A feladat annak kötelező, aki hivatalos neptun üzenetet kapott erről a tényről, hogy valamilyen szabálytalanság miatt kötelező egy ilyen feladatot elkészítenie. (Illetve mindazoknak is kötelező, akik lelkiismeretükben úgy érzik, hogy nem voltak tiszták a félévben.)

A jelenlegi eszközeinkkel már egy egész játékot lehet implementálni. Feladatunk egy teljes aknakereső játékot készíteni. A feladatokat bármilyen módon meg lehet csinálni, magasabb rendű függvény, rekurzió, listagenerátor, amelyik tetszik (hacsak az adott feladat szövege másképp nem rendelkezik). A játék konzolból játszható lesz!

A feladatok megoldása során **NE** használjunk parciális függvényeket, hiszen azok csúnya dolgok. Használjunk mintaillesztést jó sokat!

0. Modul létrehozása és mappaszerkezet létrehozása

Hozz létre egy új mappát `MineSweeper` néven. Ebbe a mappába kell majd az összes kódot pakolni, majd mappán kívülről lehet betölteni a kódot. Hozz létre egy modult `MineSweeper.Model` néven!

Hozd létre az alábbi 3 fájlt a `MineSweeper` mappán belül a nekik megadott megfelelő néven. (A TMS-ből is letölthető.)

`FieldNumber.hs`

```
module MineSweeper.FieldNumber (FieldNumber, fieldNumber, getFieldNumber) where

import Data.Function (on)
import Data.List (minimumBy)

newtype FieldNumber = FieldNumber {getFieldNumber :: Int} deriving (Eq, Ord)

instance Bounded FieldNumber where
    minBound = FieldNumber 0
    maxBound = FieldNumber 8

instance Enum FieldNumber where
    succ (FieldNumber int) = fieldNumber $ succ int
    pred (FieldNumber int) = fieldNumber $ pred int
    toEnum = fieldNumber
    fromEnum (FieldNumber int) = int
    enumFrom (FieldNumber start) = map fieldNumber $ enumFromThenTo start (succ start) 8
    enumFromThen (FieldNumber start) (FieldNumber next) = map fieldNumber $ enumFromThenTo start next $ f
    enumFromTo (FieldNumber start) (FieldNumber end) = map fieldNumber $ enumFromThenTo start (succ start
    enumFromThenTo (FieldNumber start) (FieldNumber next) (FieldNumber end) = map fieldNumber $ enumFromT

instance Show FieldNumber where
    showsPrec prec (FieldNumber integer) = showsPrec prec integer
```

```
instance Read FieldNumber where
    readsPrec prec str = map (first fromInteger) (readsPrec prec str) where
        first :: (val -> val') -> (val,other) -> (val',other)
        first f (val, other) = (f val, other)

instance Num FieldNumber where
    FieldNumber int1 + FieldNumber int2
        | 0 <= x && x <= 8 = FieldNumber x
        | otherwise = errorWithoutStackTrace "FieldNumber.(+): The sum of the two numbers exceeds 8!"
        where x = int1 + int2
    FieldNumber int1 - FieldNumber int2
        | 0 <= x && x <= 8 = FieldNumber x
        | otherwise = errorWithoutStackTrace "FieldNumber.(-): The difference of the two numbers subceeds
        where x = int1 - int2
    FieldNumber int1 * FieldNumber int2
        | 0 <= x && x <= 8 = FieldNumber x
        | otherwise = errorWithoutStackTrace "FieldNumber.(*): The product of the two numbers exceeds 8!"
        where x = int1 * int2
    abs = id
    signum x@(FieldNumber 0) = x
    signum _ = FieldNumber 1
    fromInteger = fieldNumber . fromIntegral

instance Real FieldNumber where
    toRational = toRational . toInteger

fieldNumber :: Int -> FieldNumber
fieldNumber i
    | i `elem` [0..8] = FieldNumber i
    | otherwise = errorWithoutStackTrace "FieldNumber: FieldNumber can only be from [0..8]!"

instance Integral FieldNumber where
    quotRem (FieldNumber int1) (FieldNumber int2) = let (quot,rem) = quotRem int1 int2 in (fieldNumber qu
    divMod (FieldNumber int1) (FieldNumber int2) = let (quot,rem) = divMod int1 int2 in (fieldNumber quot
    toInteger (FieldNumber int) = toInteger int
```

RandomPositionGenerator.hs

```
module MineSweeper.RandomPositionGenerator where

import System.Random ( mkStdGen, Random(..) )

type Seed = Int

generatePosition :: Seed -> (Int, Int)
generatePosition = fst . random . mkStdGen

generatePositions :: Seed -> [(Int, Int)]
generatePositions = randoms . mkStdGen

generatePositionR :: ((Int, Int), (Int, Int)) -> Seed -> (Int, Int)
generatePositionR a = fst . randomR a . mkStdGen
```

```
generatePositionRs :: ((Int, Int), (Int, Int)) -> Seed -> [(Int, Int)]
generatePositionRs a = randomRs a . mkStdGen
```

ConsoleView.hs

```
module MineSweeper.ConsoleView where

import System.Random ( randomIO )
import System.IO ( stdout, hFlush )
import MineSweeper.Model
    ( startNewGame
    , stepGame
    , Board(Board)
    , Field(Flagged, Closed, Opened)
    , FieldType(..)
    , Game(Game)
    , gameState
    , MoveType(..)
    , State(..) )
import Text.Read ( readMaybe )
import Data.List ( intercalate )

instance Show FieldType where
    show Mine = "X"
    show (Free x) = show x

instance Show Field where
    show (Closed _) = "_"
    show (Opened x) = show x
    show (Flagged _) = "F"

instance Show Board where
    show (Board _ _ x) = intercalate "\n" $ map (intercalate "|" . map show) x

instance Show State where
    show Running = ""
    show (GameOver p)
        | p           = "Nyertél!"
        | otherwise   = "Vesztettél! Bombát nyitottál!"

instance Show Game where
    show (Game b s) = intercalate "\n" [show b, show s]

iterateUntilM :: (Monad m) => (a -> Bool) -> (a -> m a) -> a -> m a
iterateUntilM p f v
    | p v       = return v
    | otherwise = f v >>= iterateUntilM p f

putStr' :: String -> IO ()
putStr' str = putStr str >> hFlush stdout

prompt :: Read a => String -> IO a
prompt str = putStr' str >> getReadable

getReadable :: Read a => IO a
getReadable = do
        input <- getLine
```

```
        case readMaybe input of
            Nothing -> putStrLn "Nem megfelelő típus! Próbálja újra!" >> getReadable
            Just x -> return x

readMaybeMoveType :: String -> Maybe MoveType
readMaybeMoveType str = case words str of
    "F":xs@[_,_] -> case map readMaybe xs :: [Maybe Int] of
        [Just a, Just b] -> Just $ Flag (a, b)
        _ -> Nothing
    xs@[_,_] -> case map readMaybe xs :: [Maybe Int] of
        [Just a, Just b] -> Just $ Open (a, b)
        _ -> Nothing
    _ -> Nothing

main :: IO ()
main = do
    seed <- randomIO :: IO Int
    w <- prompt "Adja meg a tábla szélességét: " :: IO Int
    h <- prompt "Adja meg a tábla magasságát: " :: IO Int
    b <- prompt "Adja meg a bombák számát!\n(Tábla méretének 10%-a és 50%-a között lesz a bombák száma):
    let initGame = startNewGame w h b seed
    print initGame
    iterateUntilM (\game -> case gameState game of Running -> False; _ -> True) (\game -> do
        move <- getLine
        case readMaybeMoveType move of
            Just x -> do
                let next = stepGame game x
                print next
                return next
            _ -> do
                putStrLn "Nem megfelelő beviteli formátum!"
                print game
                return game
        ) initGame
    return ()
```

Használati útmutató: Ha a játék kész lesz, akkor a `MineSweeper.ConsolView` modult importálva és a `main` függvényt futtatva lehet a játékot futtatni a ghci-ben. A fájlok betöltése kicsit körülményes. Ha van egy alapértelmezett mappád, ahol a ghci mindig indul és azon belül hoztad létre a `MineSweeper` mappát, akkor a legegyszerűbb, ekkor a ghci-ben csak `:l MineSweeper\ConsoleView.hs` parancsot kell kiadni. Ha valahol máshol indul a ghci, akkor a ghci-n belül a `:cd` parancs használatával a cmd-vel és mac és linux terminállal azonos módon el lehet navigálni odáig, ahol a `MineSweeper` mappa található és akkor lehet használni az előbbi `:l`-es kommandot. Vagy (bár ezt nem próbáltam ki) megadjuk a `:l`-nek a 4 betöltendő fájl abszolút útvonalát szóközökkel elválasztva és talán megy, de erre garanciát nem adok.

FIGYELEM! A `RandomPositionGenerator.hs`-ben van egy `import System.Random`-mal kezdődő sor. Ez a modul nem tartozik alapból a nyelvhez és a ghc telepítésekor egyik csomagban sincs benne. Ezt külön telepíteni kell. Nyissunk megy egy kedvenc konzolt vagy terminált operációs rendszertől függően. Két parancsot kell kiadni:

- `cabal v1-update`; várjuk meg, amíg ez befejezi a működését.
- `cabal v1-install random`

Ha sikerült ezt jól megtenni, akkor újranyitva a ghci-t a `System.Random` modul használható.

1. Típusszinonímák

Definiálj három típusszinonímát az alábbi neveken:

- `Width`
- `Height`
- `NumberOfBombs`

Mindhárom legyen szinonímája az `Int` típusnak. Az első a tábla szélességét, a második a tábla magasságát, a harmadik a bombák számát jelöli.

2a. Saját típusok

Hozd létre az alábbi típusokat, amivel fogjuk tudni kezelni a játék menetét:

- `FieldType`: Ezzel jelöljük, hogy egy mező számos vagy bombát tartalmaz. Legyen két konstruktora:
  - `Mine :: FieldType`: A mező bombát tartalmaz.
  - `Free :: FieldNumber -> FieldType`: A mező számot tartalmaz. A `FieldNumber` típus a `MineSweeper.FieldNumber` modulban található.

- `Field`: Ezen típus értékeiből tudjuk kideríteni, hogy egy mező milyen állapotban van. Három állapot lehetséges, így azok legyenek a konstruktorok:
  - `Closed :: FieldType -> Field`: Még nem nyitottuk ki a mezőt és nincs rajta bombát jelölő zászló.
  - `Opened :: FieldType -> Field`: Nyitott mező.
  - `Flagged :: FieldType -> Field`: Még nem nyitott mező bombát jelölő zászlóval. Mindhárom tartalmazza, hogy konkrétan mi van a mezőn.

- `Board`: Ebben a struktúrában tároljuk a táblánkat és a hozzá tartozó adatokat. Az egyetlen konstruktor neve ugyanez legyen, amelynek a típusa `Board :: Width -> Height -> [[Field]] -> Board`
- `State` : Ezzel tartjuk számon, hogy a játék folyik-e még vagy már vége van és hogy mi lett az eredmény. Épp ezért ennek két konstruktora lesz:
  - `Running :: State`, amely jelzi, hogy a játék még tart.
  - `GameOver :: Bool -> State`, amelyben eltároljuk azt, hogy a játékos nyert-e a játék végén.

- `Game` : Ez fogja az egész játékállást tárolni, valamennyi lépésben ezt az állapotot kell változtatni. Legyen egy `Game :: Board -> State -> Game` konstruktora, amely tartalmazza a táblát, illetve az állapotot, hogy a játék megy még vagy már vége van-e.

A fordítótól nem kérhetünk segítséget, illetve `Eq`, `Enum` példányt ebben a fájlban definiált saját típusra sem lehet megadni, tessék mintaillesztést használni!

2b. Getter függvények

Definiáld a fenti `Board` és `Game` típusokhoz a getter függvényeket.

- `getWidth :: Board -> Width` : Visszaadja a `Board` konstruktornak a `Width` típusú paraméterét.
- `getHeight :: Board -> Height` : Visszaadja a `Board` konstruktornak a `Height` típusú paraméterét.
- `getField :: Board -> [[Field]]` : Visszaadja a `Board` konstruktornak a `[[Field]]` típusú paraméterét.
- `gameBoard :: Game -> Board` : Visszaadja a `Game` konstruktornak a `Board` típusú paraméterét.
- `gameState :: Game -> State` : Visszaadja a `Game` konstruktornak a `State` típusú paraméterét.

3a. Biztonságos indexelés

Definiáljuk a `(!!?) :: [a] -> Int -> Maybe a` függvényt. A függvény `Nothing`-ot ad vissza, ha rosszul indexelünk a listába, egyébként `Just`-ban adjuk vissza az eredményt.

```
[1,0,9,8,10,12,33,0,3] !!? 5 == Just 12
[1,0,9,8,10,12,33,0,3] !!? 0 == Just 1
[1,0,9,8,10,12,33,0,3] !!? (-1) == Nothing
[1,0,9,8,10,12,33,0,3] !!? 9 == Nothing
[] !!? 5 == Nothing
"alma" !!? 2 == Just 'm'
"alma" !!? 3 == Just 'a'
"alma" !!? 4 == Nothing
"alma" !!? (-10) == Nothing
```

3b. `Maybe`-n belüli érték

Definiáljuk a `maybeMap :: (a -> b) -> (Maybe a -> Maybe b)` függvényt, amely egy függvényt átalakít olyan függvénnyé, amely `Maybe a` értékeken végzi el a műveletet.

```
maybeMap (+1) (Just 1) == Just 2
maybeMap (+1) Nothing == Nothing
maybeMap (take 1) (Just [1,2,3]) == Just [1]
maybeMap (take 1) (Just []) == Just []
```

3c. Több `Maybe` összevonása

Definiáljuk a `joinMaybe :: Maybe (Maybe a) -> Maybe a` függvényt, amellyel két `Maybe`-t lehet eggyé összevonni. Csak akkor lesz értékünk, ha mindkét `Maybe` a `Just` konstruktorral van definiálva, tehát van konkrét értékünk, ellenkező esetben `Nothing`.

```
joinMaybe (Just (Just 1)) == Just 1
joinMaybe (Just (Just [1])) == Just [1]
joinMaybe (Just (Just "alma")) == Just "alma"
joinMaybe (Just (Just 'a')) == Just 'a'
joinMaybe (Just Nothing) == Nothing
joinMaybe Nothing == Nothing
```

4. Egyértelmű medián számolása

Számoljuk ki egy lista mediánját a `singleMedian :: Ord a => [a] -> Maybe a`. Egyértelmű medián szükséges nekünk, így akkor is adjunk vissza `Nothing`-ot, ha a medián a két középső elem átlagának kiszámolásából jönne ki. Ezzel a függvénnyel lesz egyszerű megoldani a szükséges későbbi kikötéseket, hogy hány akna lehet a táblán. Egy rendezett adatsor mediánja annak a középső eleme. Pl. medián [6,5,9,1,0,2,8] -> rendezzük az adatokat -> medián [0,1,2,5,6,8,9] == 5, az a 4. elem balról és jobbról is egy 7 elemű adatsorból, tehát az a középső.

Segítség: A függvényhez definiáljunk egy segédfüggvényt, amelynek adjuk át kétszer a rendezett listát. Az egyik listán lépkedjünk egyesével, a másikban kettesével.

- Ha az, amelyikben kettesével lépkedtünk, elért egy pontosan egy elemű listáig, akkor a másikban az első elem a medián, ekkor adjuk vissza azt.
- Ha az, amelyikben kettesével lépkedtünk, elért egy pontosan két elemű listáig, akkor nincs egyértelmű medián.
- Ha több elem van mindkét listában, akkor az egyiken lépjünk egyet, a másikon kettőt.

- Ha valamelyik lista üres, akkor adjunk vissza `Nothing`-ot.

```
singleMedian [0,1,2,5,6,8,9] == Just 5
singleMedian [0,1,2,5,6,8,9,10] == Nothing
singleMedian [] == Nothing
singleMedian "a" == Just 'a'
singleMedian "alma" == Nothing
singleMedian "körte" == Just 'r'
singleMedian [9,0,10,5,7] == Just 7
```

5. Rendezett listában elem

Definiáljuk a `sortedElem :: Ord a => a -> [a] -> Bool` függvényt, amely egy rendezett listában nézi meg, hogy egy adott elem benne van-e a listában. Ez annyitól lesz jobb, mint a sima `elem` függvény, hogy ha az elemünk nagyobb, mint ahol tartunk a vizsgálatban, akkor ott már rövidre lehet zárni a keresést. Ez akár végtelen listán is működik, pl. `sortedElem 0 [1..] == False` lesz és nem fog a végtelenségig dolgozni. Persze, ehhez fel kell tenni, hogy a lista rendezett.

```
not $ sortedElem 0 [1..]
sortedElem 10 [1,4,7,9,10,11,13]
sortedElem 'k' "aaaabbbccddeefghijkllllmmmmmmzzzzzz"
not $ sortedElem 10 [1,3 ..]
```

6. Bombák helyének generálása

Definiáljuk a `generateBombsPositions :: Width -> Height -> NumberOfBombs -> Seed -> [(Int,Int)]` függvényt. Értelemszerűen kell egy adott táblára adott számú bomba, tehát annyi elemű listát adunk vissza. A listánk legyen rendezett a rendezett pár első, majd a második eleme alapján. A bombák száma minimum a tábla méretének 10%-a, de legfeljebb 50%-a. Tehát például egy 15x12-es táblára kell legalább 15x12/10 = 180/10 = 18 db bomba, de legfeljebb 90 bomba lehet. Ha a minimumnál kisebb értéket adunk át, akkor a bombák száma legyen a minimum. Ha maximumnál nagyobb értéket adunk át, akkor pedig legyen a maximum.

Segítség: Használjuk fel a `generatePositionRs` függvényt, amely a `MineSweeper.RandomPositionGenerator`-ben található. A random elemek között lehet több azonos is, erre figyelni kell. A `Seed` típusszinonímа szintén az előbb definiált modulban található. Ezen paraméterre a tisztaság miatt van szükség. Azonos `Seed`-ek esetén nyilván azonos bombákat kapunk vissza.

```
generateBombsPositions 10 10 60 0 == [(0,0),(0,1),(0,6),(0,7),(0,8),(0,9),(1,0),(1,1),(1,4),(1,5),(1,6),(
generateBombsPositions 10 10 50 0 == [(0,0),(0,1),(0,6),(0,7),(0,8),(0,9),(1,0),(1,1),(1,4),(1,5),(1,6),(
generateBombsPositions 10 10 10 0 == [(0,8),(3,2),(4,3),(5,1),(7,4),(7,8),(8,6),(8,8),(9,4),(9,7)]
generateBombsPositions 10 10 10 1 == [(0,0),(1,1),(1,6),(3,3),(3,7),(5,2),(7,1),(8,3),(8,5),(9,5)]
generateBombsPositions 10 10 0 1 == [(0,0),(1,1),(1,6),(3,3),(3,7),(5,2),(7,1),(8,3),(8,5),(9,5)]
generateBombsPositions 6 6 10 1 == [(0,3),(1,2),(2,0),(2,3),(2,4),(3,3),(4,2),(4,5),(5,3),(5,4)]
```

7. Szomszédos mezők

Definiáljuk a `neighbours :: (Int,Int) -> [[a]] -> [[a]]` függvényt, amely visszaadja egy mezőhöz tartozó legfeljebb 8 szomszédos mezőt és önmagát. Szomszédnak a függőlegesen, vízszintesen és átlósan 1-1 mező

távolságban lévő mezők számítanak. Listák tekintetében ez annyit jelent, hogy egy értéknek a szomszédja egy másik érték:

- ha azonos listán belül egymást követő elemek.
- ha két egymást követő listában azonos helyen lévő elemek.
- ha két egymást követő listában két egymást követő helyen lévő elemek.

A tesztekből talán egyértelműbb, hogy mit kell csinálni.

```
neighbours (1,1) [[1,2,3], [4,5,6], [7,8,9]] == [[1,2,3], [4,5,6], [7,8,9]]
 -- ebben az esetben az (1,1) helyen az 5-ös található, így mindegyik elem a szomszédja
neighbours (0,1) [[1,2,3], [4,5,6], [7,8,9]] == [[1,2,3], [4,5,6]]
 -- ebben az esetben az elemünk a 2-es, a 7,8,9 egyike sem szomszédja a 2-nek.
neighbours (0,0) [[1,2,3], [4,5,6], [7,8,9]] == [[1,2], [4,5]]
 -- ebben az esetben az elemünk az 1-es, 3,6,7,8,9 nem szomszédja az 1-esnek.
neighbours (1,1) [[1,2], [3,4,5], [6,7,8,9], [10]] == [[1,2], [3,4,5], [6,7,8]]
 -- ebben az esetben az elemünk a 4-es, a 9 és 10-es nem szomszédok a 4-essel.
```

A lista nyilván nem csak 3x3-as lehet, az se biztos, hogy egyáltalán mindegyik listák hosszai ugyanannyik.

8. Szomszédos bombák

Definiáljuk a `countNeighbouringBombs :: (Int, Int) -> [(Int, Int)] -> Int` függvényt, amely megkap egy pozíciót és a bombák helyének listáját. Számoljuk meg, hogy az adott mezőnek hány szomszédja bomba. A mezőt önmagát ne számoljuk bele akkor se, ha az bomba (nyilván egy mező nem szomszédos önmagával).

```
countNeighbouringBombs (0,1) (generateBombsPositions 10 10 0 1) == 2
countNeighbouringBombs (8,4) (generateBombsPositions 10 10 0 1) == 3
countNeighbouringBombs (0,0) (generateBombsPositions 10 10 0 1) == 1
countNeighbouringBombs (7,2) (generateBombsPositions 10 10 50 1) == 7
countNeighbouringBombs (7,5) (generateBombsPositions 10 10 50 14) == 8
```

9. Játéktábla elkészítés

Definiáljuk a `generateBoard :: Width -> Height -> [(Int, Int)] -> Board` függvényt, amely a tábla szélességéből és magasságából, továbbá a kapott bombák listájából előállítja a teljes táblát, amin játszani fogunk. Amelyik mezőn bomba szerepel, arra a helyre nyilván a `Mine` érték kerül, más helyekre pedig a `Free`-vel jelölt `FieldNumber` típusú szám. A `FieldNumber` típus a `MineSweeper.FieldNumber` modulban található. Fontos, hogy a játék kezdetén az összes mező zárt állapotú. Feltehető, hogy a listában a megadott bombák mindig a táblán helyezkednek el.

```
case generateBoard 10 10 [(0,0),(1,1),(1,6),(3,3),(3,7),(5,2),(7,1),(8,3),(8,5),(9,5)] of Board 10 10 [[C
case generateBoard 10 10 [(0,8),(3,2),(4,3),(5,1),(7,4),(7,8),(8,6),(8,8),(9,4),(9,7)] of Board 10 10 [[C
```

10. Játék kezdőállapota

Definiáljuk a `startNewGame :: Width -> Height -> NumberOfBombs -> Seed -> Game` függvényt, amely előállítja egy játéknak a kezdőállapotát a megadott paramétereknek megfelelően. Szükséges a tábla szélessége,

magassága, a bombák száma és egy "játékszám", tehát a `Seed`, amiből generáljuk a random elemeket. A játék kezdetén az összes mező zárt állapotú, ezt a `generateBoard` függvény megoldja. A játék állapota legyen futó, tehát `Running`.

```
case startNewGame 5 5 10 40 of Game (Board 5 5 [[Closed Mine,Closed Mine,Closed (Free 3),Closed Mine,Clos
case startNewGame 10 10 10 40 of Game (Board 10 10 [[Closed (Free 0),Closed (Free 0),Closed (Free 1),Clos
case startNewGame 15 5 20 2347832478 of Game (Board 15 5 [[Closed (Free 1),Closed (Free 1),Closed (Free 2
```

11a. Elemek cseréje

Definiáljuk a `replaceAt :: Int -> a -> [a] -> [a]` függvényt, amely egy listában egy adott helyen lévő elemet lecserél egy másikra.

```
replaceAt 1 'a' "Hello" == "Hallo"
replaceAt 2 3 [2,9,5,4,5] == [2,9,3,4,5]
replaceAt 2 [1,2] [[2,9,5],[4,5],[3,4]] == [[2,9,5],[4,5],[1,2]]
replaceAt 3 (Free 1) [Mine, Mine, Free 0, Free 4, Free 5, Mine] == [Mine, Mine, Free 0, Free 1, Free 5, M
```

11b. Elemek cseréje listák listájában

Definiáljuk a `replaceAtMatrix :: (Int, Int) -> a -> [[a]] -> [[a]]` függvényt, amely egy listák listájában egy adott helyen lévő elemet lecserél egy másikra. Az első index a lista sorszámát, a második az elem sorszámát adja meg.

```
replaceAtMatrix (0,0) 'a' ["elma", "barack", "szilva"] == ["alma", "barack", "szilva"]
replaceAtMatrix (2,5) 100 [[1,2,3,4], [5,6,7,8,9,10,11,12], [13,14,15,16,17,18,19,20], [21,22],[23,24,25,
replaceAtMatrix (4,3) Mine [[Free 1], [], [Free 3, Free 4, Free 2, Mine, Mine, Free 1], [Mine, Free 5, Fr
```

12. Egy mező kinyitása

Definiáljuk az `openField :: Field -> Field` függvényt, amely segítségével egy mezőt ki lehet nyitni. Tehát ha `Closed` állapotú a mező, akkor legyen `Opened`. Zászlózott mezőt nem lehet kinyitni, illetve a nyitott mező már nyitva van, így azokban az esetekben adjuk vissza önmagát a mezőt.

```
case openField (Closed (Free 0)) of Opened (Free 0) -> True; _ -> False
case openField (Closed Mine) of Opened Mine -> True; _ -> False
case openField (Opened (Free 1)) of Opened (Free 1) -> True; _ -> False
case openField (Flagged (Free 1)) of Flagged (Free 1) -> True; _ -> False
```

13. Mező kinyitása a táblán

FIGYELEM! Nem triviális ezt megoldani, ne maradjon ez a végére!
Definiáljuk az `openTile :: Game -> (Int, Int) -> Game` függvényt, amely egy játékállapotot és egy adott mezőt

kapva visszaad egy új játékállapotot. Ha minden szabályosan történik, tehát az index valóban a táblára mutat, zárt mezőt szeretnénk kinyitni és fut a játék, akkor kinyitja azt a mezőt. Figyelni kell arra, hogy ha a mező 0-ás számot takar, akkor a szomszédos mezőket is szintén ki kell nyitni. Ha egy adott számú már nyitott mezőt szeretnénk nyitni, akkor ellenőrizzük, hogy a körülötte lévő zászlók száma megegyezik-e a mezőn található számmal. Ha egyezik, akkor nyissuk ki a körülötte lévő összes zárt mezőt. Ha valami szabálytalanság történik vagy a játék `GameOver` állapotban van, akkor a játékállapot nem változik.

Segítség: A függvényben kell arra figyelni, hogy mi történik akkor, ha 0-át nyitunk, illetve ha nem 0-át nyitunk. Ehhez szükségünk lesz két segédfüggvényre, amelyeknek a típusa megegyezik az `openTile` típusával. Az egyik kezeli azt, amikor csak egy mezőt kell kinyitni és nem többet. A másik fogja kezelni azt, amikor több mezőt kell kinyitni, akár a 0-ák, akár egy nyitott mezőre való kattintáskor.

Első segédfüggvény:

Ha nem 0-át nyitunk:

- Nyissuk ki azt az adott zárt mezőt. Fontos, hogy zárt legyen kezdetben, mert az `openField` nem tudja belülről detektálni, hogy mikor kell egy külső függvénynek végeznie.
- Építsük fel a `Game` értékét úgy, hogy abban már kinyitjuk ezt az adott mezőt.
- A függvény működése álljon le, tehát nem kell se rekurzió, se más.

Ha 0-át nyitunk:

- Nyissuk ki azt az adott zárt mezőt. Fontos, hogy zárt legyen kezdetben, mert az `openField` nem tudja belülről detektálni, hogy mikor kell egy külső függvénynek végeznie.
- Építsük fel a `Game` értékét úgy, hogy abban már kinyitjuk ezt az adott mezőt.
- Lépjünk át a másik segédfüggvénybe, amelyben kinyitjuk a 0-ának a szomszédait. Az előbb felépített `Game` értéket adjuk tovább annak.

Másik segédfüggvény:

- Hívjuk meg az első segédfüggvényt a szomszédos indexekkel.
- Szükséges a táblát folyamatosan szükséges frissíteni, így talán ezt a legegyszerűbb hajtogatással megoldani. Hogy melyik irányú hajtogatás kell, annak a meghatározásában segít a típus.

Ha zárt mezőt nyitunk ki, akkor az `openTile`-ból az első segédfüggvényt kell meghívni. Ha pedig nyitottat próbálunk még egyszer kinyitni, tehát azon mező szomszédait szeretnénk kinyitni, akkor a másik segédfüggvényt kell meghívni.

```
-- 0-ás mező
case openTile (Game (Board 5 7 [[Closed (Free 0),Closed (Free 1),Closed Mine,Closed (Free 2),Closed (Free
-- bombás mező
case openTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Closed (Free 2),Closed (Free
-- game over
case openTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Opened Mine,Closed (Free 2),Closed (Free
-- már nyitott mező
case openTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Closed (Free 2),Closed (Free
-- zászlós mező
case openTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Closed (Free 2),Closed (Free
-- zászlós szomszéd
case openTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Closed (Free 2),Closed (Free
```

14. Mező zászlózása

Definiáljuk a `flag :: Field -> Field` függvényt, amellyel egy zárt mezőre zászlót lehet helyezni, egy zászlós mezőről pedig azt levenni. Nyitott mezőt ez nem befolyásolja, az maradjon ugyanaz.

```
case flag (Opened (Free 1)) of Opened (Free 1) -> True; _ -> False
case flag (Closed (Free 1)) of Flagged (Free 1) -> True; _ -> False
case flag (Flagged (Free 1)) of Closed (Free 1) -> True; _ -> False
case flag (Closed Mine) of Flagged Mine -> True; _ -> False
case flag (Flagged Mine) of Closed Mine -> True; _ -> False
```

15. Mező zászlózása a táblán

Definiáljuk a `flagTile :: Game -> (Int, Int) -> Game` függvényt, amely egy játékállapotot és egy helyet kap paraméterül. Ha a játék fut, akkor tegyünk zászlót a megadott mezőre vagy vegyük azt onnan le. Szabálytalan esetben ne változtassunk a játékállapoton.

```
-- game over
case flagTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Opened Mine,Closed (Free 2),Closed (Free
-- zárt mező zászlózása
case flagTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Closed (Free 2),Closed (Free
-- zászló levétele
case flagTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Flagged (Free 2),Closed (Fre
-- nyitott mező
case flagTile (Game (Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Flagged (Free 2),Closed (Fre
```

16. Nyitott bombás mezők keresése

Definiáljuk a `checkOpenedBomb :: Board -> Bool` függvényt, amely meghatározza, hogy van-e nyitott bombás mező a táblán.

```
not $ checkOpenedBomb $ Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Flagged (Free 2),Closed (
not $ checkOpenedBomb $ Board 1 1 [[Closed Mine]]
checkOpenedBomb $ Board 1 1 [[Opened Mine]]
checkOpenedBomb $ Board 5 7 [[Opened (Free 0),Opened (Free 1),Closed Mine,Flagged (Free 2),Closed (Free 1
```

17. Játék vége

Definiáljuk a `checkGameOver :: Game -> Game` függvényt. A játéknak kétféleképpen lehet vége. Vagy kinyitottuk az összes nem bomba mezőt vagy bombát nyitottunk. Előbbi esetben a játékos nyert, ezt a `GameOver` konstruktornak átadott `True` értékkel kell jelezni. Ha a játékos bombát nyitott, akkor jelenítsük meg az összes bombát, hogy hol voltak. Az egyszerűség kedvéért a zászlósakat is nyissuk ki ebben az esetben. Azzal nem kell foglalkozni, hogy zászló volt olyan helyen, ahol nem volt bomba.

18. Lépés típusa

Definiáljunk egy saját típust `MoveType` néven, amelynek legyen két konstruktora:

- `Flag :: (Int, Int) -> MoveType`, ez a konstruktor mondja meg, hogy mi zászlót szeretnénk helyezni a táblára.
- `Open :: (Int, Int) -> MoveType`, ez a konstruktor mondja meg, hogy mi ki szeretnénk nyitni az adott mezőt.

19. Egy lépés a játékban

Definiáljuk a `stepGame :: Game -> MoveType -> Game` függvényt, amellyel egyet lépünk a játékban. Kap egy játékállapotot és egy indexet, hogy melyik mezővel szeretnénk akciót végezni. Ha `Flag`, akkor zászlót szeretnénk lehelyezni. Ha `Open`, akkor mezőt szeretnénk nyitni. Végül szükséges ellenőrizni, hogy a játéknak vége van-e.