Funkcionális programozás 2022/23/1 1. gyakorlat

Elérhetőség: Erdei Zsófia TEAMS, (zsanart@inf.elte.hu)

Óra kezdés: ?

Konzultáció online formában (TEAMS): péntek 14:00-16:00

Követelmények

- Házi feladat
- Zárthelyik
- Nagybeadandó
- (Vizsga)

Kapcsolódó infó

Házi/beadandó leadó felület: TMS Feladatok, segédletek, interaktív oldal: Lambda Canvas Editorok: VS code, Notepad++

Haskell

GHC telepítés: útmutató a teams csoportban (fájlok fül)

- Deklaratív nyelv
  - A folyamatot nem mondjuk meg, csak hogy mit szeretnénk elérni
- Tisztán funkcionális nyelv
  - Imperatív nyelvek -> szekvencia, ciklus
  - Funkcionális nyelvek -> Függvények
  - Változók helyett értékek, nincs felüldefiniáló értékadás
  - "Matematikai" függvények, az output csak az inputtól függ
  - Definiáló egyenlőség, két oldal megegyezik
  - Hivatkozási helyfüggetlenség (egy kifejezés ugyanazt jelenti bárhol fordul elő)
- Statikus típusrendszer (fordítási időben típusellenőrzés)
- Lusta kiértékelés

Funkcionális programozás 2. gyakorlat

Szám típusok, típusosztályok hierarchiája

```
Num (+,-,*)
    Fractional (/)
        Double
        Float
    Integral (div, mod)
        Int
        Integer
```

Feladat: Mi lehet a típusa az alábbi kifejezéseknek?

```
- 1
- 2 * 16 - 5
- 2 * 16 - (5 :: Int)
- True + True
- 16 / 5 + 18
- div 16 2 * 5
- div 16 2 / 5
```

Számok konverziója

```
fromIntegral :: (Integral a, Num b) => a -> b
truncate :: (RealFrac a, Integral b) => a -> b
round :: (RealFrac a, Integral b) => a -> b
ceiling :: (RealFrac a, Integral b) => a -> b
floor :: (RealFrac a, Integral b) => a -> b
```

Pl.:

```
fromIntegral (10 `div` 2) * pi
(round pi) `mod` 2
```

Mintaillesztés/pattern matching

- Több függvényalternatíva
- Függvényparaméterek helyén minták
- Minta lehet:
  - Konstruktor (pl.: konkrét érték, mint: 0,1.5,42,True,False,‘a’,‘b’,“alma” stb. egyéb példa majd később)
  - “változó” (pl.: eddigi függvények esetén)
  - joker (_ karakter)

Pl.:

```
isZero :: Int -> Bool
```

```
isZero 0 = True
isZero _ = False
```

Feladat:

a) Ha az `isZero` függvényt a `10` paraméterrel hívjuk mi fog történni?
b) Mi történne, ha a fentebb definiált `isZero` függvényben felcserélnénk a két függvényalternatívát?

c) Mi történne abban az esetben, ha hozzáadnánk a következő harmadik függvényalternatívát: `isZero 10 = True`?

Definiáld a `singleDigitPrime` függvényt, amely eldönti egy paraméterként kapott egész számról, hogy egy számjegyű prím-e!

```
oneDigitPrime :: Int -> Bool
```

Feladat: Definiáld újra a logikai "és" függvényt!

```
and' :: Bool -> Bool -> Bool
```

Az alábbi tesztesetek közül mindegyiknek `True`-t kell adnia:

```
and' True  True  == True
and' True  False == False
and' False True  == False
and' False False == False
```

Feladat: Definiáld újra a logikai "vagy" függvényt!

```
or' :: Bool -> Bool -> Bool
```

Az alábbi tesztesetek közül mindegyiknek `True`-t kell adnia:

```
or' True  True  == True
or' True  False == True
or' False True  == True
or' False False == False
```

Feladat: Definiáld újra a logikai "xor" függvényt!

```
xor' :: Bool -> Bool -> Bool
```

Az alábbi tesztesetek közül mindegyiknek `True`-t kell adnia:

```
xor' True  True  == False
xor' True  False == True
xor' False True  == True
```

```
xor' False False == False
```

Feladat: Definiáljunk egy függvényt, amely egy sortörést egy szóközre cserél!

```
replaceNewline :: Char -> Char
```

Rendezett n-es/Tuple

```
- Összetett adattípus, komponensekből áll
- Bármilyen típusú értéket képes tárolni

tuple1 :: (Int, Int)
tuple1 = (2, 3)

tuple2 :: (Int, Char)
tuple2 = (2, 'c')

tuple3 :: (Int, Char, Bool)
tuple3 = (2, 'c', True)
```

Feladat: Definiáld az `isEvenTuple` függvényt, amely egy egész számot vár paraméterül és visszatérési értéke egy rendezett pár, amelynek első eleme maga a szám és második eleme `True`, ha páros és `False`, ha páratlan a szám!

```
isEvenTuple :: Int -> (Int, Bool)
```

Pl.:

```
isEvenTuple 2 == (2,True)
isEvenTuple 3 == (3,False)
```

Feladat: Definiálj függvényt, amely eldönti egy koordinátáról, hogy az origó-e!

```
isOrigo :: (Double, Double) -> Bool
```

Pl.:

```
isOrigo (0,0) == True
isOrigo (1,0) == False
```

Feladat: Definiálj függvényt, amely eldönti egy koordinátáról, hogy az x-tengelyen van-e!

```
isOnXAxis :: (Double, Double) -> Bool
```

Pl.:

```
isOnXAxis (0,0) == True
isOnXAxis (0,5) == True
isOnXAxis (1,0) == False
```

Feladat: Definiáld törtek szorzását!

```
mul :: (Int, Int) -> (Int, Int) -> (Int, Int)

mul (1, 2) (1, 2) == (1, 4)
mul (4, 3) (6, 5) == (24, 15)
```

Funkcionális programozás 3. gyakorlat

Emlékeztető: Rendezett n-es/Tuple

- Összetett adattípus, komponensekből áll
- Bármilyen típusú értéket képes tárolni
- Párokra illeszthető minta: (x, y)
- http://lambda.inf.elte.hu/Patterns.xml#mint%C3%A1k-p%C3%A1rokra-prelude.fst

Példák:

```
fst :: (a, b) -> a
fst (a, b) = a

snd :: (a, b) -> b
snd (a, b) = b
```

1. Definiáljuk a `swap` függvényt, amely megcseréli egy pár két elemét!

```
swap (1, True) == (True, 1)
swap ('a', 2.5) == (2.5, 'a')
```

2. Definiáljuk a `triplicate` függvényt, amely egy tetszőleges paraméterként adott x esetén egy rendezett hármast ad vissza, amelynek minden eleme x!

```
triplicate 0 == (0,0,0)
triplicate True == (True, True, True)
```

3. Definiáljuk az `addPair` függvényt, amely elemenként összead két számokból álló rendezett párt! Oldjuk meg a feladatot mintaillesztéssel és a `fst`, `snd` függvények segítségével is!

```
addPair (0,0) (1,2) == (1,2)
addPair (1,2) (-1,-2) == (0,0)
addPair (2,2) (4,5) == (6,7)
```

(4.) Definiáljuk a `divAndMod` függvényt, amely megadja két egész számnak az egymással való egész osztás eredményét és az osztási maradékot egy rendezett párként!

```
divAndMod 10 5 == (2,0)
divAndMod 9 5 == (1,4)
```

Listák

http://lambda.inf.elte.hu/Lists.xml

Példa konkrét listákra:

```
exampleList1 :: Num a => [a]
exampleList1 = [1,2,3,4,5]

exampleList2 :: [Bool]
exampleList2 = [True, False, False, True]

exampleList3 :: [Char] -- == String
exampleList3 = ['a', 'l', 'm', 'a']

exampleList4 :: [(Integer, Bool)]
exampleList4 = [(2, True), (-6, False)]
```

Pontpont kifejezések

```
exampleList5 = [1..10]
exampleList6 = [-10,-9..10]
exampleList7 = ['a'..'z']
```

Függvények számokat tartalmazó listákon

```
exampleSum = sum [1..10]
exampleProd = product [1..10]
```

5. Gyakorló feladatok

a) Soroljuk fel 0-tól 30-ig a számokat hármasával!
b) Soroljuk fel 10-től visszafelé -10-ig a számokat!
c) Számoljuk ki a 20 és 30 között lévő páros számok összegét!

Listagenerátorok/halmazkifejezések (list comprehension)

http://lambda.inf.elte.hu/Comprehensions.xml

Példa: páros számok négyzete az [1..10] intervallumban

```
listComp :: [Integer]
listComp = [n^2 | n <- [1..10], even n]
```

6. Gyakorló feladatok

a) Állítsuk elő a 2 hatványait növekvő sorrendben 1-től 2^10-ig!

b) Soroljuk fel az első 10 négyzetszám kétszeresét!

c) Soroljuk fel 1 és 100 között azokat a számokat, amelyek oszthatóak 3-mal, de nem oszthatóak 5-tel!

d) Soroljuk fel a 60 osztóit!

e) Állítsunk elő 1-50 között a számokat és 3-mal való osztási maradékukat egy rendezett párokat tartalmazó listában.

f) Állítsuk elő azt a listát, amely sorrendben tartalmazza az összes (óra, perc) párt!

7. Definiáljuk az `isPrime` függvényt, amely eldönti egy tetszőleges számról, hogy prím-e!

Funkcionális programozás 4. gyakorlat

Rekurzió

http://lambda.inf.elte.hu/Recursion.xml

- Egy függvény közvetlenül, vagy közvetve önmagát hívja
- Pl.:

```
fact 0 = 1
fact n = n * fact (n-1)
```

Kiértékelődés:

```
fact 3
3 * (fact 2)
3 * (2 * (fact 1))
3 * (2 * (1 * (fact 0)))
3 * (2 * (1 * (1)))
3 * (2 * 1)
3 * 2
6
```

Összegzés

Készítsünk egy függvényt, amely vár egy számot és 0..n-ig összeadja a számokat. (Először tegyük fel, hogy csak pozitív számot adhatunk meg. Később próbáljuk meg, hogy ha negatív számot kapunk, akkor hogyan alakítsuk a függvényt!)

```
sumN 0 == 0
sumN 3 == 6
sumN 8 == 36
```

Fibonacci

Definiálj függvényt, amely visszaadja az n. Fibonacci-számot!

```
fib 0 == 0
fib 1 == 1
fib 2 == 1
fib 4 == 3
fib 5 == 5
```

Minták listákra

http://lambda.inf.elte.hu/Patterns.xml#mint%C3%A1k-list%C3%A1kra

- Üres lista minta: `[]`
- Egyelemű lista: `[a]`
- Kételemű lista: `[a,b]`

- Nemüres lista minta: `a:b`

Fejelem (head) lekérése: `head [1,2,3,4] == 1`

Farok rész (tail) lekérése: `tail [1,2,3,4] == [2,3,4]`

Utolsó elem lekérése: `last [1,2,3,4] == 4`

Lista lekérése az utolsó elem nélkül: `init [1,2,3,4] == [1,2,3]`

Lista ürességének ellenőrzése: `null [] == True`

Elem hozzáfűzése egy lista elejére

```
5 : [1,2,3] == [5,1,2,3]
1 : 2 : 3 : [] == [1,2,3]
```

Definiáljuk a null' függvényt mintaillesztés segítségével, amely eldönti egy listáról, hogy üres-e!

```
null' :: [a] -> Bool
```

Definiáljuk a head' függvényt mintaillesztés segítségével, amely visszaadja egy lista első elemét!

```
head' :: [a] -> a
```

Definiáljuk az "1 elemű-e a lista" függvényt mintaillesztéssel!

```
isSingleton :: [a] -> Bool
```

Definiáljunk egy olyan függvényt, amely eldönti, hogy 0-val kezdődik-e a lista!

```
headZero :: (Num a, Eq a) => [a] -> Bool
```

Definiáljuk újra a tail függvényt, amely megadja egy lista fejelem utáni részét!

```
tail' :: [a] -> [a]
```

Definiáljuk újra a length függvényt, amely megadja egy lista hosszát!

```
length' :: [a] -> Int
```

Definiáljuk újra a sum függvényt, amely összegez egy számlistát!

```
sum' :: Num a => [a] -> a
```

Definiáljuk újra a last függvényt, amely visszaadja egy lista utolsó elemét!

```
last' :: [a]{-nemüres-} -> a
```

Definiáljuk újra az init függvényt, amely egy lista összes elemét visszaadja, az utolsót kivéve!

```
init' :: [a]{-nemüres-} -> [a]
```

Funkcionális programozás 4. gyakorlat

Emlékeztető: Minták listákra

http://lambda.inf.elte.hu/Patterns.xml#mint%C3%A1k-list%C3%A1kra

- Üres lista minta: `[]`
- Egyelemű lista: `[a]`
- Kételemű lista: `[a,b]`
- Nemüres lista minta: `a:b`

1. Definiáljuk a `duplicateElements :: [a] -> [a]` függvényt, amely előállít egy olyan listát, hogy az eredeti lista minden elemét kétszer tartalmazza!

```
duplicateElements [] == []
duplicateElements "alma" == "aallmmaa"
duplicateElements [1..10] == [1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10]
```

2. Definiáljunk egy `everySecond :: [a] -> [a]` függvényt, amely kiválogat minden második elemet egy listából!

```
everySecond [] == []
everySecond [1..10] == [2,4,6,8,10]
```

3. Definiáljuk a `concat' :: [[a]] -> [a]` függvényt, amely összefűz egy listák listáját egyetlen listává!

```
concat' [] == []
concat' [[1],[2..5],[3]] == [1,2,3,4,5,3]
concat' ["ez ", "egy ", "lista"] == "ez egy lista"
```

4. Definiáljuk a `take' :: Int -> [a] -> [a]` függvényt, amely visszaadja egy lista első n elemét és eldobja a maradékát!

```
take' 0 [1..5] == []
take' 100 [1..5] == [1,2,3,4,5]
take' 5 "Hello Haskell" == "Hello"
```

5. Definiáljuk a `drop' :: Int -> [a] -> [a]` függvényt, amely eldobja egy lista első n elemét és visszaadja a maradékát!

```
drop' 0 [1..5] == [1,2,3,4,5]
drop' 100 [1..5] == []
drop 5 "Hello Haskell" == " Haskell"
```

6. Definiáljuk az `isPrefixOf :: Eq a => [a] -> [a] -> Bool` függvényt, amely eldönti, hogy az első paraméterként megadott lista prefixe-e a második listának!

```
isPrefixOf [] [1..5] == True
isPrefixOf "Hello" "Hello Haskell" == True
isPrefixOf "Hello Haskell" "Hello" == False
```

7. Definiáljuk az `insert :: Ord a => a -> [a] -> [a]` függvényt, amely beszúr egy paraméterként megadott elemet egy rendezett listába.

```
insert 5 [] == [5]
insert 5 [1,10] == [1,5,10]
insert 5 [1,2] == [1,2,5]
take 5 (insert 0 [1..]) == [0,1,2,3,4]
```

8. Definiáljuk a `merge :: Ord a => [a] -> [a] -> [a]` függvényt, amely összefésül két rendezett listát.

```
merge [] [1..5] == [1,2,3,4,5]
merge [1..5] [] == [1,2,3,4,5]
merge [1,3..9] [2,4..10] == [1,2,3,4,5,6,7,8,9,10]
```

9. Készítsünk egy `sublist :: Int -> Int -> [a] -> [a]` függvényt, ami kivág egy listából egy részt! Az első paraméter az index, ahonnan a részlista kezdődik és a második paraméter a vágás hossza.

```
sublist 1 2 [] == []
sublist 0 5 "Hello Haskell!" == "Hello"
sublist 6 7 "Hello Haskell!" == "Haskell"
sublist 6 100 "Hello Haskell!" == "Haskell!"
sublist 100 100 "Hello Haskell!" == []
```

10. Listában, rendezett párokban tároljuk egy pizza összetevőinek nevét és árát. Adjuk meg a kész pizza árát, ha tudjuk, hogy az alapanyagokon túl 500 Ft munkadíjat számolunk fel.

```
pizza :: [(String, Int)] -> Int

pizza [("teszta", 200), ("paradicsomszosz", 150), ("pepperoni", 200), ("cheddar", 300)] == 1
pizza [("teszta", 200), ("sajtszosz", 130), ("bacon", 200), ("chili", 300), ("tukortojas", 2
pizza [("vekony teszta", 180), ("besamel", 120), ("bazsalikom", 200), ("kaviar", 2000), ("go
```

Funkcionális programozás 6. gyakorlat

Egyéb hasznos függvények

1. Egy titkosított kódból nyerjük ki az első, olyan két karakter hosszú betűsort, amit szám követ. Az egyes karakterek azonosításához használjuk a Data.Char függvényeit. (`isLetter`, `isDigit`)

   ```
   cipher :: String -> String

   cipher "PYdg7iT4vdOOn4AgmGfUpRzogAf" == "dg"
   cipher "4vkYyAO174midQTtO" == "AO"
   cipher "BwxwEwqCKHuMTAaPn" == ""
   cipher "dM7" == "dM"
   cipher "Kmz" == ""
   cipher "Zk"  == ""
   cipher "T4"  == ""
   cipher "" == ""
   ```

2. Definiáljuk a `zip' :: [a] -> [b] -> [(a,b)]` függvényt, amely két listából párok listáját állít elő!

   ```
   zip' [1..] "almafa" == [(1,'a'),(2,'l'),(3,'m'),(4,'a'),(5,'f'),(6,'a')]
   ```

3. Definiáljuk a `numberedABC :: [(Int, Char)]` függvényt, amely egy párok listája formájában visszaadja az ábécé betűit megszámozva!

4. Definiáld a `stars :: String` függvényt, amely a következő végtelen szöveget állítja elő: `"* ** *** **** *****..."`

   ```
   take 20 stars == "* ** *** **** ***** "
   ```

Lokális definíciók

- Függvények definiálása korlátozott látókörrel
- Kulcsszó: `where`
- A lokális függvények nagyobb behúzással kezdődnek

pl.:

```
f :: Int
f = g where
    g :: Int
    g = 1
```

- A `g :: Int` függvény csak az `f :: Int` definícióján belül használható

```
ghci> g
<interactive>:31:1: error: Variable not in scope: g
```

- Kicsit értelmesebb példa:

```haskell
minToHour :: Int -> (Int, Int)
minToHour x = (h, m) where
    h = x `div` 60
    m = x `mod` 60
```

1. Definiáld a `toUpperFirsts :: String -> String` függvényt, amely egy tetszőleges szöveg minden szavát átalakít nagy kezdőbetűsre!

unzip megoldás - példa

1. Definiáljuk az `unzip' :: [(a,b)] -> ([a],[b])` függvényt, amely párok listájából egy olyan lista párt állít elő, ahol az első lista a párok első elemét és a második lista a második elemét tartalmazza!

   ```haskell
   unzip' [] == ([],[])
   unzip [('a',1), ('b',2), ('c',3)] == ("abc",[1,2,3])
   ```

   Megoldás:

   ```haskell
   unzip :: [(a,b)] -> ([a],[b])
   unzip [] = ([],[])
   unzip ((x,y):xs) = (x:a, y:b) where
       (a,b) = unzip xs
   ```

Hogyan működik?

```haskell
unzip [('a',1), ('b',2), ('c',3)] = ('a':a, '1':b) where
    (a,b) = unzip [('b',2), ('c',3)]
            ------------------------

unzip [('b',2), ('c',3)] = ('b':a, 2:b) where
    (a,b) = unzip [('c',3)]
            ---------------

unzip [('c',3)] = ('c':a, 3:b) where
    (a,b) = unzip []
            --------

unzip [] = ([],[])

unzip [('c',3)] = ('c':[], 3:[])

unzip [('b',2), ('c',3)] = ('b':'c':[], 2:3:[])

unzip [('a',1), ('b',2), ('c',3)] = ('a':'b':'c':[], '1':2:3:[])
```

1. Definiáljuk az `unzip3' :: [(a, b, c)] -> ([a], [b], [c])` függvényt!

```
unzip3' [] == ([],[],[])
unzip3' [(1,2,3),(4,5,6)] == ([1,4],[2,5],[3,6])
```

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
```

Funkcionális programozás 9. gyakorlat

Magasabbrendű függvények

- Függvény a paraméter
- Függvény a visszatérési érték

Curry-zés

- Minden függvény tulajdonképpen 1 paraméteres
- Pl.:

```
max 1 5 == (max 1) 5
--          ^^^^^^^
```

1. Mi a típusa a `(max 1)` kifejezésnek?

```
add3 :: (Int -> (Int -> (Int -> (Int))))    --  :: Int -> Int -> Int -> Int
add3 a b c = a+b+c
```

1. Mi a típusa az `add3 1` kifejezésnek?

Szeletek

- Operátorokból képzett (egyparaméteres) függvények
- Pl.: `(+3)`, `(<2)`, `(*10)`
- Magasabb rendű függvények paraméterezésére használhatóak

1. Definiáld az `apply :: Num a => (a -> a) -> a -> a` függvényt, amely alkalmazza a paraméterül kapott szeletet a második paraméterére!
2. Definiáld az `applyTwice :: Num a => (a -> a) -> a -> a` függvényt, amely kétszer alkalmazza a paraméterül kapott szeletet a második paraméterére!

Elemenként feldolgozás

1. Definiáljuk újra az elemenként feldolgozás függvényét!

```
map :: (a -> b) -> [a] -> [b]

map (+10) [1..10] == [11,12,13,14,15,16,17,18,19,20]
```

Szűrés

1. Definiáljuk újra a szűrés feldolgozás függvényét!

```
filter :: (a -> Bool) -> [a] -> [a]

filter (>5) [1..10] == [6,7,8,9,10]
```

Funkcionális programozás 10. gyakorlat

Magasabbrendű függvények

1. Definiáld újra a `zipWith` függvényt, mely hasonló a `zip` hez, azonban nem párokat készít, hanem egy kétparaméteres függvényt alkalmaz!

```
zipWith min [1,9,2,5] [5,0,3,8] == [1,0,2,5]
zipWith min [1,0,3] [5,2,10,1] == [1,0,3]
zipWith (*) [2,0,6] [1,5,4,9] == [2,0,24]
```

1. A zipWith felhasználásával készítsünk egy olyan függvényt, amely számok egy sorozatából előállítja azok páronkénti különbségeinek sorozatát!

```
differences [1..5] == [1, 1, 1, 1]
differences [5,4..1] == [-1, -1, -1, -1]
differences [0,1,4,9,16] == [1, 3, 5, 7]
```

1. Definiáljuk az `isUniform :: Eq b => (a -> b) -> [a] -> Bool` függvényt, amely eldönti egy megadott függvényről, hogy egyenletes eloszlású-e, azaz hogy mindig ugyanazzal az értékkel tér-e vissza a megadott értelmezési tartományon.

```
isUniform (*0) [1..100] == True
isUniform length ["alma", "toll", "baba"] == True
isUniform (*1) ([1..100]) == False
isUniform length ["alma", "barack", "baba"] == False
isUniform id ['a'..'z'] == False
```

1. Definiáld a `selectiveApply :: (a -> a) -> (a -> Bool) -> [a] -> [a]` függvényt, amely csak azokra a listaelemekre alalmazza a paraméterül kapott függvényt, amelyekre teljesül a szintén paraméterként kapott predikátum.

```
selectiveApply (*3) (>10) [] == []
selectiveApply (*3) (>10) [1,2,3,4,5] == [1,2,3,4,5]
selectiveApply (*3) (>5) [5,10,0,-2,6,11] == [5,30,0,-2,18,33]
selectiveApply (^2) (even) [4,8,2,1,9,7] == [16,64,4,1,9,7]
```

Dollár operátor

1. Definiáljuk újra a dollár operátort!

Kompozíció

1. Definiáljuk újra a függvénykompozíciót!
2. Definiáljuk a `dropSpaces` függvényt, mely szóközöket dob el egy `String` elejéről!

```
dropSpaces " hi h i " == "hi h i "
dropSpaces "alma fa " == "alma fa "
dropSpaces "" == ""
```

1. Definiáljuk a `trim` függvényt, mely szóközöket dob el egy `String` mindkét végéről!

```
trim " hello! " == "hello!"
trim "Haskell" == "Haskell"
trim "" == ""
```

1. Definiáljuk a `monogram` függvényt, mely egy név monogramját adja meg! Használd a `words`-öt és magasabbrendű függvényt.

```
monogram "Jim Carrey" == "J. C."
monogram "Ilosvai Selymes Péter" == "I. S. P."
```

1. Definiáljuk a `uniq :: Ord a => [a] -> [a]` függvényt, mely elhagyja az ismétléseket!

A megoldáshoz érdemes használni rendezést (`sort` a `Data.List` -ből) majd a `group` függvényt (`Data.List`).

```
uniq "Mississippi" == "Mips"
uniq "papagaj" == "agjp"
uniq "" == ""
```

Névtelen függvények (lambdák)

1. Írj a következő függvényekkel ekvivalens névtelen függvényt!

```
f1 :: Int -> Bool
f1 a = even a && a `mod` 5 == 0

f2 :: Num a => a -> a -> a
f2 a b = a^2 + b^2

f3 :: (b, a) -> (a, b, a)
f3 (a,b) = (b,a,b)

f4 :: [b] -> ([b], b)
f4 (x:xs) = (xs, x)
```

1. Adott egész számok listáinak listája. Add meg azon listák listáját az `evenList :: Integral a => [[a]] -> [[a]]` függvény segítségével, amelyekben a számok összege páros!
2. Definiáljuk a fibonacci végtelen listát az `iterate` függvény segítségével! A sorozat előállítási szabálya: `(a, b) -> (b, a+b)`.

Fogalmak

- Magasabbrendű függvény
  - Fogalom + adj példát, magasabb rendű függvény-e az `elem`?
- Szeletek
  - Mik a szeletek, adj 3 példát!/ Igaz-e, hogy a szeleteket csak magasabbrendű függvények paramétereként tudjuk használni?
- Parciális alkalmazás
  - Fogalom + adj példát, magasabb rendű-e a (map (+1)) függvény
- Névtelen függvény/lambda
  - Fogalom, írj egy függvénnyel ekvivalens lambdát vagy fordítva
- Függvénykompozíció
  - Mi a különbség a `(map (+1)) . (filter (<5))` és `(filter (<5)) . (map (+1))` között?