Examen Programare Declarativă

Nr 2

Pentru rezolvarea cerințelor folosiți fișierul `nr2.hs` care conține unele definiții de funcții, teste, constante cu care puteti testa funcțiile.

În cele ce urmează vom implementa exerciții inspirate din jocul Țintar (Moară). Pentru a rezolva exercițiile nu trebuie să știți cum se joacă acest joc. Jocul presupune în primul rând așezarea unor piese pe o tablă ca în imaginea alaturată.

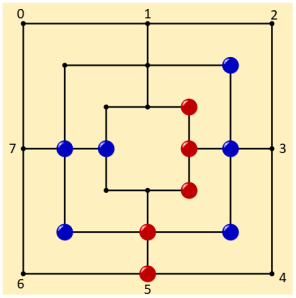

Figure 1: Tabla

Sunt doi jucători. Unul joacă cu piese albe, iar celălalt cu piese negre. Vom considera următorul tip de date pentru a reprezenta piesele jucătorilor.

```
data Piesa = One   -- primul jucător
           | Two   -- al doilea jucător
           | Empty -- căsuță liberă pe tablă
   deriving (Show, Eq)
```

Tabla de joc va fi reprezentată printr-o listă de careuri. Fiecare careu va conține o listă de piese. Primul careu este cel exterior, urmând în ordine, spre interior. Parcurgerea pieselor se va face în sensul acelor de ceas, ca în imaginea de mai sus.

```
data Careu = C [Piesa]
   deriving (Show,Eq)
type Tabla = [Careu]
   deriving (Show,Eq)
```

Exercițiul 1: (2p) Să se verifice dacă o tablă este validă. O tablă este validă dacă conține fix 3 careuri de lungime 8 si fiecare jucător a pus pe tablă maxim 9 piese.

```
validTabla :: Tabla -> Bool
validTabla  = undefined
```

Exercițiul 2: (2p)

O poziție pe tabla de joc va fi reprezentată prin doi indici: numărul careului pe care este așezată și poziția în lista respectivă.

```
data Pozitie = Poz (Int,Int)
```

Pentru o poziție `Poz (x,y)` - x reprezintă indicele careului în tablă, iar y reprezintă poziția în lista corespunzătoare.

O poziție este validă dacă `0 <= x <= 2, 0 <= y <= 7`.

Atenție! În toate exercițiile vom presupune că pozițiile sunt valide: primul indice este un număr între 0 și 2, al doilea indice este un număr între 0 și 7.

Două poziții sunt conectate direct dacă, în desenul de mai sus, sunt unite de o linie care nu mai trece prin alte puncte intermediare. De exemplu, `P(0,1)` este conectată direct cu `P(0,0)`, `P(0,2)` și `P(1,1)`. O mutare va avea loc pornind dintr-o poziție inițială, mutând piesa într-o căsuță goală de pe tablă ce este conectată direct cu poziția inițială.

Să se scrie o funcție care are ca parametrii o tablă și două poziții valide și determină tabla rezultată în urma mutării piesei de pe prima poziție în a doua poziție. Dacă mutarea nu este validă (a doua poziție nu este conectată direct cu prima poziție sau nu este goală sau pe prima poziție nu este nicio piesă) rezultatul funcției va fi `Nothing`.

```
move :: Tabla -> Pozitie -> Pozitie -> Maybe Tabla
move = undefined
```

Definiții pentru Exercițiul 3 și Exercițiul 4

Se dă următorul tip de date folosit pentru a reprezenta jocul.

```
data EitherWriter a = EW {getValue :: Either String (a, String)}
```

Reamintim că definiția tipului de date `Either a b` este:

```
data Either a b = Left a | Right b
```

Componenta `Left` reprezintă cazul de eroare - de exemplu, o mutare nu este validă (în acest caz nu se mai propagă efectele laterale), iar `Right` reprezintă cazul în care jocul se desfășoară fără probleme și acumulează mesajele corespunzătoare fiecărei mutări.

Exercițiul 3 (1p)

Faceți `EitherWriter` instanță a clasei `Monad`.

Exercițiul 4: (2p)

În cadrul acestui exercițiu veți simula o partidă de joc, mutarea pieselor pe tabla dar **fără** a implementa conceptul de moară sau eliminarea pieselor adversarului.

În rezolvarea exercițiului se va folosi monada `EitherWriter`.

Pentru fiecare jucător se dă câte o listă de perechi de poziții pe tabla de joc. Alternativ, fiecare jucător va executa mutarea dată de perechea curentă din listă. Dacă mutarea respectivă nu este validă, jocul se va opri, iar funcția `playGame` va întoarce `EW $ Left error`, unde `error` este un mesaj ce precizează eroarea. Dacă toate mutările sunt valide, funcția `playGame` va întoarce `EW $ Right (table,game)`, unde `table` este tabla finală, iar `game` este un mesaj care descrie jocul (este concatenarea mesajelor care descriu fiecare mutare în parte).

În fișierul sursă aveți implementată funcția `printGame` care vă ajută sa afișați frumos rezultatul întors de funcția `playGame`.

```
playGame :: Tabla -> [(Pozitie, Pozitie)] -> [(Pozitie, Pozitie)] -> EitherWriter Tabla
playGame = undefined
```

Succes!