A házi feladatot egy `Homework6` 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.)

Egyszerű `data`

Golf kifejezések

A golfban a pontozás az alapján megy, hogy hány ütésből tudta valaki belőni a labdát a lyukba - minél kevesebb annál jobb. Ehhez az alábbi terminológiát szokás használni:

- `Ace` - Amikor valaki első ütésre belövi a labdát
- `Albatross` - Amikor valaki a lyuk limitje alatt legalább 3-al lőtte be a golflabdát
- `Eagle` - Amikor valaki a lyuk limitje alatt 2-vel lőtte be a labdát
- `Birdie` - Amikor valaki a lyuk limitje alatt 1-el lőtte be a labdát
- `Par` - Amikor valaki a lyuk limitjével egyező lövésszámmal lőtte be a golflabdát
- `Bogey` - Amikor valaki túllépi a limitet

Definiáljunk egy `GolfScore` adattípust, aminek a fenti konstruktorai vannak. A `Bogey`-nak legyen egy Int paramérere ami azt reprezentálja, mennyivel léptük túl a limitet.

- Definiáljunk egy `score` függvényt ami egy limit és egy lövésszám alapján kiszámolja mi a pontszáma valakinek! (`score :: Int -> Int -> GolfScore`)
- Kérjünk a fordítótol automatikus `Show` instance megírását erre a típusta!
- Írjunk manuálisan `Eq` instanceot erre a típusra!

Tesztek a működésre:

```
score 2 3 == Bogey 1
score 3 2 == Birdie
score 4 1 == Ace
score 103 1 == Ace
score 103 2 == Albatross
score 103 100 == Albatross
score 103 101 == Eagle
score 10 10 == Par
```

`Maybe` típus

Az orán vettük a `Lehet a` típust, ami azt reprezentálta, hogy lehet, hogy tárol `a` típusú kifejezést. Ezt a standard library-ben `Maybe`-nek hívják és így van definiálva

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

Ugye itt a `Van`-nak a `Just`, a `Nincs`-nek meg a `Nothing` felel meg.

- Definiáljunk egy függvényt ami egy `a` típusú kifejezést kap paraméterül és egy `Maybe a`-t. Ha a `Maybe a` típusú kifejezés tárol magában `a` típusú kifejezést, adjuk azt vissza, különben adjuk vissza az első paramétert! (`withFallback :: a -> Maybe a -> a`)

- Definiáljunk egy függvényt ami egy paraméterül kapott függvényt alkalmaz egy `Maybe a`-ban tárolt elemre (ha van benne!) (`maybeMap :: (a -> b) -> Maybe a -> Maybe b`)
- Definiáljunk egy függvényt ami egy nestelt `Maybe` kifejezésből egy darab `Maybe`-t csinál. (`joinMaybe :: Maybe (Maybe a) -> Maybe a`)
- Definiáljunk egy függvényt ami leszűri a `Nothing`-okat egy listából. (`filterNothing :: [Maybe a] -> [a]`)
- Definiáljunk egy függvényt ami egy lista minden elemére alkalmazza a paraméterül kapott függvényt, majd a `Nothing`-okat leszűri! (`bfm :: (a -> Maybe b) -> [a] -> [b]`)

Tesztek a működésre:

```
withFallback 1 (Just 3) == 3
withFallback 2 Nothing == 2
maybeMap (+1) (Just 2) == Just 3
maybeMap (+1) Nothing == Nothing
joinMaybe Nothing == Nothing
joinMaybe (Just Nothing) == Nothing
joinMaybe (Just (Just 1)) == Just 1
filterNothing [Just 1, Nothing, Just 3, Just 4] == [1,3,4]
take 10 (filterNothing (map Just [1..10] ++ repeat Nothing)) == [1..10]
bfm (\x -> if x > 5 then Nothing else Just (x * 2)) [1,2,3,4,5,6,7,8] == [2,4,6,8,10]
bfm (\x -> if x < 10 then Just x else Nothing) [1..20] == filter (<10) [1..20]
bfm (\x -> Just (x + 1)) [1..20] == map (+1) [1..20]
```

Magasabbrendű gyakorlás

Az alábbi feladatokhoz lehet (és néha jó ötlet is) a `Data.List` modul függvényeit használni. Ha onnan szeretnétek függvényeket használni ezt egy import állítással a fájl elején meg lehet tenni így:

```
module Homework6 where

import Data.List

-- kód ide
```

A modulban található függvényeket hoogle-ön vagy itt találhatjátok: https://hackage.haskell.org/package/base-4.19.0.0/docs/Data-List.html

- Definiáljuk a takeWhilePair függvényt amely ugyanúgy működik, mint a takeWhile, csak páronként alkalmaz egy predikátumot! (`takeWhilePair :: (a -> a -> Bool) -> [a] -> [a]`)
- Definiáljuk a dropWhilePair függvényt amely ugyanúgy működik, mint a dropWhile, csak páronként alkalmaz egy predikátumot! (`dropWhilePair :: (a -> a -> Bool) -> [a] -> [a]`)
- Definiáljuk
- Definiáljuk a `descendingSegments` függvényt amely egy listában a csökkenő szegmenseket adja vissza! (`descendingSegments :: Ord a => [a] -> [[a]]`)
- Definiáljuk az `unorderedEq` ami eldönti, hogy két lista ugyanazokat az elemeket tárolja-e pontosan ugyanannyiszor, viszont az elemek sorrendje nem számít! (`unorderedEq :: Eq a => [a] -> [a] -> Bool`)
- Definiáljuk a `minimumBy` függvény amely egy listában megkeresi a legkisebb elemet egy tulajdonság alapján! (`minimumBy :: Ord b => (a -> b) -> {- nem üres -} [a] -> a`)

- Definiáljuk a `powerSet` függvényt ami egy lista összes részlistáját visszaadja! (`powerSet :: [a] -> [[a]]`)

```
takeWhilePair (>) [10,9..1] == [10,9..1]
takeWhilePair (<) [1,2,3,4,5,4,6,7,8] == [1,2,3,4,5]
takeWhilePair (<=) [1,2,3,4,4,5,6,5] == [1,2,3,4,4,5,6]
takeWhilePair (>) [1] == [1]
dropWhilePair (>) [10,9..1] == []
dropWhilePair (<) [1,2,3,4,5,4,6,7,8] == [5,4,6,7,8]
dropWhilePair (<=) [1,2,3,4,4,5,6,5] == [6,5]
dropWhilePair (>) [1] == []
descendingSegments [10,9,8,7,8,7,6,5,4,5,6,7] == [[10,9,8,7],[8,7,6,5,4]]
descendingSegments [] == []
descendingSegments [1] == []
descendingSegments [1,2,1,3] == [[2,1]]
descendingSegments [3,3,3] == []
take 10 (descendingSegments $ cycle [3,2,1]) == replicate 10 [3,2,1]
unorderedEq [1..10] [10,9..1]
unorderedEq [1,2] [1,2]
not (unorderedEq [1,1] [1])
minimumBy (1/) [1..10] == 10
minimumBy id [1..10] == 1
minimumBy (\x -> if x == 5 then 0 else 1000000000000) [1..10] == 5
unorderedEq (powerSet [1..4])
[[],[4],[3],[3,4],[2],[2,4],[2,3],[2,3,4],[1],[1,4],[1,3],[1,3,4],[1,2],[1,2,4],[1,2,3],[1,
2,3,4]]
unorderedEq (powerSet []) [[]]
unorderedEq (powerSet [100..106])
[[],[106],[105],[105,106],[104],[104,106],[104,105],[104,105,106],[103],[103,106],[103,105]
,[103,105,106],[103,104],[103,104,106],[103,104,105],[103,104,105,106],[102],[102,106],[102
,105],[102,105,106],[102,104],[102,104,106],[102,104,105],[102,104,105,106],[102,103],[102,
103,106],[102,103,105],[102,103,105,106],[102,103,104],[102,103,104,106],[102,103,104,105],
[102,103,104,105,106],[101],[101,106],[101,105],[101,105,106],[101,104],[101,104,106],[101,
104,105],[101,104,105,106],[101,103],[101,103,106],[101,103,105],[101,103,105,106],[101,103
,104],[101,103,104,106],[101,103,104,105],[101,103,104,105,106],[101,102],[101,102,106],[10
1,102,105],[101,102,105,106],[101,102,104],[101,102,104,106],[101,102,104,105],[101,102,104
,105,106],[101,102,103],[101,102,103,106],[101,102,103,105],[101,102,103,105,106],[101,102,
103,104],[101,102,103,104,106],[101,102,103,104,105],[101,102,103,104,105,106],[100],[100,1
06],[100,105],[100,105,106],[100,104],[100,104,106],[100,104,105],[100,104,105,106],[100,10
3],[100,103,106],[100,103,105],[100,103,105,106],[100,103,104],[100,103,104,106],[100,103,1
04,105],[100,103,104,105,106],[100,102],[100,102,106],[100,102,105],[100,102,105,106],[100,
102,104],[100,102,104,106],[100,102,104,105],[100,102,104,105,106],[100,102,103],[100,102,1
03,106],[100,102,103,105],[100,102,103,105,106],[100,102,103,104],[100,102,103,104,106],[10
0,102,103,104,105],[100,102,103,104,105,106],[100,101],[100,101,106],[100,101,105],[100,101
,105,106],[100,101,104],[100,101,104,106],[100,101,104,105],[100,101,104,105,106],[100,101,
103],[100,101,103,106],[100,101,103,105],[100,101,103,105,106],[100,101,103,104],[100,101,1
03,104,106],[100,101,103,104,105],[100,101,103,104,105,106],[100,101,102],[100,101,102,106]
,[100,101,102,105],[100,101,102,105,106],[100,101,102,104],[100,101,102,104,106],[100,101,1
02,104,105],[100,101,102,104,105,106],[100,101,102,103],[100,101,102,103,106],[100,101,102,
103,105],[100,101,102,103,105,106],[100,101,102,103,104],[100,101,102,103,104,106],[100,101
,102,103,104,105],[100,101,102,103,104,105,106]]
and [length (powerSet [1..k]) == 2 ^ k | k <- [1..15]]
```