Funkcionális programozás
6. előadás

MÁRTON Gyöngyvér

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

2024, tavaszi félév

Miről volt szó?

- operátorok, függvények listákon: filter, reverse, take, takeWhile, drop, dropWhile, elem, zip, splitAt, notElem, concat, repeat, replicate, cycle, iterate, any, all
- Haskell, a mellékhatások kezelése (side effects)
- kiíratási műveletek, a `mapM_` használata
- feladatok:
  - magánhangzók, mássalhangzók szűrése

Miről lesz szó?

- feladatok, kiíratások listákkal
- Haskell projektek
- rendezések: `sortOn`, összefésülő rendezés (merge sort)
- feladatok:
  - madárnyelv
  - maximum elem és pozíciói
  - válogatások
  - statisztikai számítások
  - Benford törvénye

Feladatok

1. feladat

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

2. 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

3. 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

4. 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 írhatjuk az adatokat, a második elem szerinti rendezés:

```
import Data.List
import Data.Ord

myThd :: (a, b, c) -> b
myThd(t1, t2, t3) = t2

rendezNovekvo :: a -> IO ()
rendezNovekvo ls = (putStr . myShow) $ sortOn myThd ls2

rendezCsokkeno1 :: a -> IO ()
rendezCsokkeno1 ls = (putStr . myShow) $ sortOn (Down . myThd) ls2

rendezCsokkeno2 :: a -> IO ()
rendezCsokkeno2 ls = (putStr . myShow) $ sortOn (negate . myThd) ls2

>>> rendezNovekvo ls2
    ...
```

Kiíratási műveletek

5. feladat

Írjunk egy Haskell-függvényt, amely a bemeneti lista elemeit kétfelé válogatja, meghatározza egy listába a lista első eleménél nagyobb számokat egy másikba pedig a kisebbeket.

```
import Data.List
foValogat :: (Show a, Integral a) => [a] -> IO ()
foValogat ls = do
    let (r1, r2) = valogatNr ls
    putStrLn $ "a kisebb szamok listaja: " ++ show r1
    putStrLn $ "a nagyobb szamok listaja: " ++ show r2

valogatNr :: (Integral a) => [a] -> ([a], [a])
valogatNr [] = error "ures lista"
valogatNr (k : ve) = (kisebbL, nagyobbL)
    where
    kisebbL = [i | i <- ve, i < k]
    nagyobbL = ve \\ kisebbL

> foValogat [24, 97, 5, 11, 74, 41, 61, 19, 100]
a kisebb szamok listaja: [5,11,19]
a nagyobb szamok listaja: [97,74,41,61,100]
```

Kiíratási műveletek

6. 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
...
```

Kérdések

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

7. 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
```

Haskell-projektek

- általunk megírt függvényeket a Haskell-könyvtármodulokban szereplő függvényekkel együtt fogjuk használni
- probléma akkor adódik, ha egy ugyanolyan nevű függvényt írunk, mint amilyen nevű szerepel abban a Haskell-könyvtármodulban, amit importálni szeretnénk
- a következő állományba több függvényt írunk:
  - az `isDigit` eldönti egy karakterről, hogy számjegy-e vagy sem,
  - az `isAlpha` eldönti egy karakterről, hogy angol ábécébeli betű vagy sem,
  - az `ord` meghatározza egy angol ábécébeli betű ASCII kódját,
  - mindhárom ugyanilyen néven megtalálható a `Data.Char`-ban
  - a `fugvDigit1` illetve a `fugvDigit2` meghatározza egy karakterláncban a számjegy-karaktereket és a számjegy-karakterek előfordulási pozícióit,
  - a `fugvAlpha1` illetve a `fugvAlpha2` meghatározza angol ábécébeli betűket és azok ASCII kódját
  - mindkét esetben az 1 változat a könyvtár függvényekkel dolgozik, a 2 változat pedig az általunk megírtakkal

Haskell-projektek

```
isDigit :: Char -> Bool
isDigit x = x >= '0' && x <= '9'

ord :: Char -> Int
ord k = snd $ head zipAbc
    where
        abc = ['A'..'Z'] ++ ['a'..'z']
        kod = [65..90] ++ [97..122]
        zipAbc = dropWhile (aux k) $ zip abc kod
            where
                aux k (c, o) = k /= c

isAlpha :: Char -> Bool
isAlpha k = ('a' <= k && k <= 'z') || ('A' <= k && k <= 'Z')
```

Haskell-projektek

```
import qualified Data.Char as DC

fugvDigit1 :: [Char] -> [(Char, Int)]
fugvDigit1 ls = aux ls 0
    where
    aux :: [Char] -> Int -> [(Char, Int)]
    aux [] i = []
    aux (k : ve) i
        | DC.isDigit k = (k, i) : aux ve (i+1)
        | otherwise = aux ve (i+1)

fugvDigit2 :: String -> [(Char, Int)]
fugvDigit2 ls = filter aux $ zip ls [0..]
    where
        aux :: (Char, Int) -> Bool
        aux (k, i) = isDigit k
```

Haskell-projektek

```
fugvAlpha1 :: [Char] -> [(Char, Int)]
fugvAlpha1 [] = []
fugvAlpha1 (k : ve)
    | DC.isAlpha k = (k, DC.ord k) : fugvAlpha1 nVe
    | otherwise = fugvAlpha1 nVe
        where
            nVe = [x | x <- ve, x /= k]

fugvAlpha2 :: [Char] -> [(Char, Int)]
fugvAlpha2 ls = aux $ filter isAlpha ls
    where
        aux [] = []
        aux (k : ve) = (k, ord k) : aux nVe
            where
            nVe = [x | x <- ve, x /= k]
```

Haskell-projektek

```
> fugvDigit1 "Sapientia 2023 oktober 3 mvh"
[('2',10),('0',11),('2',12),('3',13),('3',23)]

> fugvAlpha2 "Sapientia 2001 oktober 3 mvh"
[('S',83),('a',97),('p',112),('i',105),('e',101)...
```

- a `Data.Char` importálása másképp történik: az `import qualified Data.Char as DC` kódsorral lehetőségünk lesz arra, hogy a `Data.Char` könyvtármodulhoz tartozó függvényekre az általunk választott `DC` névvel hivatkozhassunk
- a `DC.isDigit, DC.isAlpha, DC.ord` meghívások a `Data.Char` könyvtármodulban található függvények kiértékelését végzik
- az `isDigit, isAlpha, ord` függvény ekesetében az általunk megírt függvények kerülnek kiértékelésre
- a `Data.Char` könyvtármodul importálása a következőképpen is történhetett volna: `import Data.Char as DC`, azaz elhagyható `qualified`, ekkor a saját `isDigit, isAlpha, ord` függvényeink meghívása a `Main.isDigit`, stb. sorral oldható meg

A @ minta (as-pattern)

Egyszerű jelölést tesz lehetővé, ha egy lista egészére, illetve, ha csak egy részére (első elemére, végére) szeretnénk hivatkozni.

8. feladat

Határozzuk meg két rendezett lista összefésült értékét.

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ls = ls
merge ls [] = ls
merge ls1@(k1 : ve1) ls2@(k2 : ve2)
    | k1 < k2 = k1 : merge ve1 ls2
    | otherwise = k2 : merge ls1 ve2

> merge [1, 5, 10, 15] [2, 4, 6, 8, 20, 22, 24]
[1,2,4,5,6,8,10,15,20,22,24]
```

A függvényben a harmadik feltétel megadásakor a @ mintát használva elértük, hogy az egyenlőség jobb oldalán egyaránt tudunk hivatkozni a listák egészére, az `ls1`-re vagy az `ls2`-re, illetve a listák első elemére, a `k1`-re vagy `k2`-re és a listák végére, a `ve1`-re vagy a `ve2`-re.

Összefésülő rendezés

- az összefésülő rendezésben (merge sort) a `merge` két rendezett lista összefésülését végzi, amelyet korábban adtunk meg
- az eredeti listát két részlistára osztjuk, a `bLs` fogja tartalmazni a bemeneti lista első felét, a `jLs` pedig a második felét
- mindkét részlistát rekurzívan, a `mergeS` függvénnyel rendezzük, majd a `merge` függvénnyel összefésüljük

```
mergeS :: (Ord a) => [a] -> [a]
mergeS  [] = []
mergeS [k] = [k]
mergeS ls = merge bLista jLista
    where
    db = div (length ls) 2
    bLista = mergeS (take db ls)
    jLista = mergeS (drop db ls)

> mergeS [3,12,6,7,5,9,10,2,10,1]
[1,2,3,5,6,7,9,10,10,12]
```