A házi feladatot egy `Homework7` nevű modulként kell beadni. Figyeljetek arra, hogy a függvényeitek a module szóval egy "oszlopba" kerüljenek, azaz ne legyenek beljebb húzva! Minden definiálandó függvényhez adjuk meg a hozzá tartozó típus szignatúrát is! (Ezt most megadtam, a saját modulotokba is másoljátok be a definíciótok elé.)

Rekurzív típusok

Definiáljunk egy `Tree a` nevű típust (ahol `a` egy típusparaméter). Az adattípust egy bináris fát fog reprezentálni a feladatok során. Egy bináris fa hasonló egy listához, csak ahelyett, hogy a `:` konstruktorban 1 db `a` típusú paraméter és 1 db rekurzív paraméter van, a fában 2 db rekurzív paraméter van ÉS nem lehet üres fát leírni. A definíciója

```
data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving (Eq, Show)
```

Példa egy bináris fára:

```
-- Ezt másold be a kódba
tr1 :: Tree Int
tr1 = Node (Leaf 1) 2 (Node (Node (Leaf 3) 4 (Leaf 5)) 6 (Leaf 7))
{- Kirajzolva:

    2
  /   \
 1     6
     /   \
    4     7
  /   \
 3     5
-}
```

- Definiáljunk a `map` műveletet fára! A függvény az összes `a` típusú kifejezést cserélje le a függvény által `b` típusúra. (`mapTree :: (a -> b) -> Tree a -> Tree b`)
- Definiáljunk egy függvényt ami egy fában az összes számot összeadja! (`sumTree :: Num a => Tree a -> a`)

Tesztek:

```
mapTree (+1) tr1 == Node (Leaf 2) 3 (Node (Node (Leaf 4) 5 (Leaf 6)) 7 (Leaf 8))
sumTree tr1 == 28
```

Vegyük az órán írt `Stream` típust.

```
-- Ezt is másoljátok be a kódba
data Stream a = SCons a (Stream a)

takeStream :: Int -> Stream a -> [a]
takeStream x (SCons a as)
    | x <= 0 = []
    | otherwise = a : takeStream (x - 1) as

repeatStream :: a -> Stream a
repeatStream x = SCons x (repeatStream x)

iterateStream :: a -> (a -> a) -> Stream a
iterateStream x f = SCons x (iterateStream (f x) f)
```

- Definiáljuk a `zipWith` függvény `Stream`-re! A függvény páronként kombinálja a streamek elemeit! (`zipWithStream :: (a -> b -> c) -> Stream a -> Stream b -> Stream c`)
- Definiáljuk a `takeWhile` függvényt `Stream`-re! A függvény addig veszi az elemeket egy streamből amíg igaz az adott predikátum! (`takeWhileStream :: (a -> Bool) -> Stream a -> [a]`)

Tesztek:

```
takeStream 3 (zipWithStream (,) (repeatStream 1) (repeatStream 2)) == [(1,2),(1,2),(1,2)]
takeWhileStream (<5) (iterateStream 1 (+1)) == [1,2,3,4]
```

Peano számok

A természetes számokat funkcionális nyelvekben szokás a teljes indukció segítségével reprezentálni. Definiáljunk egy `Nat` adattípust két konstruktorral amelyeknek a típusai az alábbiak:

- `Zero :: Nat`
- `Succ :: Nat -> Nat`

A típusban a `Zero` konstruktor a 0-t reprezentálja a `Succ` (azaz successor) pedig a `+1` függvényt. Azaz ha pl 6-ot szeretnénk leírni ezzel a konstrukcióval, az így nézne ki:

```
six :: Nat
six = Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
-- 6 db Suc
```

Használjuk az alábbi `Show` instance-ot a feladat során

```
peanoToInt :: Nat -> Int
peanoToInt Zero = 0
peanoToInt (Succ x) = 1 + peanoToInt x

instance Show Nat where
    show p = "Peano " ++ show (peanoToInt p)
```

- Definiáljunk (értelemszerű) `Eq` instance-ot a saját típusunkra MANUÁLISAN! Az instance írás során ne használjuk a `peanoToInt` függvényt vagy a típus `Show` instance-át.
- Definiáljunk (értelemszerű) `Ord` instance-ot a saját típusunkra MANUÁLISAN! A fenti szabályok erre a feladatra is applikálnak.
- Írjunk `length` függvény listára ami a saját típusunkra képez! (`lengthP :: [a] -> Nat`)

Tesztek:

```
Zero == Zero
Succ Zero /= Zero
Zero /= Succ Zero
Succ (Succ Zero) == Succ (Succ Zero)
Zero < Succ (Succ Zero)
Zero >= Zero
lengthP [1,2,4] == Succ (Succ (Succ Zero))
lengthP [1..] > lengthP [1]
```