Funkcionális programozás 8. gyakorlat

Saját adattípusok

Színek

Definiáljuk a `Colour` adattípust, amely segítségével színeket reprezentálhatunk RGB (piros, zöld, kék) formátumban.

```
data Colour = RGB Int Int Int --deriving (Show, Eq)

red :: Colour
red = RGB 255 0 0

green :: Colour
green = RGB 0 255 0

blue :: Colour
blue = RGB 0 0 255

instance Show Colour where
    show (RGB r g b) = "R = " ++ show r ++ " G = " ++ show g ++ " B = " ++ show b

instance Eq Colour where
    (==) (RGB r1 g1 b1) (RGB r2 g2 b2) = r1 == r2 && g1 == g2 && b1 == b2
```

Definiáljuk az `isGray :: Colour -> Bool` függvényt, amely eldönti, hogy a paraméterként megadott szín a szürke egy árnyalata-e! Ez akkor teljesül, ha a szín három komponense egyenlő, de nem a fehér (rgb = (255, 255, 255)) illetve nem a fekete (rgb = (0, 0, 0)).

Előjeles egészek

1. Definiáljuk az előjeles egész számok reprezentálására a Numbers adattípust, amely rendelkezzem az egy `Int` paraméteres `Positive` és `Negative` illetve a paraméter nélküli `Zero` adatkonstruktorokkal. Kérjük az `Eq` és a `Show` típusosztályok automatikus példányosítását!
2. Definiáljuk a `mkNumbers :: Integer -> Numbers` függvényt, amely egy `Integer`-ből `Numbers` típusú számot állít elő!
3. Definiáljuk a `fromNumbers :: Numbers -> Integer` függvényt, amely egy `Numbers` típusú számból `Integer`-t állít elő!
4. Definiáljuk az `addPositive :: [Numbers] -> Numbers` függvényt, amely összeadja egy `Numbers` típusú elemeket tartalmazó listából a pozitív számokat!

Alakzatok

1. Hozzunk létre egy `Point` adattípust, amely egy síkbeli pontot fog reprezentálni. Egyetlen adatkonstruktora (`Point`) két `Double` típusú paraméterrel rendelkezzen, amely az x és y koordinátáját adja meg.
2. Definiáljuk a `getX :: Point -> Double` és `getY :: Point -> Double` függvényeket, amely visszaadja egy `Point` x, illetve y koordinátáját!
3. Definiáljuk a `displace :: Point -> Point -> Point` függvényt, amely eltol egy pontot a síkon!
4. Definiáljuk a Shape adattípust típust, amely egy `Circle` és egy `Rect` konstruktorral rendelkezik. Egy kört a középpontja és a sugara segítségével tudunk reprezentálni. A téglalapokat a bal felső és jobb felső csúcsa segítségével reprezentáljuk. A `Circle` konstruktor rendelkezzen egy a kör középpontját megadó `Point` típusú paraméterrel és egy a sugarat megadó `Double` típusú paraméterrel. A `Rect` adatkonstruktor egy-egy `Point`-ként kapja meg a téglalap bal felső illetve jobb alsó sarkát! Kérjük az `Eq` és a `Show` típusosztályok automatikus példányosítását!
5. Definiáljuk az `area :: Shape -> Double` függvényt, amely kiszámítja egy alakzat területét!
6. Definiáljuk a `displaceShape :: Shape -> Point -> Shape` függvényt, amely eltol egy alakzatot a síkon.

Maybe

```haskell
data Maybe a = Nothing | Just a
    deriving (Show, Eq)
```

1. Definiáljuk a `safeHead :: [a] -> Maybe a` függvényt, amely visszaadja egy lista első elemét `Just elem` formában. Ha a lista üres, adjon vissza `Nothing`-ot.
2. Definiáljuk a `safeDiv :: Double -> Double -> Maybe Double` függvényt, amely megvalósítja a biztonságos osztás műveletét. Ha a második paraméter 0, adjon vissza a függvény `Nothing`-ot, egyébként pedig az osztás eredményét.
3. Definiáld a `safeIndex :: [a] -> Int -> Maybe a` függvényt, ami biztonságos módon adja vissza egy lista n-edik elemét. Ha az index túl kicsi vagy túl nagy az eredmény legyen `Nothing`, különben az eredmény legyen a lista n-edik eleme `Just`-ba csomagolva. Az indexelés kezdődjön 0-tól.

```haskell
safeIndex [] 3 == Nothing
safeIndex [1] 0 == Just 1
safeIndex [1,2,3,4] (-2) == Nothing
safeIndex [1,2,3] 2 == Just 3
safeIndex [1..10] 33 == Nothing
```

```
safeIndex "haskell" 4 == Just 'e'
safeIndex [1..] 9 == Just 10
```

4. Adott egy `Maybe a` értékeket tartalmazó lista. Számold meg, hány `Nothing` szerepel a listában!

```
countNothings :: [Maybe a] -> Int

countNothings [] == 0
countNothings [Just 3, Just 5] == 0
countNothings [Just "Haskell", Just "Clean"] == 0
countNothings [Just "Haskell", Just "Clean", Nothing] == 1
countNothings [Nothing, Just "Haskell", Just "Clean", Nothing] == 2
countNothings [Nothing, Just "Haskell", Nothing, Just "Clean", Nothing] == 3
countNothings [Nothing, Just Nothing, Nothing, Just Nothing, Nothing] == 3
```

5. Egy `Maybe`-be csomagolt elemet fűzz egy lista elejére! Ha az elem `Nothing`, az eredmény az eredeti lista legyen!

```
addBefore :: Maybe a -> [a] -> [a]

addBefore (Just 1) [2,3,4] == [1,2,3,4]
addBefore (Just 'E') "LTE" == "ELTE"
addBefore Nothing [True, False] == [True, False]
```

Gyakorló feladat mintaillesztéshez

Adott a következő adattípus:

```
data Gyumolcs = Alma Int Char | Barack | Cseresznye String
```

1. Adj meg olyan kifejezést, amelyet ha paraméterül adunk a `foo1` függvénynek 1, 2 vagy 3 lesz a visszatérési érték!

```
foo1 :: [Gyumolcs] -> Int
foo1 [Alma 1 'a',_, Cseresznye _] = 0
foo1 (Alma 1 'a':_: Cseresznye _:_) = 1
foo1 (Alma _ _:[]) = 2
foo1 (Alma x y:_:xs)
    | x > 100 = 3
```

2. Adj meg olyan kifejezést, amelyet ha paraméterül adunk a `foo2` függvénynek 1, 2 vagy 3 lesz a visszatérési érték!

```
foo2 :: [(Gyumolcs, Gyumolcs)] -> Int
foo2 [(Barack, Cseresznye ('z':_))] = 0
foo2 ((Alma _ _, _):(_,Barack):_) = 1
foo2 ((Barack,_):(Alma _ 'a', Alma 1 _):xs) = 2
foo2 (_:(Alma _ 'a', Alma 1 _):xs) = 3
```