Description:

Előzetes tudnivalók

Használható segédanyagok:

- Haskell könyvtárak dokumentációja,
- Hoogle,
- a tárgy honlapja, és a
- Haskell szintaxis összefoglaló.

Ha bármilyen kérdés, észrevétel felmerül, azt a felügyelőknek kell jelezni, **nem** a diáktársaknak!

FONTOS: A megoldásban legalább az egyik (tetszőleges) függvényt rekurzívan kell megadni. Azaz a vizsga csak akkor érvényes, ha az egyik feladatot rekurzív függvénnyel adtátok meg és az helyes megoldása a feladatnak. A megoldást akkor is elfogadjuk, ha annak egy segédfüggvénye definiált rekurzívan. A könyvtári függvények (length, sum, stb.) rekurzív definíciója nem fogadható el rekurzív megoldásként.

A feladatok tetszőleges sorrendben megoldhatóak. A pontozás szabályai a következők:

- Minden teszten átmenő megoldás ér teljes pontszámot.
- Funkcionálisan hibás (valamelyik teszteseten megbukó) megoldás nem ér pontot.
- Fordítási hibás vagy hiányzó megoldás esetén a teljes megoldás 0 pontos.

Ha hiányos/hibás részek lennének a feltöltött megoldásban, azok kommentben szerepeljenek.

Tekintve, hogy a tesztesetek, bár odafigyelés mellett íródnak, nem fedik le minden esetben a függvény teljes működését, határozottan javasolt még külön próbálgatni a megoldásokat beadás előtt vagy megkérdezni a felügyelőket!

A *Visual Studio Code Haskell Syntax Highlighting* bővítmény a csatolt fájlok között megtalálható.

Telepítés:

1. Bővítmények megnyitása bal oldalt (`Ctrl + Shift + X`)
2. `...` a megnyíló ablak jobb felső sarkában
3. `Install from VSIX...` , majd a letöltött állomány kitallózása

Feladatok

Nagy-kicsi (2 pont)

Definiáljunk egy függvényt, amely eldönti egy szövegről, hogy abban rendre nagy-, és kisbetűk váltakozva szerepelnek-e!

A szövegnek nagybetűvel kell kezdődnie!

Megjegzés: Használhatjuk a `Data.Char` függvényeit.

```
upLow :: String -> Bool
```

```
upLow
upLow "A"
upLow "AlMa"
upLow "HaSkElL"
not $ upLow "a"
not $ upLow "aLmA"
not $ upLow "fagylalt"
not $ upLow "KIABALAS"
not $ upLow (repeat 'A')
```

Szöveg manipulálás

Cseréljük ki egy szövegben a paraméterként megadott karaktereket egy másik paraméterként megadott karakterre!

Véges listával (1 pont)

```
replaceGivens :: {- mit -} Char -> {- mire -} Char -> {- miben -} String
```

```
replaceGivens 'a' 'e' "" == ""
replaceGivens 'a' 'e' "alma" == "elme"
replaceGivens 'a' 'a' "alma" == "alma"
replaceGivens 'a' 'e' "nincsbenne" == "nincsbenne"
replaceGivens '?' '!' "???!!!???!!!" == "!!!!!!!!!!!!"
```

Végtelen listával (+1 pont)

Megjegzés: Amennyiben az eredeti függvényünk is megoldja ezt a feladatot, használhatjuk a következő függvénytörzset:

```
replaceGivensInf = replaceGivens
```

```
replaceGivensInf :: {- mit -} Char -> {- mire -} Char -> {- miben -} Str
```

```
take 20 (replaceGivensInf 't' 'b' (cycle "ette")) == "ebbeebbeebbeebbeebb
take 20 (replaceGivensInf 'r' 't' (cycle "haskell")) == "haskellhaskellha
```

`Maybe` csomagolás

Definiáljuk a `packMaybeTuple` függvényt, amely egy listányi `(Maybe a, Maybe b)` rendezett párból egy listányi `Maybe (a,b)` típusú értékre képez!

Ha a két `Maybe` értékből az egyik `Nothing` akkor az eredmény `Nothing` legyen, egyébként meg `Just` -ba csomagolva adjuk vissza a rendezett pár elemeit.

Véges listával (1 pont)

```
packMaybeTuple :: [(Maybe a, Maybe b)] -> [Maybe (a,b)]
```

```
packMaybeTuple [] == []
packMaybeTuple [(Just 1, Just "almafa"), (Just 2, Just "banán")] == [Just
packMaybeTuple [(Nothing, Just 0), (Just 1, Nothing), (Just 0, Just 1)] =
packMaybeTuple [(Nothing, Just (6 `div` 0)), (Just [1..], Nothing), (Just
reverse (packMaybeTuple ([(Just x, Just (x + 1)) | x <- [1..10]] ++ [(No
```

Végtelen listával (+1 pont)

használhatjuk a következő függvénytörzset:

```
packMaybeTupleInf = packMaybeTuple
```

```
packMaybeTupleInf :: [(Maybe a, Maybe b)] -> [Maybe (a,b)]
```

```
take 5 (packMaybeTupleInf [(Just i, Just 'a') | i <- [0..]]) == [Just (0
take 8 (packMaybeTupleInf ([(Just 'a', Just 5), (Just 'b', Nothing)] ++ 
take 6 (packMaybeTupleInf (cycle [(Just 'a',Nothing), (Nothing, Just "al
take 10 (packMaybeTupleInf ([(Just x, Nothing) | x <- [1..]])) == replic
```

Usz beszéd (2 pont)

Definiáljuk azt a függvényt, amely egy paraméterül kapott szöveget *usz* beszédben ad vissza! Az *usz* beszédet úgy kell képezni, hogy a szöveg minden szavát *usz*-szal egészítjük ki. A szövegről feltehető, hogy véges!

```
uszSpeech :: String -> String
```

```
uszSpeech "" == ""
uszSpeech "a" == "ausz"
uszSpeech "Szia hogy vagy" == "Sziausz hogyusz vagyusz"
uszSpeech "Avada Kedvara" == "Avadausz Kedvarausz"
uszSpeech "Expecto Patronum" == "Expectousz Patronumusz"
uszSpeech "Haskell" == "Haskellusz"
```

Páros-páratlan összeadás-kivonás (2 pont)

Adjuk össze egy számokat tartalmazó lista elemeit úgy, hogy a páratlan indexű elemeket kivonjuk, a páros indexűeket hozzáadjuk az összeghez!

Az indexelést kezdjük `0` -tól!

A listáról feltehető, hogy véges!

```
alternatingSum :: Num a => [a] -> a
```

```
alternatingSum [] == 0
alternatingSum [1] == 1
alternatingSum [1,1] == 0
alternatingSum [1,2,3] == 2
alternatingSum [1..10] == -5
alternatingSum [-10..10] == 0
```

Maybe kompozíció (2 pont)

Definiáljuk a `composeMaybe` függvényt amely `Maybe` eredményű függvényeket komponál össze!

```
composeMaybe :: (b -> Maybe c) -> (a -> Maybe b) -> (a -> Maybe c)
```

```
composeMaybe (\x -> if x == 0 then Nothing else Just $ div 3 x) (\y -> J
composeMaybe (\x -> if x == 0 then Nothing else Just $ div 3 x) (\y -> J
composeMaybe (\x -> Just $ x + 1) (\y -> if y == 0 then Nothing else Just
composeMaybe (\x -> Just $ x + 1) (\y -> if y == 0 then Nothing else Just
```

Címzés (2 pont)

Haskellben nincs beépített lehetőség arra, hogy az adatokra a memóriabeli tárolási címük alapján hivatkozzunk.

hivatkozási címmel rendelkező értékeket reprezentálja. A típus rendelkezzen egy `Null` adatkonstruktorral az üres hivatkozás reprezentálására, valamint rendelkezzen egy `Ref` adatkonstruktorral is, melynek paraméterei egy az érték hivatkozási címét tároló `Int` szám, valamint egy `t` típusú érték.

Gondoskodjunk róla, hogy a `Reference t` típusú értékek kiírathatóak legyenek, valamint hogy össze lehessen hasonlítani két értéket az `==` művelet segítségével.

Definiáljuk azt a függvényt, mely két érték hivatkozási cím szerinti egyezőségét vizsgálja. Két érték akkor egyezik meg hivatkozási cím szerint, ha hivatkozási címük megegyezik. Továbbá az üres hivatkozás kizárólag az üres hivatkozással egyezik meg.

```
refEq :: Reference t -> Reference t -> Bool
```

```
refEq (Ref 0 "alma") (Ref 0 "alma")
refEq (Ref 0 999) (Ref 0 999)
refEq (Ref 0 "alma") (Ref 0 "narancs")
refEq (Ref 110 [1..]) (Ref 110 [0])
refEq Null Null
not $ refEq (Ref 0 "alma") (Ref 1 "alma")
not $ refEq (Ref 0 "alma") (Ref 1 "narancs")
not $ refEq Null (Ref 0 "alma")
not $ refEq (Ref 0 "alma")  Null
```

Feltételes összefűzés (2 pont)

Definiáljuk az `applyWhen` függvényt, amely egy függvényt, vagyis "transzformációt", egy predikátumot és két listát kap paraméterül. A két lista elemein párhuzamosan halad és amennyiben a két elemre teljesül a megadott feltétel, úgy ezekre alkalmazza a "transzformációt". Amennyiben nem teljesül a feltétel, úgy ezeket az elemeket hagyja figyelemen kívül.

Az eredménylista hossza maximum, a paraméterül kapott listák hosszainak minimuma.

```
applyWhen :: (a -> b -> c) -> (a -> b -> Bool) -> [a] -> [b] -> [c]
```

```
applyWhen (,) (>) [1..10] [10,9..1] == [(6,5),(7,4),(8,3),(9,2),(10,1)]
applyWhen (,) (>) [] [1..] == []
applyWhen (,) (>) [1..] [] == []
applyWhen (+) (\a b -> a + b == 5) [0,1,2,3,4,5] [5,4,3,2,1,0] == replic
applyWhen (++) (\a b -> null a || null b) [[], "alma", "retek", []] ["mo
applyWhen (\_ b -> (b, not b)) (flip const) (repeat undefined) (take 100
```

Törlés szövegből (3 pont)

Definiáljuk azt a függvényt, amely vár két listát paraméterül, és a második listából törli az első listával teljesen megegyező szakaszokat! Ha két egyező szakasz egymásba lóg, akkor az elsőt kell teljes egészében törölni a listából.

```
deleteInfixes :: Eq a => [a] -> [a] -> [a]
```

```
deleteInfixes [] [] == []
deleteInfixes "valami" "" == ""
deleteInfixes "" "dandelion" == "dandelion"
deleteInfixes [2] [1,2,2,1,3,2,1] == [1,1,3,1]
deleteInfixes [1,2,3] [1..10] == [4..10]
deleteInfixes "alma" "almalmasajtalma" == "lmasajt"
deleteInfixes "alma" "almaalmasajtalma" == "sajt"
deleteInfixes [True,False] [True,True,False,True,False,False,True] == [T
take 10 (deleteInfixes [4,5] [1..]) == [1,2,3,6,7,8,9,10,11,12]
```

Tevék és kebabok (3 pont)

Definiáljuk azt a függvényt, ami egy *Camel case* konvencióban írott szöveget átír *Kebab case* konvencióra!

Példa *Camel case* konvencióra: camelCase, bufferedReader, gameOverDialogPanel
Példa *Kebab case* konvencióra: camel-case, buffered-reader, game-over-dialog-panel

A bemenetről feltehető, hogy véges és *Camel case* stílusban adott!

```
camelToKebab :: String -> String
```

```
camelToKebab "camelCase" == "camel-case"
camelToKebab "bufferedReader" == "buffered-reader"
camelToKebab "scoobyDoo" == "scooby-doo"
camelToKebab "bigRedChicken" == "big-red-chicken"
camelToKebab "algebraicDataStructure" == "algebraic-data-structure"
camelToKebab "gameOverDialogPanel" == "game-over-dialog-panel"
```