Programare Declarativa: test de laborator

Introducere

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

Problemele din acest test 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)
```

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

-- Exercitiul 1
mutaDacaValid :: Pozitie -> Mutare -> Pozitie
mutaDacaValid = undefined
```

Mutări posibile; mutări indexate. 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,-2),(-1,2),(1,2)]
```

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]
```

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]
```

Exercițiul 2 (2 puncte): simularea unui joc

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.

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)]
     -- deoarece 9 nu e un index valid in mutariPosibile
ex2t3 = joaca ('a',8) [0,3,2,7] == [('a',8),('c',7)]
     -- deoarece doar mutarea dată de indicele 2 poate fi efectuată

-- Exercitiul 2
joaca :: Pozitie -> Joc -> DesfasurareJoc
joaca = undefined
```

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)
```

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

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

Exercitiul 3 (2 puncte): Generarea unui arbore de joc

Implementați o funcție **`genereazã`** care dată fiind o poziție generează arborele de joc corespunzător traseelor unui cal care pleacă din poziția dată și face sărituri pe tabla de șah, fără a ieși în afara ei, și fără a ajunge pe aceeași poziție de 2 ori (pe orice drum de la rădăcină spre frunze, nodurile au poziții distincte).

Exemple:

```
ex3t1, ex3t2, ex3t3 :: Bool
ex3t1 = -- generez arborele de joc pentru ('e',5) pana la adâncimea 1
    parcurge 1 (genereaza ('e', 5))
    == Nod ('e',5) -- poziția inițială
          [ Nod ('c',4) []
          , Nod ('c',6) []
          , Nod ('g',4) []
          , Nod ('g',6) []
          , Nod ('d',3) []
          , Nod ('f',3) []
          , Nod ('d',7) []
          , Nod ('f',7) []
          ] -- cele 8 poziții la care pot ajunge într-o mutare
ex3t2 = -- generez arborele de joc pentru ('a',1) pana la adâncimea 1
    parcurge 1 (genereaza ('a', 1))
    == Nod ('a',1) [Nod ('c',2) [],Nod ('b',3) []]
ex3t3 = -- generez arborele de joc pentru ('a',1) pana la adâncimea 2
    parcurge 2 (genereaza ('a', 1))
    == Nod ('a',1)
          [ Nod ('c',2)   -- nivelul I
              [ Nod ('a',3) [] -- nivelul II cu nivelul III vid
              , Nod ('e',1) []
              , Nod ('e',3) []
              , Nod ('b',4) []
              , Nod ('d',4) []
              ]
```

```
          , Nod ('b',3)   -- nivelul I
              [ Nod ('d',2) []
              , Nod ('d',4) [] -- d4 apare din nou, dar pe altă cale, e OK
              , Nod ('c',1) []
              , Nod ('a',5) []
              , Nod ('c',5) []
              ]
          ]
```

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

Atenție! Nu încercați să afișați întreg arborele de joc. Folosiți funcția `parcurge` pentru a restricționa numărul de nivele.

```
-- Exercitiul 3
genereaza :: Pozitie -> ArboreJoc
genereaza = undefined
```

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])

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)
```

Exercitiul 4 (1 punct): Simulare joc cu obținerea mutărilor valide

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

1. La fel ca la exercițiul 3, o mutare care duce spre o poziție deja vizitată devine invalidă
2. 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.

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])
     -- deoarece doar mutarea dată de indicele 2 poate fi efectuată
```

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

```haskell
-- Exercitiul 4
joacaBine :: Pozitie -> Joc -> JocWriter DesfasurareJoc
joacaBine = undefined
```

Exercițiu suplimentar (0,5p): Identifică mutarea

Date fiind două poziții `p1` și `p2`, să se determine dacă un cal poate fi mutat de la `p1` la `p2` printr-o singură mutare, și in caz pozitiv să se afle indicele corespunzător mutarii în tabela `mutariPosibile`.

Exemple:

```haskell
exst1, exst2, exst3 :: Bool
exst1 = gasesteMutare  ('e', 5) ('f', 3) == Just 5
exst2 = gasesteMutare  ('a', 8) ('c', 7) == Just 2
exst3 = gasesteMutare  ('e', 5) ('f', 6) == Nothing

-- Exercitiul suplimentar
gasesteMutare :: Pozitie -> Pozitie -> Maybe IndexMutare
gasesteMutare = undefined
```