Programare declarativă - Implementarea unui joc

Traian Șerbănuță (33) Ioana Leuștean (34)

Departamentul de Informatică, FMI, UNIBUC
traian.serbanuta@fmi.unibuc.ro, ioana@fmi.unibuc.ro

Jocul Nim

În acest curs vom implementa in Haskell jocul Nim:

Joc mai mulți jucători care mută pe rând

Tabla: mai multe gramezi cu piese

Mutare: jucatorul ia una sau mai multe piese dintr-o gramadă (poate lua toată grămada, dar piesele trebuie luate din aceeași grămadă)

Regula: câștigă jucătorul care a făcut ultima mutare

Jocul Nim - Exemplu

```
> game
Board: [2,5,4]

Possible run of the Nim game:

Alice ia 4 obiecte din gramada 2
Bob ia 1 obiecte din gramada 3
Alice ia 2 obiecte din gramada 1
Bob ia 1 obiecte din gramada 1
Alice ia 3 obiecte din gramada 1
```

Jocul Nim - reprezentarea datelor

```haskell
import Data.Word
import Data.Maybe (fromJust)

type HeapName = Int  -- numarul gramezii (incepe de la 1)
type HeapVal = Word32      -- numarul de elemente
type Board = [ HeapVal ]   -- lista gramezilor
type Turn =  (HeapName, HeapVal)
-- ^ din gramada x iau y elemente
type Player = String -- jucatorul care a facut mutarea
type Game = [(Player, Turn)]
```

Initial starea jocului este formata numai din lista gramezilor

```haskell
type GameState = Board
```

In definirea jocului vom folosi monada `State GameState`

Jocul Nim - exerciții pregătitoare

- scrieți o funcție care face o mutare pe o tablă

```
oneTurn :: Turn -> Board -> Maybe Board
oneTurn (n, v) b
  | n > length b = Nothing
  | otherwise =
      let (l1, v':l2) = splitAt (n - 1) b
      in  if v > v'
          then Nothing
          else Just (l1 ++ (v' - v):l2)
```

```
> oneTurn (2,3) [1,4,5]
Just [1,1,5]
```

Jocul Nim - exerciții pregătitoare

- reprezentarea jucătorilor și a ordinii în care mută

```
next :: Player -> Player
next "Bob" = "Alice"
next "Alice" = "Bob"
```

Presupunem că avem numai doi jucători

Jocul Nim - exerciții pregătitoare

- generați toate mutările posibile pentru o tablă dată

```
allTurns :: Board -> [Turn]
allTurns xs = [(i,x)| (i,z) <- (zip [1..] xs), x<-[1..z]]
```

```
 > allTurns [2,3]
 [(1,1),(1,2),(2,1),(2,2),(2,3)]
```

```
nextTurns :: Board -> [Board]
nextTurns b = [b' | Just b' <- map (`oneTurn` b) (allTurns b)]
```

```
 > nextTurns [2,3]
 [[1,3],[0,3],[2,2],[2,1],[2,0]]
```

Jocul Nim - arborele jocului

- construim arborele jocului pentru o tablă dată astfel:
- nodurile arborelui sunt table
- fiii unui nod sunt tablele la care se ajunge făcând o singură mutare
- frunzele sunt table care nu au fiii (în frunze vor fi liste care conțin numai 0)

```
data GameTree = V Board [GameTree]
  deriving Show
```

Jocul Nim - arborele jocului

```
gameTree :: Board -> GameTree
gameTree b = V b subTrees
  where
    subTrees = [gameTree b' | b' <- nextTurns b]
```

```
> gameTree [2,1]
V [2,1] [V [1,1] [V [0,1] [V [0,0] []],
                  V [1,0] [V [0,0] []]],
         V [0,1] [V [0,0] []],
         V [2,0] [V [1,0] [V [0,0] []],V [0,0] []]]
```

Jocul Nim - variante câștigătoare

- pentru fiecare jucător calculați numărul de variante câștigătoare
  - cu 1 numarăm `null` pe nivele pare 0, 2, ... din arborele jocului
  - cu 0 numarăm `null` pe nivelele impare 1,3, ... din arborele jocului

```
wins :: Int -> GameTree -> Int
wins 1 (V _ [])  = 1  -- frunza se numara
wins 0 (V _ [])  = 0  -- frunza nu se numara
wins 0 (V _ xs) = sum (map (wins 1) xs)
wins 1 (V _ xs) = sum (map (wins 0) xs)

-- de cate ori castiga primul jucator
firstWins, secondWins :: Board -> Int
firstWins board = wins 1  (gameTree board)
secondWins board  = wins 0 (gameTree board)
```

```
> firstWins [2,5,6]
518388
```

Jocul Nim - structura generală

```
gameBoard =  [ 2, 5, 4]

game =  do
         putStrLn ("Board: " ++ (show gameBoard))
         putStrLn $ showGame. playGame $ gameBoard

type Game = [(Player, Turn)]
type GameState = Board

playGame :: Board  -> State GameState Game

showGame :: State GameState Game -> String
```

Monada State

- Vom folosi monada State pentru a menține starea jocului

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

instance Monad (State state) where
  return a = State (\ s -> (a, s))
  ma >>= k = State g
    where g state = let (a, aState) = runState ma state
                    in runState (k a) aState

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

instance Functor (State state) where
  fmap f ma = pure f <*> ma
```

Monada State - funcții ajutătoare

- Vom folosi monada State pentru a menține starea jocului

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

get :: State state state
get = State (\s -> (s, s))  -- întoarce starea curentă

put :: s -> State s ()
put s = State (\_ ->((), s)) -- schimba starea curentă

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

Jocul Nim - showGame

- afișează șirul de mutări

```
showGame :: State GameState Game -> String
showGame g =  showG a
  where (a, _) = runState g []

showG :: Game -> String
showG [] = ""
showG ((w, (x ,v)) : ws) =
    w ++ " ia " ++ (show v)
    ++ " obiecte din gramada " ++ (show x)  ++ "\n"
    ++ (showG ws)
```

Jocul Nim - playGame

- în această variantă fiecare jucător are o strategie

```
playGame :: Board  -> State GameState Game
playGame initb
  = do     put  initb -- seteaza starea initiala
           loop "Alice"   -- Alice face prima mutare
  where
    loop w = do
      turn <- playWithStrat (strategy w)
      -- executa mutarea data de strategie
      -- si o intoarce pentru oprire/afisare
      if turn == Nothing
        then return []
        else do
            turnl <- loop (next w)
            -- trece la urmatorul jucator
            return $ [(w, fromJust turn)] ++ turnl
```

Jocul Nim - playWithStrat

```
strategy :: Player -> (Board -> Turn)

playWithStrat :: (Board -> Turn) -> State GameState (Maybe Tur
playWithStrat strat
  = do
    currentBoard <- get
    if all (==0) currentBoard
    then return Nothing -- jocul s-a încheiat
    else do
        let         -- mutarea este data de strategie
          turn = strat currentBoard
          Just newBoard = oneTurn turn currentBoard
        put newBoard
        return (Just turn)
```

Starea jocului este menținută de monada State

Jocul Nim - strategii

Mutarea este determinata de o strategie

```
strategy :: Player -> (Board -> Turn)
strategy "Alice" = strat1
strategy "Bob"   = strat2
```

Strategiile sunt foarte simple

```
strat1 xs =  (head pozitii, 1)
  where pozitii = [p | (p, y) <- zip [1..] xs, y > 0]

strat2 xs =  (last pozitii, 1)
  where pozitii = [p | (p, y) <- zip [1..] xs, y > 0]
```

Exemplu de rulare cu strat1 si strat2

```
> game
Board: [2,5,4]
Alice ia 1 obiecte din gramada 1
Bob ia 1 obiecte din gramada 3
Alice ia 1 obiecte din gramada 1
Bob ia 1 obiecte din gramada 3
Alice ia 1 obiecte din gramada 2
Bob ia 1 obiecte din gramada 3
Alice ia 1 obiecte din gramada 2
Bob ia 1 obiecte din gramada 3
Alice ia 1 obiecte din gramada 2
Bob ia 1 obiecte din gramada 2
Alice ia 1 obiecte din gramada 2
```

Jocul Nim - cu mutari generate random

- În loc să folosim strategii, vom genera random mutările;
- În acest caz, starea jocului va trebui să actualizeze și generatorul de numere aleatoare

```
type Seed = Word32
data RandomGameState = GS {seed :: Seed,  board :: Board}
  deriving Show
type StateRandomGame = State RandomGameState

randomGame :: IO ()
randomGame
  = do putStrLn ("Board: " ++ (show gameBoard))
       putStrLn "Seed = "
       initialSeed <- readLn
       putStrLn "Possible run of the Nim game:"
       putStrLn . showRandomGame . playRandomGame
           $ GS initialSeed gameBoard -- starea initiala
```

Jocul Nim - cu mutari generate random

```
type Seed = Word32
data RandomGameState = GS {seed :: Seed,  board :: Board}
type StateRandomGame = State RandomGameState

showRandomGame :: StateRandomGame Game -> String
showRandomGame g =  showG a
  where (a, s) = runState g (GS 0 [])
```

Jocul Nim - cu mutari generate random

```haskell
data GameState = GS {seed :: Seed,  board :: Board}

playRandomGame :: RandomGameState -> StateRandomGame Game
playRandomGame initstate
  = do
    put  initstate  -- starea initiala
    loop "Alice"
  where
    loop w = do
        --  mutarea efectuata e generata aleator
        turn <- playRandom
        if (turn == Nothing)
        then return []
        else do
            turnl <- loop (next w)
            return $ [(w, fromJust turn)] ++ turnl
```

Jocul Nim - cu mutari generate random

- în funcția `playRandom` trebuie să generăm aleator:
  - indicele unei grămezi nenule
  - valoarea pe care o extragem din acea grămadă

```
playRandom ::  StateRandomGame (Maybe Turn)
playRandom
  = do
    cBoard <- board <$> get
    let availHeaps = [(p,h) | (p,h) <- zip [1..] cBoard, h>0]
    if null availHeaps
    then return Nothing -- jocul s-a încheiat
    else do -- mutarea este aleasa aleator
        turn <- randomTurn availHeaps
        let Just newBoard = oneTurn turn cBoard
        modify (\s -> s { board = newBoard })
        return (Just turn)
```

Jocul Nim - cu mutari generate random

```
randomTurn :: [Turn] -> StateRandomGame Turn
randomTurn availableHeaps
  = do
    rnd1 <- random
    let index = fromIntegral rnd1 `mod` length availableHeaps
        (idxHeap, valHeap) = availableHeaps !! index
    rnd2 <- random
    let removeHeap = 1 + fromIntegral rnd2 `mod` valHeap
    return (idxHeap, removeHeap)
  where
    cMULTIPLIER = 1664525 ; cINCREMENT = 1013904223
    random = do
        currentSeed <- seed <$> get
        let nextSeed = cMULTIPLIER * currentSeed + cINCREMENT
        modify (\s -> s { seed = nextSeed })
        return nextSeed
```

Exemplu de rulare cu mutari random

```
> randomGame
Board: [2,5,4]
Seed =
567
Possible run of the Nim game:
Alice ia 4 obiecte din gramada 2
Bob ia 2 obiecte din gramada 3
Alice ia 1 obiecte din gramada 3
Bob ia 1 obiecte din gramada 3
Alice ia 1 obiecte din gramada 1
Bob ia 1 obiecte din gramada 2
Alice ia 1 obiecte din gramada 1
```

Pe săptămâna viitoare!