Funkcionális programozás
11. előadás

MÁRTON Gyöngyvér

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

2023, tavaszi félév

Miről volt szó?

- paraméterezett típusok
- rekurzív típusok
- bináris keresőfák
  - létrehozás
  - bejárások: inorder, preorder, postorder
  - maximum elem, a bináris keresőfa mélysége
- kombinatorikai feladatok
  - `n` elem `m`-ed rendű kombinációi
  - az `n` királynő feladat

Miről lesz szó?

- kombinatorikai feladatok
  - általánosan:
    - a lexicografikus sorrend,
    - az `n` elem `m`-ed rendű kombinációi feladat,
    - az `n` királynő feladat
  - részhalmazok előállítása
  - Pascal-háromszög
- `ByteString`-ek

Kombinatorikai feladatok

- észrevehető az algoritmikai hasonlóság a 10. előadáson vett `lGen`, `komb`, illetve `kiralyno` függvények között
- ezek megadhatóak általánosan: különböző feltételeket definiáló függvényt írunk a megfelelő listák kiszűrésére, a generálást pedig a `rekurzio` fogja végezni:

```
rekurzio :: Int -> Int -> (Int -> [Int] -> Bool) -> [[Int]]
rekurzio n 0 fg = [[]]
rekurzio n m fg = [k : ve | ve <- rekurzio n (m-1) fg, k <- [1..n], fg k ve]

lGenR :: Int -> Int -> [[Int]]
lGenR n m = rekurzio n m (\k ve -> True)

> lGenR 2 3
[[1,1,1],[2,1,1],[1,2,1],[2,2,1],[1,1,2],[2,1,2],[1,2,2],[2,2,2]]

permutacioR :: Int -> [[Int]]
--permutacioR n = rekurzio n n (\k ve -> notElem k ve)
permutacioR n = rekurzio n n notElem

> permutacioR 3
[[3,2,1],[2,3,1],[3,1,2],[1,3,2],[2,1,3],[1,2,3]]
```

Kombinatorikai feladatok

```
feltKomb :: (Ord a) => a -> [a] -> Bool
feltKomb x = all (aux x)
    where
    aux x k = k > x
```

```
kombR :: Int -> Int -> [[Int]]
kombR n m = rekurzio n m feltKomb

> kombR 3 2
[[1,2],[1,3],[2,3]]
```

```
feltKir :: Int -> [Int] -> Bool
feltKir x ls = auxFeltK 1 x ls
    where
    auxFeltK :: Int -> Int -> [Int] -> Bool
    auxFeltK i x ls = all (aux x) $ zip [1..] ls
        where
        aux x (i, k) = k /= x) && (abs(x - k) /= i)
```

```
kiralynoR :: Int -> [[Int]]
kiralynoR n = rekurzio n n feltKir

> kiralynoR 6
[[5,3,1,6,4,2],[4,1,5,2,6,3],[3,6,2,5,1,4],[2,4,6,1,3,5]]
```

Adott összeg előállítása

1. feladat

Írjunk egy Haskell-függvényt, amely meghatározza, hogy hányféleképpen állítható elő egy adott `sumV` összeg az `ls` listában megadott számokból, ha mindegyik számot csak egyszer használhatjuk fel.

- a `rekurzio` függvényt használva is megoldható a feladat
- jobb hatékonyságú, ha meghatározzuk az `ls` listából előállítható részhalmazokat, kivéve az üres halmazt,
- a részhalmazokból válogatunk, a feltételnek megfelelően: kiválasztjuk azokat, amelyek elemeinek összege `sumV`

```
felSum :: Int -> [Int] -> [[Int]]
felSum sumV ls = [ kLs | kLs <- reszH ls, sum kLs == sumV ]
```

```
> felSum 30 [1, 2, 3, 5, 7, 8, 9, 10, 15]
[[1,2,3,5,9,10],[1,2,3,7,8,9],[1,2,3,9,15],[1,2,5,7,15],
...
```

Részhalmazok előállítása

A `reszH` az `1,2,3` elemekből a következőképpen állítja elő a megfelelő részhalmazokat:

- a kiindulási pont az üres lista lesz, ezt fogjuk rendre bővíteni az `1, 2, 3` elemekkel
- feltételezzük, hogy előállítottuk az `1,2` elemekből képezhető részhalmazokat:

  `[[2,1],[2],[1],[]]`
- az `1,2,3` elemekből képezhető részhalmazokat az előző lista segítségével határozzuk meg:
  - beszúrjuk a 3-as elemet minden már előállított részhalmazba (ezt végzi az `auxH` függvény):

    `[[3,2,1],[3,2],[3,1],[3]]`
  - egymásután fűzzük a két listát:

    `[[3,2,1],[3,2],[3,1],[3],[2,1],[2],[1],[]]`

Részhalmazok előállítása

```
reszH :: [Int] -> [[Int]]
reszH [] = []
reszH (k : ve) = auxH k nVe ++ nVe
    where
    nVe = reszH ve

auxH_ :: Int -> [[Int]] -> [[Int]]
auxH_ x [] = [[x]]
auxH_ x (kL : veL) = (x : kL) : auxH_ x veL

auxH :: Int -> [[Int]] -> [[Int]]
auxH x = foldr (op x) [[x]]
    where
    op x kL veL  = (x : kL) : veL

> reszH [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3]]

> auxH 3 [[2, 1], [2], [1], []]
[[3,2,1],[3,2],[3,1],[3]]
```

Adott összeg előállítása, állománykezelés

2. feladat

Az előző feladatot oldjuk meg több bemeneti értékre, ahol a bemeneti értékek a `szamok.txt` állományban vannak, sorokba tördelve. Az eredményt az `eredmeny.txt` állományba írjuk.

- a `szamok.txt` állomány több tesztesetet tartalmaz
- minden teszteset egy sorban található
- egy teszteset esetén az első szám a `sumV` összeget jelöli (ezt kell előállítani), a többi számból, amelyek szóközökkel elválasztva következnek, pedig elő kell állítani a `sumV` összeget
- példa a `szamok.txt` tartalmára:

```
30 1 2 3 5 7 9 10 15
230 2 90 10 120 15 40 20 50 70
13 4 2 7 9
```

Adott összeg előállítása, állománykezelés

```
import System.IO
import Control.Exception
import Data.List.Split

foOsszeg = do
    inf <- catch probaOpen kivetelOpen
    putStr "kimeneti allomanynev: "
    nev <- getLine
    outf <- openFile nev WriteMode
    temp <- olvasL inf
    let lsT = map auxF1 temp
    mapM_ (auxF2 outf) lsT
    hClose outf
    hClose inf

probaOpen :: IO Handle
probaOpen = do
    putStr "bemeneti allomanynev: "
    nev <- getLine
    openFile nev ReadMode

kivetelOpen :: SomeException -> IO Handle
kivetelOpen err = do
        let ls = "Hiba jellege: " ++ show err
        putStrLn ls
        probaOpen
```

Adott összeg előállítása, állománykezelés

- az `olvasL` az állománytartalom alapján létrehoz egy `[Int]` típusú elemekből álló listát: minden egyes listaelem az állomány egy adott sorában levő számokat fogja tartalmazni
- egy adott listaelemre a szükséges számítási folyamatot az `auxF1` végzi, és azért, hogy ez minden egyes listaelemre megtörténjen, a `map` paramétereként hívtuk meg, ezért az `auxF1`-be kerül meghívásra az előző feladatnál megadott `felSum`
- a `temp` feldarabolását a `wordsBy` végzi, ehhez importálni kellett a `Data.List.Split`-et
- a `Data.List.Split` használatához szükség van a `split` instalálására, amelyhez Windows esetében az admin módban elindított `PowerShell`-be be kell írni:

```
> cabal v1-install split
```

```
olvasL :: Handle -> IO [[Int]]
olvasL inf = do
    temp <- hGetContents inf
    let ls = map auxF $ wordsBy (== '\n') temp
    return ls
        where
        auxF :: String -> [Int]
        auxF ls = map (read :: String -> Int) $ words ls
```

Adott összeg előállítása, állománykezelés

- az auxF1-be kerül meghívásra az előző feladatnál megadott `felSum`
- az `eredmeny.txt` állományba való formázott kiíratásért az auxF2 felelős, és hogy ez minden listaelemre megtörténjen, a `mapM_`-nek adtuk át a függvényt paraméterként

```
auxF1 :: [Int] -> (Int, [Int], [[Int]])
auxF1 kLs = (sumV, ls, felSum sumV ls)
    where
    sumV = head kLs
    ls = tail kLs

auxF2 :: Handle -> (Int, [Int], [[Int]])-> IO ()
auxF2 outf kLs = do
    hPutStrLn outf $ show k1 ++ " " ++ show k2 ++ "\n" ++ show k3
    hPutStrLn outf ""
        where
        (k1, k2, k3) = kLs
```

A Pascal-háromszög

3. feladat

Írjunk egy Haskell függvényt, amely kiírja a Pascal háromszög első `n` sorát egy szövegállományba, háromszög formában.

- a Pascal-háromszög a binomiális együtthatók háromszög formában való rendezését jelenti
- a binomiális együttható azt a pozitív egész számot jelenti, amely az `(1 + x)`$^{\texttt{n}}$ polinom `x`$^{\texttt{k}}$ tagjának az együtthatója, amely ugyanakkor egyenlő lesz `n` elem `k`-ad rendű kombinációinak a számával
- a Pascal-háromszög kiíratása során az `n` értéke a sor, míg a `k` értéke az oszlop értékét jelöli, és fennáll: $\texttt{n} \geq \texttt{k} \geq 0$
- a binomiális együtthatók értékét direkt módon a következő képlettel is meg tudjuk határozni:

$$\binom{\texttt{n}}{\texttt{k}} = \frac{\texttt{n!}}{\texttt{k!} \cdot \texttt{(n-k)!}}$$

- a binomiális együtthatók értékét azonban meg lehet határozni úgy is, hogy az `(n-1)`-edik sorban levő értékekből generáljuk az `n`-edik sorban levő értékeket, ahol a nulladik sorban az `1` érték található, amely $\binom{0}{0}$ értéknek felel meg

A Pascal-háromszög

- a `pascal` függvény az előző oldal utolsó pontjában megadott a gondolatmenetet követi
- a nulladik sort egy egyelemű listaként, az `1`-es értékkel definiáljuk
- a következőt, azaz az első sort a nulladik, a második sort az első stb. alapján fogjuk meghatározni
- az új sor első elemeként mindig az `1`-es értéket rögzítjük, majd az előző sor elemei alapján a többi elemet az `auxPascal` függvénnyel határozzuk meg:
  - az előző sorban (listában) levő egymás melletti két elemet összeadjuk, majd a kapott értékekből felépítjük az új listát, azaz az új sor elemeit
  - az előző sor (lista) utolsó elemét pedig változatlanul áttesszük az új listába
- példa, az első négy sor meghatározására:

```
[1] → 1 :[1] → [1,1]
[1,1] → 1 : [1+1, 1] → [1,2,1]
[1,2,1] → 1 : [1+2, 2+1, 1] → [1,3,3,1]
[1,3,3,1] → 1 : [1+3, 3+3, 3+1, 1] → [1,4,6,4,1]
```

A Pascal-háromszög

```
auxPascal :: Integral a => [a] -> [a]
auxPascal [k] = [k]
auxPascal (k1 : k2 : ve) = (k1 + k2) : auxPascal (k2 : ve)

pascal :: Integral a => Int -> [a]
pascal 0 = [1]
pascal n = 1 : auxPascal (pascal (n-1))

> pascal 5
[1,5,10,10,5,1]

> auxPascal [1, 5, 10, 10, 5, 1]
[6,15,20,15,6,1]
```

A Pascal-háromszög

- a Pascal-háromszög első `n` sorának a generálását a `pascalN` fogja végezni, ez a korábbi `pascal` függvény módosított változata lesz
- a generált listákat egymás után fűzzük a `++` operátorral, az `auxPascal` függvény bemenetének pedig az eredménylistához utolsónak hozzáfűzött listát adjuk meg

```
pascalN :: Integral a => Int -> [[a]]
pascalN 0 = [[1]]
pascalN n = temp ++ [1 : auxPascal (last temp)]
    where
    temp = pascalN (n - 1)

> pascalN 4
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
```

A Pascal-háromszög

- ahhoz, hogy háromszög formában tudjuk megjeleníteni a számokat, a számok közé egy-egy szóközt, illetve minden egyes sor elé, bizonyos számú szóközt kell tenni
- a sorok elejére kerülő szóközök számát a `db` változó jelöli, és a `szokozok` függvényben kerül pontos meghatározásra
- a `db` érték attól függ, hogy hány számjegy, illetve szóköz található a Pascal-háromszög legutolsó, illetve aktuális sorában

```
szokozok :: String -> Int -> String
szokozok ls i = replicate db ' '
    where
    db = div (i - length ls) 2
```

A Pascal-háromszög

- a Pascal-háromszög legutolsó sorában található szóközök és számjegyek számát az `i` jelöli, és a `printPascal`-ban, a feladat *főfüggvényében* kerül meghatározásra
- a `listaToStr` függvény első paramétere ez az `i` érték lesz, második paramétere pedig a Pascal-háromszög soraiban levő számok lesznek
- a `listaToStr` az `auxSor`-t alkalmazva, betesz az aktuális sor elejére `db` darab szóközt, majd az aktuális sorban levő további számokat az `auxSz`-et meghívva, szóközökkel elválasztva összefűzi

```
listaToStr :: Show a => Int -> [[a]] -> [String]
listaToStr i = map (auxF i)

auxF :: Show a => Int -> [a] -> String
auxF i k = auxSor i k ++  auxSz k

auxSor :: Show a => Int -> [a] -> String
auxSor i k = szokozok (unwords $ map show k) i

auxSz :: Show a => [a] -> String
auxSz [] = ""
auxSz (k : ve)= show k ++ " " ++ auxSz ve
```

A Pascal-háromszög

Az állományba való kiíratást a `printPascal` végzi, ez lesz a feladat főfüggvénye, amelynek paraméterként meg kell adni az állomány nevét, illetve hogy hány sort kell meghatározni a Pascal-háromszögből:

```
printPascal :: String -> Int -> IO ()
printPascal nev n = do
    let pLs = pascalN n
    let i = length $ unwords $ map show $ last pLs
    let tLs = listaToStr i pLs
    outf <- openFile nev WriteMode
    mapM_ (hPutStrLn outf) tLs
    hClose outf

 > printPascal "pascal30.txt" 30
```

ByteString-ek

- a `ByteString`-ek az állományok egy hatékonyabb feldolgozási módját teszik lehetővé
- a Haskell megkülönböztet lassú (lazy) és szigorú (strict) `ByteString`-eket
- a `Data.ByteString` könyvtármodulban vannak a szigorú feldolgozási mód szerint működő függvények, ahol minden karakter 8 biten van tárolva
- a `Data.ByteString.Lazy`, illetve a `Data.ByteString.Lazy.Char8` könyvtármodulban pedig a lusta kiértékelési stratégia szerint működő függvények vannak, ahol egy `ByteString` elemei úgynevezett `chunk`-okban, azaz nagyobb darabokban vannak tárolva, amelyek mérete maximum 64 kilobájt
- ezekben számos olyan függvény van, ugyanolyan néven, amelyek ugyanazt végzik, mint a `Prelude`-ben, a `Data.List`-ben vagy a `Data.Char`-ban stb.-ben található függvények, például: `readFile, lines, words, intercalate`
- ahhoz, hogy különbséget tegyünk az ugyanolyan nevű, de különböző könyvtármodulokban elhelyezkedő függvények az `L8` névválasztással jelezni fogjuk, hogy mikor használjuk a `Data.ByteString.Lazy.Char` könyvtármodul függvényeit

ByteString-ek

- `ByteString` típusú adatok használatakor legtöbbször szükség van `String` és `ByteString` típusok közötti átalakításra, amelyeket a `pack`, illetve `unpack` végeznek
- példa:

```
> import qualified Data.ByteString.Lazy.Char8 as L8
> ls = "Borgoi-hago Torcsvari-hago Gyimesi-hago"
> lsB = L8.pack ls

> lsBL = L8.words lsB
> lsBL
["Borgoi-hago","Torcsvari-hago","Gyimesi-hago"]

> lsB1 = L8.intercalate (L8.pack "#") lsBL
> lsB1
"Borgoi-hago#Torcsvari-hago#Gyimesi-hago"

> ls1 = L8.unpack lsB1
> ls1
"Borgoi-hago#Torcsvari-hago#Gyimesi-hago"
```

- érdemes megvizsgálni a különböző adatok típusát is, ezért próbáljuk ki a következőket: `:t ls`, `:t lsB`, `:t lsBL`, `:t lsB1`, `:t ls1`.

ByteString-ek

- általában a Haskellben a "" közötti karakterszekvenciáknak `String` típusa van. Ha azonban engedélyezve van az `OverloadedStrings` nyelvi kiterjesztés, akkor a Haskell megengedi, hogy a `ByteStringek` megadásakor is használhassuk a "" jelet, ilyenkor fölösleges a `pack`, `unpack` használata
- más programozási nyelvekhez hasonlóan a GHC Haskell is több pragmával, a fordító felé irányuló utasítással rendelkezik, amelyek segítségével befolyásolható a kód hatékonysága
- a pragmákat `{-# pragma_nev...#-}` közé kell írni, és az állomány első sorában kell feltüntetni
- a `LANGUAGE` pragma használatával nyelvi kiterjesztéseket lehet engedélyezni, alkalmazásával a következőkben az `OverloadedStrings` használatát tesszük lehetővé

ByteString-ek

- a `foBS`, illetve `foBS_` függvényekben a szóközöket #-re helyettesítjük
- a két függvényhívás ugyanazt eredményezi, a feldolgozásra kerülő karakterláncok azonban különböző típusúak:

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString.Lazy.Char8 as L8

foBS :: IO ()
foBS = do
    let lsBL = L8.words lsB
    print lsBL
    let lsB1 = L8.intercalate "#" lsBL
    print lsB1
        where
        lsB :: L8.ByteString
        lsB = "Borgoi-hago Torcsvari-hago Gyimesi-hago"

foBS_ :: IO ()
foBS_ = do
    let lsBL = L8.words $ L8.pack ls
    print lsBL
    let lsB1 = L8.intercalate "#" lsBL
    print lsB1
        where
        ls :: String
        ls = "Borgoi-hago Torcsvari-hago Gyimesi-hago"
```

ByteString-ek

4. feladat

A `film.txt` szövegállományban egy adott filmről `9` típusú adat van eltárolva: megjelenési év, filmcím, a film hossza, a film típusa, népszerűségi index, díjazott-e a film, a főszerepet játszó színész, a színésznő és a rendező. Írjunk egy Haskell-programot, amely meghatározza az `fEv`-ben készült filmek listáját, ahol az `fEv` értékét a billentyűzetről olvassuk be.

- a `film.txt`-ben a filmekre vonatkozó adatok sorokba vannak tördelve
- egy sorban, a különböző típusú adatok között tabulátor jel van és `Unknown` jelenik meg, ha valamely adatra vonatkozóan nincs meghatározott érték
- egy ilyen szerkezetű állomány letölthető a következő linkről:
  https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/film.txt
- az állomány tartalmát a `readFile` függvénnyel fogjuk beolvasni, majd a soronkénti feldolgozását a `lines` függvénnyel fogjuk végezni

ByteString-ek

- az `fLs` a kiválogatott filmcímeket tartalmazza,
- az `intercalate` segítségével `\n` jeleket fűzünk a filmcímek, azaz a listaelemek közé, és az így kapott `ByteString` típusú értéket írjuk ki a `writeFile` függvény segítségével a `filmekL.txt` állományba

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString.Lazy.Char8 as L8

foFilm1 :: IO ()
foFilm1 = do
    putStr "ev: "
    tStr <- getLine
    let fEv = read tStr :: Int
    temp <- L8.readFile "film.txt"
    let tLs = map (strProcBS fEv) $ L8.lines temp
    let fLs = filter auxFilter tLs
    let rLs = L8.intercalate "\n" $ map auxMap fLs
    L8.writeFile "filmekL.txt" rLs
        where
        auxMap (Just k) = k
        auxFilter val = case val of
            Just k -> True
            Nothing -> False
```

ByteString-ek

- a `strProcBS`-ben történik az aktuális sor feldarabolása
- a `split` az első paramétere mentén végzi a feldarabolást, ami jelen esetben egy tabulátor
- a feldarabolt `ByteString` elemtípusú listából a nulladik elem fogja tartalmazni a film megjelenési évét, a következő a film címét, majd a film hossza következik, és így tovább, aszerint, ahogy azt korábban leírtuk

```
strProcBS :: Int -> L8.ByteString -> Maybe L8.ByteString
strProcBS fEv iLs = res
    where
    ls = L8.split '\t' iLs
    temp = L8.readInt (ls !! 0)
    Just (kEv, tLs) = temp
    kCim = ls !! 1
    res = if kEv == fEv then Just kCim else Nothing
```

ByteString-ek

- a nulladik elemet `readInt` függvénnnyel `Int` típusú értékké alakítjuk
- a `readInt` a bemeneti `ByteString`-ben levő prefix számjegyekből létrehoz egy egész számot, majd az első olyan karaktertől kezdve, ahol már nem számjegyek vannak, létrehoz egy `ByteString` típusú adatot
- az egész szám a kimeneti tuple első eleme, míg a `ByteString` típusú adat a tuple második eleme lesz
- példa:

```
> import qualified Data.ByteString.Lazy.Char8 as L8
> ls = "1802 Kolozsvar december 15"
> L8.readInt $ L8.pack ls
Just (1802," Kolozsvar december 15")
```

ByteString-ek

A feladat következő implementációjában két módosítást végzünk:

- a keresett évszámot nem alakítjuk át `Int`-té, sem a beolvasásnál, sem az összehasonlításnál, a film megjelenési évét is `String` típusú adatként kezeljük
- a `filmekL.txt`-be soronként írunk a `mapM_` függvény segítségével:

```
import System.IO
import qualified Data.ByteString.Lazy.Char8 as L8

foFilm2 :: IO ()
foFilm2 = do
    putStr "ev: "
    fEv <- getLine
    temp <- L8.readFile "film.txt"
    let tLs = map (strProcBS2 fEv) $ L8.lines temp
    let fLs = filter auxFilter tLs
    outf <- openFile "filmekL.txt" WriteMode
    mapM_ (auxMap outf) fLs
    hClose outf
        where
        auxMap outf (Just k) = hPutStrLn outf (L8.unpack k)
        auxFilter val = case val of
            Just k -> True
            Nothing -> False
```

ByteString-ek

```
strProcBS2 :: String -> L8.ByteString -> Maybe L8.ByteString
strProcBS2 fEv iLs = res
    where
    ls = L8.split '\t' iLs
    kEv = L8.unpack $ ls !! 0
    kCim = ls !! 1
    res = if kEv == fEv then Just kCim else Nothing
```

ByteString-ek

A következő kódsor egy harmadik megoldást mutat

- a filmcímek kiválasztásához előbb kiválasztjuk az állomány azon sorait, ahol a megjelenési év megegyezik a keresett évvel,
- a filmcímek képernyőre való kiíratását a `myPutStr`-el végezzük
- az aktuális sor feldarabolását a korábban megadott `strProcBS` végezzük

```
import System.IO
import qualified Data.ByteString.Lazy.Char8 as L8

foFilm3 :: IO ()
foFilm3 = do
    putStr "ev: "
    tStr <- getLine
    let fEv = read tStr :: Int
    temp <- L8.readFile "film.txt"
    let fLs = filter (auxFilter . strProcBS fEv) $ L8.lines temp
    mapM_ (myPutStr . L8.unpack) fLs
       where
       auxFilter val = case val of
          Just k -> True
          Nothing -> False

myPutStr :: String -> IO()
myPutStr str = putStrLn kCim
    where
    ls = splitOn "\t" str
    kCim = ls !! 1
```

ByteString-ek

5. feladat

Írjunk egy Haskell-programot, amely ábécésorrendbe rendezi az előző feladatban is használt `film.txt` állományban megjelenő rendezőket, majd kiírja az így kapott adatokat a `rendezoL.txt` állományba.

```
...
import Data.List (sort)
foRendezo :: IO ()
foRendezo = do
    temp <- L8.readFile "film.txt"
    let tLs = halmazL $ (map strProcBS3 . L8.lines) temp
    let rLs = sort tLs
    L8.writeFile "rendezoL.txt" $ L8.intercalate "\n" rLs
```

- az állomány egy adott sorát az `strProcBS3`-ben dolgozzuk fel
- a kapott lista elemeit rendezzük, majd az `intercalate` segítségével az elemek közé `\n`-t szúrunk, és ezt írjuk ki a `writeFile`-lal az állományba
- a `halmazL`-ben szűrjük ki a többször előforduló rendezőket, mivel egy adott rendező több filmet is rendezhet

ByteString-ek

```
strProcBS3 :: L8.ByteString -> Maybe L8.ByteString
strProcBS3 iLs = res
    where
    ls = L8.split '\t' iLs
    kRendezo = ls !! 8
    res = if kRendezo /= L8.pack "Unknown" then Just kRendezo else Nothing

halmazL :: Eq a => [Maybe a] ->[a]
halmazL [] = []
halmazL (val: ve) = case val of
    Just k -> k : halmazL [x | x <- ve, Just k /= x]
    Nothing -> halmazL ve
```

- `strProcBS3`-ben először tördeljük a sort a tabulátorok mentén, majd kiválasztjuk a rendezőre vonatkozó értéket, ha ez az érték `Unknown`, akkor `Nothing` lesz az eredmény
- a `halmazL` kimenete egy tetszőleges lista, amely csak a rendezők neveit fogja tartalmazni