Reexaminare Programare declarativă, 4 iunie 2018

1. Scrieți o funcție `f :: [(a,a)] -> [a]` care primește ca argument o listă de perechi de elemente de același tip și calculează o listă conținând primul element din fiecare pereche de pe o poziție pară și al doilea element din fiecare pereche de pe o poziție impară, unde numerotarea pozițiilor începe de la 1.

   De exemplu:

   ```
   f [(0,3),(1,2),(5,7),(3,8),(4,9)] = [3,1,7,3,9]
   f [('a','b')] = ['b']
   f [] = []
   ```

   1. (1p) Folosind doar funcții de bază, descrieri de liste și funcții din biblioteci, dar fără recursie.
   2. (1p) Folosind doar funcții de bază și recursie, dar fără descrieri de liste sau alte funcții din biblioteci.

2. Scrieți o funcție `p :: [Int] -> Int` care calculează produsul rezultatelor înmulțirii fiecărui număr pozitiv impar din lista dată ca argument cu 3.

   Dacă lista este vidă sau nu conține numere pozitive impare, rezultatul funcției va fi 1.

   De exemplu:

   ```
   p [1,6,-15,11,-9] = 3*1 * 3*11 = 99
   p [3,6,9,12,-9,9] = 3*3 * 3*9 * 3*9 = 6561
   p [] = 1
   p [-1,4,-15] = 1
   ```

   1. (1p) Folosind doar funcții de bază, descrieri de liste și funcții din biblioteci, dar fără recursie.
   2. (1p) Folosind doar funcții de bază și recursie, dar fără descrieri de liste sau alte funcții din biblioteci.
   3. (1p) Folosind următoarele funcții de ordin înalt (fără recursie sau descrieri de liste):

      ```
      map :: (a -> b) -> [a] -> [b]
      filter :: (a -> Bool) -> [a] -> [a]
      foldr :: (a -> b -> b) -> b -> [a] -> b
      ```

Funcții de bază

```
div, mod :: Integral a => a -> a -> a
(<), (<=), (>), (>=) :: Ord => a -> a -> Bool
(&&), (||) :: Bool -> Bool -> Bool
max, min :: Ord a => a -> a -> a
toLower, toUpper :: Char -> Char

(+), (*), (-), (/) :: Num a => a -> a -> a
(==), (/=) :: Eq a => a -> a -> Bool
not :: Bool -> Bool
isAlpha, isLower, isUpper, isDigit :: Char -> Bool
```

Funcții din biblioteci

```
sum, product :: (Num a) => [a] -> a
sum [1.0,2.0,3.0] = 6.0
product [1,2,3,4] = 24

and, or :: [Bool] -> Bool
and [True,False,True] = False
or [True,False,True] = True

maximum, minimum :: (Ord a) => [a] -> a
maximum [3,1,4,2] = 4
minimum [3,1,4,2] = 1

concat :: [[a]] -> [a]
concat ["go","od","bye"] = "goodbye"

(++) :: [a] -> [a] -> [a]
"good" ++ "bye" = "goodbye"

reverse :: [a] -> [a]
reverse "goodbye" = "eybdoog"

head :: [a] -> a
head "goodbye" = 'g'

tail :: [a] -> [a]
tail "goodbye" = "oodbye"

init :: [a] -> [a]
init "goodbye" = "goodby"

last :: [a] -> a
last "goodbye" = 'e'

take :: Int -> [a] -> [a]
take 4 "goodbye" = "good"

drop :: Int -> [a] -> [a]
drop 4 "goodbye" = "bye"

zip :: [a] -> [b] -> [(a,b)]
zip [1,2,3,4] [1,4,9] = [(1,1),(2,4),(3,9)]

takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile isLower "goodBye" = "good"

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile isLower "goodBye" = "Bye"

(!!) :: [a] -> Int -> a
[9,7,5] !! 1 = 7

length :: [a] -> Int
length [9,7,5] = 3

elem :: (Eq a) => a -> [a] -> Bool
elem 'd' "goodbye" = True

replicate :: Int -> a -> [a]
replicate 5 '*' = "*****"
```

3. Acest subiect descrie un limbaj simplu de comanda a unui robot, conținând instrucțiuni pentru mișcarea robotului înainte și înapoi pe o linie infinită.

Următoarele declarații descriu comenzile posibile:

```
data Move
    = Go Int        -- avansează în direcția curentă cu valoarea dată
    | Turn          -- inversează direcția
    | Signal        -- trimite un semnal, fără a schimba direcția

data Command
    = Nil                       -- comanda vidă - nu face nimic
    | Command :#: Move          -- comandă urmată de o mișcare
```

Înainte și după fiecare comandă, robotul se află într-o stare, care constă din poziția la care se află pe linie și orientarea (dreapta / stânga)

```
type Position = Int

data Direction = L | R

type State = (Position, Direction)
```

1. (1p) Scrieți o funcție `state :: Move -> State -> State` care, dată fiind o mișcare și stare curentă a robotului, calculează starea robotului în urma acelei mișcări.
   De exemplu:

   ```
   state (Go 3) (0,R) = (3,R)
   state (Go 3) (0,L) = (-3,L)
   state Turn (-2,L) = (-2,R)
   state Signal (4,R) = (4,R)
   ```

2. (2p) Atunci când un robot se mișcă conform direcțiilor dintr-o comandă, el trece printr-o secvență de stări, începând cu starea sa originală și terminând cu starea sa finală.
   Scrieți o funcție `trace :: Command -> State -> [State]` care calculează secvența de stări. De exemplu:

   ```
   trace (Nil) (3,R) = [(3,R)]
   trace (Nil :#: Go 3 :#: Turn :#: Go 4) (0,L)
   = [(0,L),(-3,L),(-3,R),(1,R)]
   trace (Nil :#: Go 3 :#: Signal :#: Turn :#: Turn) (0,R)
   = [(0,R),(3,R),(3,R),(3,L),(3,R)]
   trace (Nil :#: Go 3 :#: Turn :#: Go 2 :#: Go 1 :#: Turn :#: Go 4) (4,L)
   = [(4,L),(1,L),(1,R),(3,R),(4,R),(4,L),(0,L)]
   ```

3. (1p) Se consideră monada `Stare` descrisă mai jos:

   ```
   -- Încapsulează stare de tip 's' și calculează o valoare de tip 'a'
   newtype Stare s a = Stare { runState :: s -> (a,s) }

   instance Monad (Stare s) where
       return x = Stare $ \s -> (x,s)
       (Stare h) >>= f = Stare $ \s -> let (a, newState) = h s
                                           (Stare g) = f a
                                       in  g newState


   get :: Stare s s              -- acțiune care întoarce starea curentă
   get = Stare $ \s -> (s,s)
   put :: s -> Stare s ()        -- comandă care setează starea curentă
   put newState = Stare $ \s -> ((),newState)
   ```

   Rescrieți funcția `state` definită la punctul (3.1)în monada `Stare State`

   ```
   monadState :: Move -> Stare State ()
   ```

   Indicație: puteți folosi funcțiile `get` și `put` definite mai sus, precum și funcția `state` de la punctul (3.1).