Funkcionális programozás
5. előadás

MÁRTON Gyöngyvér

Sapientia Egyetem, Matematika-Informatika Tanszék
Marosvásárhely, Románia
mgyongyi@ms.sapientia.ro

2023, tavaszi félév

Miről volt szó?

- a Haskell kiértékelési stratégiája
- operátorok, függvények listákon: null, init, last, sum, map, filter, reverse, take, takeWhile, drop, dropWhile, elem, zip
- Haskell, a mellékhatások kezelése (side effects)
- kiíratási műveletek, a `mapM_` használata
- feladatok:
  - (mod p) szerinti hatványértékek
  - számok és a négyzetgyökök kiíratása

Miről lesz szó?

- operátorok, függvények listákon: splitAt, notElem, concat, repeat, replicate, cycle, iterate, any, all
- kiíratási műveletek, a `map`, `mapM_` használata
- Haskell projektek
- feladatok:
  - prímszámok, összetett számok szétválogatása
  - statisztikai számítások
  - magánhangzók, mássalhangzók szűrése
  - maximum számolás
  - maximum elem és pozíciói
  - Benford törvénye

Függvények listákon

A következők további listákat kezelő függvények:

`splitAt :: Int -> [a] -> ([a], [a])`
A megadott indexérték alapján a bemeneti listát két listára osztja. Az eredmény egy tuple típusú érték lesz.

```
> splitAt 4 [1,2,3,4,5,6,7,8,9]
([1,2,3,4],[5,6,7,8,9])

> splitAt 10 "Sapientia University"
("Sapientia ","University")
```

`notElem :: (Foldable t, Eq a) => a -> t a -> Bool`
Kimeneti értéke `True`, ha az első paraméterként megadott elem nincs benne a második paraméterként megadott listában, ellenkező esetben `False`.

```
> notElem 'O' "Maros-volgyi fatorzsbarlangok"
True
> notElem 'o' "Maros-volgyi fatorzsbarlangok"
False
```

Függvények listákon

```
concat :: Foldable t => t [a] -> [a]
```

A függvény sajátos esetben egy olyan listát vár, amelynek elemei szintén lista típusúak, eredményként pedig a bemeneti listákból egyetlenegy listát épít.

```
> concat ["torjai", " Budos", "-barlang"]
"torjai Budos-barlang"
> concat [[2, 4, 6], [1, 3, 5, 7, 9]]
[2,4,6,1,3,5,7,9]
```

```
repeat :: a -> [a]
```

A megadott elemmel egy végtelen listát határoz meg. Ajánlott a `take` vagy a `takeWhile` függvényekkel együtt használni.

```
> take 5 (repeat "hello")
["hello","hello","hello","hello","hello"]
```

Függvények listákon

```
replicate :: Int -> a -> [a]
```

A megadott elemmel egy `n` elemű listát határoz meg.

```
 > replicate 5 "hello"
 ["hello","hello","hello","hello","hello"]
```

```
cycle :: [a] -> [a]
```

A paraméterként megadott lista elemeit végtelenszer fűzi egymás után.

```
> take 10 $ cycle "barlang"
"barlangbar"
```

```
iterate :: (a -> a) -> a -> [a]
```

Alkalmazza a megadott függvényt, kiindulva a kezdeti értékként megadott paraméterből.

```
> take 10 (iterate (\x -> 2 * x ) 1)
[1,2,4,8,16,32,64,128,256,512]
```

Függvények listákon

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

Megvizsgálja, hogy a megadott feltételt teljesíti-e valamelyik listabeli elem.

```
> any isUpper "Sapientia University"
True
```

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

Megvizsgálja, hogy a megadott feltételt teljesíti-e minden listabeli elem.

```
> all isUpper "Sapientia University"
False
```

Feladatok

1. feladat

Írjunk két függvényt, amelyek rendre meghatározzák a magánhangzókat, illetve a mássalhangzókat egy karakterláncban.

```
import Data.List
abc = ['A'..'Z'] ++ ['a'..'z']
maganH = "aeiouAEIOU"
massalH = abc \\ maganH

maganHFilter :: [Char] -> [Char]
maganHFilter ls = filter (\c -> elem c maganH) ls

massalHFilter :: [Char] -> [Char]
massalHFilter ls = filter (flip elem massalH) ls

main = do
    let ls = "sapientia 2023, erdelyi magyar tud egyetem"
    let (mgH, msH) = (maganHFilter ls, massalHFilter ls)
    putStrLn $ "a maganhangzok: " ++ show mgH
    putStrLn $ "a massalhangzok: " ++ show msH
```

Feladatok

2. feladat

Írjunk egy függvényt, amely meghatározza a bemeneti karakterlánc *madárnyelv* változatát, ahol egy karakterlánc madárnyelv változata azt jelenti, hogy minden *m* magánhangzót kicserélünk *mpm*-re

```
auxMadarNy :: Char -> String
auxMadarNy k =
    if (elem k maganH) then [k] ++ "p" ++ [k]
    else [k]

> map auxMadarNy "Lucs-tozeglap"
["L","upu","c","s","-","t","opo","z","epe","g","l","apa","p"]

madarNy1 :: String -> String
madarNy1 ls = concat $ map auxMadarNy ls

> madarNy1 "Lucs-tozeglap"
"Lupucs-topozepeglapap"
```

Feladatok

Lehet alkalmazni az `intercalate` függvényt is:

```
> import Data.List (intercalate)
> intercalate [0,0] [[1,2,3], [4,5], [6,7,8,9]]
[1,2,3,0,0,4,5,0,0,6,7,8,9]

> ls = ["fenyokut","tozeglap","korond"]
> intercalate "-" ls
"fenyokut-tozeglap-korond"

madarNy2 :: String -> String
madarNy2 ls = intercalate "" $ map auxMadarNy ls

> madarNy2 "Lucs-tozeglap"
...
```

Feladatok

3. feladat

Írjunk egy függvényt, amely egy kételemű tuple elemtípusú lista esetében maximum értékeket számol a második elem szerepét betöltő listaelemeken.

```
lsSz = [("mari", [10, 6, 5.5, 8]), ("feri", [8.5, 9.5]),
        ("zsuzsa", [4.5, 7.9, 10]), ("levi", [8.5, 9.5, 10, 7.5])]

maxTu ls = mapM_ print $ zip nLs maxLs
    where
    nLs = [k1 | (k1, k2) <- ls]
    jLs = [k2 | (k1, k2) <- ls]
    maxLs = map maximum jLs
```

Feladatok

4. feladat

Írjunk egy Haskell-függvényt, amely meghatározza egy lista elemei közül a legnagyobbat és a legnagyobb elem listabeli pozícióit.

```
myMaximum1 :: (Num b, Enum b, Ord a) => [a] -> [(a, b)]
myMaximum1 ls = filter fg $ zip ls [0, 1..]
    where
    m = maximum ls
    fg k = fst k == m

myMaximum2 :: (Num b, Enum b, Ord a) => [a] -> (a, [b])
myMaximum2 ls = (m, map snd $ filter fg $ zip ls [0,1..])
    where
    m = maximum ls
    fg k = fst k == m

> myMaximum1 [3, 5, 6, 10, 3, 10, 8, 7, 6, 10]
[(10,3),(10,5),(10,9)]

> myMaximum2 [3, 5, 6, 10, 3, 10, 8, 7, 6, 10]
(10, [3,5,9])
```

Kiíratási műveletek

5. feladat

Írjunk Haskell-függvényt, amely kiírja külön sorokba azokat az elemeket egy (String, Int, Int) típusú elemhármasokból álló lista esetében, ahol a harmadik elem pozitív.

```
ls2 = [("Samsung", 1000, 1), ("Apple", 2000, 1), ("Huawei", 1500, -1),
            ("BlackBerry",700,-1), ("HTC",1200,1), ("Nokia",1100,1)]

myShow :: [(String, Integer, Integer)] -> String
myShow [] = ""
myShow (k : ve)
    | k3 > 0 = temp ++ myShow ve
    | otherwise = myShow ve
        where
        temp = k1 ++" "++ show k2 ++ "\n"
        (k1, k2, k3) = k
```

Figyeljük meg a két lekérdezés közötti különbséget:

```
> (putStr . myShow) ls2
> mapM_ print [(t1, t2) | (t1, t2, t3) <- ls2, t3 > 0]
```

Kiíratási műveletek

Rendezett sorrendben is kiírhatjuk az adatokat, például a harmadik elem szerinti rendezéshez felhasználjuk a `myThd` és `sortOn` függvényeket:

```
import Data.List
import Data.Ord

rendezNovekvo ls = (putStr . myShow) $ sortOn myThd ls2
    where
        myThd(t1, t2, t3) = t2

rendezCsokkeno1 ls = (putStr . myShow) $ sortOn (Down . myThd) ls2
    where
        myThd(t1, t2, t3) = t2

rendezCsokkeno2 ls = (putStr . myShow) $ sortOn myThd ls2
    where
        myThd(t1, t2, t3) = negate t2

>>> rendezNovekvo ls2
    ...
```

Kiíratási műveletek

6. feladat

Írjunk egy Haskell-függvényt, amely a bemeneti lista elemeit kétfelé válogatja, meghatározza egy listába a prímszámokat, egy másikba pedig az összetett számokat.

Feltételezzük, hogy a `primT` függvény az `eload3.hs` állományban van, és első sora:

```
module Eload3 where
```

```
import Data.List ((\\))
import Eload3 (primT)

foValogat :: (Show a, Integral a) => [a] -> IO ()
foValogat ls = do
    let rLs = valogatNr ls
    putStrLn $ "a primszamok listaja: " ++ show (fst rLs)
    putStrLn $ "az osszetett szamok listaja: " ++ show (snd rLs)

valogatNr :: (Integral a) => [a] -> ([a], [a])
valogatNr ls = (primL, osszL)
    where
    primL = [i | i <- ls, primT 3 i]
    osszL = ls \\ primL

> foValogat [24, 97, 5, 11, 74, 41, 61, 19, 100]
a primszamok listaja: [97,5,11,41,61,19]
az osszetett szamok listaja: [24,74,100]
```

Kiíratási műveletek

7. feladat

Írjunk egy Haskell-függvényt, amely egy bemeneti lista elemein statisztikai számításokat végez: rendezi a listaelemeket az előfordulási számuk szerinti sorrendbe, illetve a lexicografikus sorrend szerint.

A `csoportosit` bemeneti paramétere egy lista, egy kételemű tuple elemtípusú listát hoz létre, ahol az első érték a listaelemet, a második pedig az előfordulási számot jelöli.

```
csoportosit :: Eq a => [a] -> [(a, Int)]
csoportosit [] = []
csoportosit ls = lsK : csoportosit lsVe
    where
    y = head ls
    lsK = (y, length [x | x <- ls, x == y])
    lsVe = [x | x <- ls, x /= y]

> csoportosit "hjdgaJDGAjhgaJAgj"
[('h',2),('j',3),('d',1),('g',3),('a',2),('J',2),('D',1),('G',1),('A',2)]
```

Kiíratási műveletek

- a `statisztika` a `csoportosit` által meghatározott listát rendezi
- a könyvtárfüggvény `sortOn` első paramétere a rendezési kritérium: az `fst` vagy `snd` függvény

```
import Data.List

statisztika :: Ord a => [a] -> [(a, Int)]
--statisztika ls = sortOn snd $ csoportosit ls -- elofordulási szam
statisztika ls = sortOn fst $ csoportosit ls -- lexi sorrend

> comparing fst (3, 4) (3, 2)
EQ

> statisztika "hjdgaJDGAjhgaJAgj"
[('A',2),('D',1),('G',1),('J',2),('a',2),('d',1),('g',3),('h',2),('j',3)]
```

Kiíratási műveletek

Az eredmény elegáns kiíratását a `foStat` végzi, a `myPrint` alig tér el a korábban megadotthoz képest.

```
foStat :: (Show a, Ord a) => [a] -> IO ()
foStat ls = mapM_ myPrint (statisztika ls)
    where
    myPrint :: (Show a, Show b) => (a, b) -> IO ()
    myPrint (t1, t2) = do
        putStrLn $ show t1 ++ ": " ++ show t2
```

```
> foStat "hjdgaJDGAjhgaJAgj"
'A': 2
'D': 1
'G': 1
...
```

```
> foStat [4444, 333, 222, 333, 5555, 22, 333, 555, 111, 5555, 222, 22, 4444]
22: 2
111: 1
222: 2
...
```

Haskell-projektek

- nagyobb programok, projektek esetén a programot részekre modulokra bontjuk
- egy Haskell-projekt több állományból, azaz több független modulból áll
- egy modul első sora a `module` kulcsszót, a modul nevét és a `where` kulcsszót tartalmazza
- a modul nevét nagy kezdőbetűvel kell írni, az állomány neve pedig ugyanaz kell legyen, mint a modul neve,
- a következő sorokba kerülnek az importok és a kódsorok

8. feladat

Írjunk egy Haskell-projektet, három `hs` állományba szerkesztve, amely az alábbi számsorozatokban a számok első számjegye szerint előfordulási statisztikát készít, azaz figyeljük meg, hogy fennáll-e Benford törvénye:

- az x^i számsorozatban, ahol `i = 1,...,n`,
- az első `n` szám faktoriálisának sorozatában,
- az `n` elemű Fibonacci-sorozatban.

Haskell-projektek

Benford törvénye: vannak olyan számsorozatok, amelyekben fennáll, hogy a számok első számjegyei között az 1-es számjegy előfordulásának esélye kb. 31%, a 2-es előfordulása kb. 19%, és a százalékok a számjegyek növekedésével csökkennek.

- az első `Szamsorozatok.hs` állományba megírjuk azokat a függvényeket, amelyek meghatározzák a kért sorozatokban található számok első számjegyeit
- a második állomány a `Benford.hs`, azokat a függvényeket tartalmazza, amelyek az első számjegyek szerinti statisztikai eredményeket határozzák meg
- a harmadik állomány a `Fo.hs` lesz, amelyben a kiíratást is elvégző főfüggvény, a `foBenford`-ot tesszük

Az állományok a jegyzetben találhatóak, a főfüggvény meghívása:

```
>>> foBenford 5 10000

>>> benford $ fibL 10000
```