:

```
Maybe Int, Maybe String
```

- Tipurile polimorfe sunt ca si containerele ce sunt capabile sa contina valori de diverse tipuri:
  - **`Maybe String`** ar putea fi privit ca un container **`Maybe`** ce poate contine valori de tip **`String`** (sau **`Nothing`**)
- Se poate ca tipul containerului sa fie polimorf: **`m a`** reprezinta un container de un anumit tip ce poate contine valori de un anumit tip
- Se folosesc adesea variabile tip cu constructorii de tip pentru a descrie trasaturile abstracte ale unui calcul: **`Maybe a`** este tipul tuturor calculelor care pot returna o valoare sau **`Nothing`**

Monada in Haskell

- Monada este data prin:
  - un constructor de tip (sa zicem m) care este numit constructorul de tip monada
  - O functie `a -> m a` care construeste valori de acest tip, numita **`return`**
  - O functie `m a -> (a -> m b) -> m b` care combina valori de acest tip cu calcule (functii) ce produc valori de acest tip pentru a produce un nou calcul pentru valori de acest tip. Aceasta functie este denumita **`bind`** si se scrie **`>>=`**

Monada in Haskell

- constructorul de tip monada defineste un tip de calcul
- functia **`return`** creaza valori primitive ale acestui tip
- operatorul (functia) >>= combina calcule de acest tip pentru a obtine calcule mai complexe de tipul respectiv
- Analogia cu containerul:
  - constructorul de tip m este un container ce poate contine diferite valori
  - m a este un container ce contine valori de tip a
  - functia **`return`** pune o valoare in container
  - functia >>= ia valoarea din container si o transmite unei functii pentru a produce un container ce contine o noua valoare, posibil de alt tip
  - functia >>= se cheama **`bind`** pentru ca ea leaga valoarea din containerul monada cu primul argument al functiei

Exemplu: Monada **Maybe**

- Studiu de caz: determinarea stramosilor: bunici, strabunici etc. din arborele genealogic
- Tipul de data Person:

```
data Person= Person {name::String,
                     mother::Maybe Person,
                     father::Maybe Person}
```

- Pentru a determina bunicul unei persoane am putea defini o functie:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather s = case (mother s) of
                            Nothing -> Nothing
                            Just m -> father m
```

Exemplu: Monada **Maybe**

- Un alt stramos:

```
mothersPaternalGrandfather :: Person -> Maybe Person
mothersPaternalGrandfather s = case (mother s) of
                                 Nothing -> Nothing
                                 Just m -> case (father m) of
                                             Nothing -> Nothing
                                             Just gf -> father gf
```

- Solutia nu este eficienta
- Odata gasita valoarea `Nothing` intr-un punct al calculului, aceasta ramane `Nothing` ca rezultat final
- Sa incercam sa implementam acest lucru o singura data

Exemplu: Monada **Maybe**

- Crearea unui combinator ce inglobeaza acest aspect:

```
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing _ = Nothing
comb (Just x) f = f x
```

- Utilizarea acestui combinator pentru a construi secvente mai complicate:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather s = (Just s) `comb` mother `comb` father

fathersMaternalGrandmother :: Person -> Maybe Person
fathersMaternalGrandmother s = (Just s) `comb` father `comb`
                               mother `comb` mother

mothersPaternalGrandfather :: Person -> Maybe Person
mothersPaternalGrandfather s = (Just s) `comb` mother `comb`
                               father `comb` father
```

Exemplu: Monada **Maybe**

- Combinatorul **comb**
  - Este o functie polimorfa, nu este specializata pentru tipul Person
  - Captureaza strategia generala de combinare a calculelor care pot, in particular, sa esueze
  - Poate fi folosit si in alte aplicatii: interogare baze de date, cautare in dictionare, etc.

- Tipul **Maybe** impreuna cu functia **Just** (care actioneaza ca **return**) si combinatorul **comb** (ce actioneaza ca >>=) formeaza o monada folosita pentru construirea de calcule ce pot esua

Monada []

- Constructorul de tip `[]` (pentru construirea listelor) este de asemenea o monada:
  - Functia return creaza lista singleton

```
return x = [x]
```

  - Operatia de legare pentru liste creaza o noua lista continand rezultatul aplicarii functiei tuturor valorilor listei originale

```
l >>= f = concat map f l
```

- Monada Lista inglobeaza o strategie de executie simultana a calculelor posibile intr-un calcul ambiguu (drumurile posibile se reprezinta ca si liste)

Clasa **Monad**

- In Haskell exista o clasa standard **Monad** care defineste numele si signatura functiilor **return** si **>>=**

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

- **Hugs> :info Monad**
- Se recomanda ca monadele utilizator sa fie instante ale clasei **Monad**

Exemplul 1 revizuit

- Monada Maybe este instanta a clasei Monad:

```
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return = Just
```

- Se pot utiliza operatorii standard din Monad:

```
maternalGrandfather s = (return s) >>= mother >>= father

fathersMaternalGrandmother s = (return s) >>= father >>=
    mother >>= mother
```

Notatia do

- Alternativa in a utiliza functiile monadice
- Similara cu utilizarea "list comprehensions" pentru liste
- Scrierea calculeor monadice in stil pseudo-imperativ folosind variabile:
  - Rezultatul unui calcul monadic poate fi "asignat" unei variabile folosind operatorul <-
  - Utilizarea variabilei intr-o subsecventa de calcul monadic, se face automat legarea
  - Tipul expresiei din dreapta lui semnului <- este tip monadic m a

Notatia do

```
mothersPaternalGrandfather s = do m <- mother s
                                  gf <- father m
                                  father gf
```

sau:

```
mothersPaternalGrandfather s = do {m <- mother s;
   gf <- father m; father gf}
```

- Monadele ofera prin notatia do posibilitatea de a crea calcule in stil imperativ in cadrul programelor functionale
- Notatia do este doar "syntactic sugar"

Transformarea do in >>=

- Comanda **x <- expr1** devine:

  expr1 >>= \x ->

- Comanda **expr2** devine:

  expr2 >>= _ ->

- De exemplu:

```
mothersPaternalGrandfather s = do  m <- mother s
                                   gf <- father m
                                   father gf
```

devine

```
mothersPaternalGrandfather s = mother s >>= \m ->
                               father m >>= \gf ->
                               father gf
```

- Operatorul >>= (bind) leaga valoarea din monada cu argumentul din urmatoarea lambda expresie

Legi ale monadei

- Operatiile unei monade trebuie sa indeplineasca trei legi fundamentale (conceptul provine din teoria categoriilor):
  - **`return`** este element neutru la stanga pentru **`>>=`**

    ```
    (return x) >>= f == f x
    ```

  - **`return`** este element neutru la dreapta:

    ```
    m >>= return == m
    ```

  - operatia **`>>=`** este asociativa:

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

- Orice constructor de tip impreuna cu **`return`** si **`>>=`** ce indeplineste aceste legi este o monada; programatorul are sarcina sa asigure acest lucru la definirea unei noi monade
- Clasa Monad mai are definite doua functii: **`fail`** si **`>>`**

fail

- Are implementarea standard:

```
fail :: String -> a b
fail s = error s
```

- In monada **Maybe**:

```
fail _ = Nothing
```

- Functia **fail** este apelata atunci cand apare un esec in incercarea de potrivire a paternurilor. De exemplu in functia:

```
f :: Int -> maybe [Int]
f i = do let l = [Just [1,2,3], Nothing, Just [], Just [2..7])
         (x:xs) <- l!!i
         return xs
```

Functia >>

- Leaga doua calcule monadice independente (nu intereseaza rezultatul primului calcul)
- Se poate defini in functie de **>>=**:

```
(>>)  :: m a -> m b -> m b
m >> k = m >>= (\_ -> k)
```

- Exemplu:

  IO a este instanta a clasei **Monad**:

```
Hugs> putChar 'O' >> putStr " echipa" >> putStr " mare!"
O echipa mare!
Hugs> getChar >> getChar>> getChar
abc
```

Studiu de caz: evaluator monadic pentru impartire

- Expresii cu operatorul de împărțire:

```
data Term = Con Int | Div Term Term
                    deriving Show

e1, e2 :: Term
e1 = Div(Div(Con 1972)(Con 2))(Con 23)
e2 = Div(Con 2)(Div(Con 1)(Con 0))
```

Evaluator monadic pentru împărțire

- Evaluatorul eval: dacă m este o monadă atunci eval primeşte la intrare un term si realizează un calcul ce conduce la un întreg astfel:
  - evaluarea lui Con x înseamnă return x
  - evaluarea lui Div t u înseamnă:
    - evaluarea lui t şi legarea lui x cu valoarea lui t
    - evaluarea lui u şi legarea lui y cu valoarea lui u
    - return x `div` y

```
eval :: Monad m => Term -> m Int
eval(Con x) = return x
eval(Div t u) = do x <-eval t
                   y <-eval u
                   return (x `div` y)
```

Evaluare fara semnalare exceptii

- Evaluatorul evalId: specializarea lui eval pentru m = Id
- Monada Id este declarată pentru tipul Id a care este izomorf cu a:
  - return este izomorm cu funcția identitate
  - >>=este izomorf cu funcția aplicație

```
newtype Id a = MkId a
instance Monad Id where
                return x = MkId x
                (MkId x) >>= q = q x
```

- Pentru a specifica modul de afişare:

```
instance Show a => Show(Id a) where
show(MkId x) = "valoare exp: " ++ show x
```

- Evaluatorul de bază:

```
evalId :: Term -> Id Int
evalId = eval
```

- Exemple:

```
Main> evalId e1
valoare exp: 42
Main> evalId e2
valoare exp:
Program error: divide by zero
```

Evaluare cu semnalarea excepțiilor

- Tipul Exc a al excepțiilor pentru tipul a:

```
data Exc a = Raise Exception | Return a
type Exception = String
```

- Monada Exc:

```
instance Monad Exc where
      return x = Return x
      (Raise e) >>= q = Raise e
      (Return x) >>= q = q x
      raise:: Exception -> Exc a
      raise e = Raise e
```

Evaluare cu semnalarea excepțiilor

```
evalEx :: Term -> Exc Int
evalEx(Con x) = return x
evalEx(Div t u) = do   x <- evalEx t
                       y <- evalEx u
                       if y == 0
                              then raise "impartire prin zero\n"
                              else return (x `div` y)

instance Show a => Show(Exc a) where
show(Raise e) = "exceptie: " ++ e
show(Return x) = "valoare exp: " ++ show x

Main> evalEx e1
valoare exp: 42
Main> evalEx e2
exceptie: impartire prin zero
```