Funkcionális programozás
11. előadás

MÁRTON Gyöngyvér

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

2025, tavaszi félév

Miről volt szó?

- Haskell I/O műveletek, állománykezelés:
  `hGetContents, writeFile, readFile`
- Eratosztenész szitája
- Haskell I/O műveletek, bináris állományok, feladatok
  - bináris állomány hexa alakja
  - állomány mérte, bájtban
  - állomány tartalmának titkosítása (xor)
- hibakezelés: `error, catch`
- típusok és adatszerkezetek: rekord típusok

Miről lesz szó?

- típusok és adatszerkezetek: rekord típusok
- ByteString-ek
- Algebrai adattípusok

Rekord típusok, szövegállományok

1. feladat

Egy cég egy adott alkalmazottról a következő adatokat tárolta el: név, év-jövedelem értékpárok. Írjunk egy Haskell-függvényt, amely egy megadott évre meghatározza minden alkalmazott jövedelmét.

```
type Jovedelem = (Int, Int)
data Alkalmazott = Alkalmazott {
    alkNev :: String,
    alkJovedelem :: [Jovedelem]
    } deriving (Show, Read)
```

Feltételezzük, hogy a cég adatai az `alkalmazottData.txt` állományban vannak, pl:

```
[ Alkalmazott {alkNev = "KissCs",
  alkJovedelem = [(2012,8600),(2013,8900),(2014,8700)]},
  Alkalmazott {alkNev = "SzaboJ" ,
  alkJovedelem = [(2010,18000),(2011,21000),(2013,20000),(2014,24000)]},
  Alkalmazott {alkNev = "NagyS",
  alkJovedelem = [(2011,22000),(2012,23000),(2013,19000),(2014,24000)]},
  Alkalmazott {alkNev = "KovacsM",
  alkJovedelem = [(2013,9900),(2014,9800)]}]
```

Rekord típusok, szövegállományok

Az állomány szerkezete lehetővé teszi az egyszerű adatkiolvasást: a `temp`-be kerülő `String` típusú adatot egyetlen `read` segítségével át tudjuk alakítani `[Alkalmazott]` típusú értékké:

```
mainAlkalmazott :: IO ()
mainAlkalmazott = do
    temp <- readFile "alkalmazottData.txt"
    let lsAlk = (read :: String -> [Alkalmazott]) temp
    putStr "ev: "
    temp <- getLine
    let ev = read temp :: Int
    foJovedelem ev lsAlk

> mainAlkalmazott
ev: 2012
KissCs, 8600
SzaboJ, nincs jovedelem
...
```

Rekord típusok, szövegállományok

További két függvényt írtunk:

- az `auxEv` ha lehetséges, akkor meghatározza az évre vonatkozó jövedelmet, ellenkező esetben `Nothing` lesz a kimeneti értéke
- a `foJovedelem`-ben minden egyes alkalmazott esetében meghívásra kerül az `auxEv` függvény, amelynek az eredményét egy `case`-ben elemezzük

```
auxEv :: Int -> [Jovedelem] -> Maybe Int
auxEv ev = foldr (op ev) Nothing
    where
    op ev (k1, k2) res =
        if ev == k1 then Just k2 else res

> auxEv 2010 [(2012,8600),(2013,8900),(2014,8700)]
Nothing
```

Rekord típusok, szövegállományok

```
foJovedelem :: Int -> [Alkalmazott] -> IO()
foJovedelem ev = mapM_ (auxF ev)
    where
    auxF :: Int -> Alkalmazott -> IO()
    auxF ev k =
        case res of
            Just x -> putStrLn $ alkNev k ++ ", " ++ show x
            Nothing -> putStrLn $ alkNev k ++ ", nincs jovedelem"
        where
        res = auxEv ev (alkJovedelem k)

> foJovedelem 2010 lsAlk
KissCs, nincs jovedelem
SzaboJ, 18000
...
```

Rekord típusok, szövegállományok

2. feladat

Egy cég egy adott alkalmazottról a következő adatokat tárolta el: név, év-jövedelem értékpárok. Írjunk egy Haskell-függvényt, amely meghatározza egy megadott évre a maximális jövedelmet és azon alkalmazottakat, akiknek maximális volt a jövedelme.

- a feladat megoldásához a korábban használt típusszinonimaként definiált `Jovedelem` típust fogjuk használni, illetve az `auxEv` függvényt
- feltételezve, hogy ezek az `feladatJov.hs`-ben vannak megadva, módosítsuk az `feladatJov.hs` tartalmát, az első sorba írjuk be: `module FeladatJov where`
- ezután a feladathoz tartozó definíciókat, illetve függvényeket írjuk egy másik fileba, amelynek első sorába írjuk: `import qualified FeladatJov as FJ`
- ekkor megadható a következő típus, amelyet a feladat adatainak a feldolgozásakor fogunk használni
- vegyük észre hogy ez más mint ahogy az `feladatJov.hs`-ben definiáltuk az Alkalmazott típust:

```
data Alkalmazott = Alkalmazott String [FJ.Jovedelem]
    deriving (Show, Read)
```

Rekord típusok, szövegállományok

- feltételezzük, hogy a cég adatai a következő formában, az `alkalmazott.txt` állományban vannak, pl:

```
KissCs [(2012,8600),(2013,8900),(2014,8700)]
SzaboJ [(2010,18000),(2011,21000),(2013,20000),(2014,24000)]
NagyS [(2011,22000),(2012,23000),(2013,19000),(2014,24000)]
KovacsM [(2013,9900),(2014,9800)]
```

- vegyük észre, hogy a korábbi `alkalmazottData.txt` állomány szerkezete bonyolultabb volt:

```
[ Alkalmazott {alkNev = "KissCs",
  alkJovedelem = [(2012,8600),(2013,8900),(2014,8700)]},
  ...
```

Rekord típusok, szövegállományok

A feladat főfüggvénye a következő:

```
mainAlkalmazott :: IO ()
mainAlkalmazott = do
    lsAlk <- myReadFile "alkalmazott.txt"
    --print lsAlk
    putStr "ev: "
    temp <- getLine
    let ev = read temp :: Int
    foMaxJovedelem ev lsAlk
```

A `myReadFile` az adatbeolvasást végzi, a `foMaxJovedelem` pedig megvalósítja maximum keresést, illetve az eredményeket kiíratását.

```
> mainAlkalmazott
ev: 2014
maximalis jovedelem: 24000
SzaboJ
NagyS
```

Rekord típusok, szövegállományok

- a `myReadFile` soronként fogja, a `lines` függvényt alkalmazva feldolgozni a `temp`-el jelölt állománytartalmat
- a sorokat a `words`-al bontjuk szavakra

```
myReadFile :: FilePath -> IO [Alkalmazott]
myReadFile nev = do
    temp <- readFile nev
    let ls = map auxF $ lines temp
    return ls
        where
            auxF ls = Alkalmazott nevA jovA
                where
                [nevA, tJovA] = words ls
                jovA = (read :: String -> [FJ.Jovedelem]) tJovA
```

Rekord típusok, szövegállományok

```
foMaxJovedelem :: Int -> [Alkalmazott] -> IO()
foMaxJovedelem ev ls = do
    let (mLs, max) = maximumAlk ev ls
    case max of
        -1 -> putStrLn "nincs jovedelem"
        _  -> do
          putStrLn $ "maximalis jovedelem: " ++ show max
          mapM_ putStrLn mLs
```

A `maximumAlk` kimenete egy tuple típusú érték, a függvénytörzs a következő oldalon lesz:

- az `mLs`-ben azoknak az alkalmazottaknak a nevei kerülenk, akiknek az adott évben maximális volt a jövedelme
- a `max` a maximális jövedelmet jelöli, amely `-1` lesz, ha az adott évben senkinek sincs jövedelem nyilván tartva
- a maximum keresés algoritmusát korábbi előadáson már tárgyaltuk, a különbség a `maximumAlk`-ban az, hogy a lista elemei, most `Alkalmazott` típusúak

Rekord típusok, maximum keresés

```
maximumAlk :: Int -> [Alkalmazott] -> ([String], Int)
maximumAlk ev ls = (mLs, max)
    where
    (mLs, max) = foldr op res ls
    res = ([], -1)
    op :: Alkalmazott -> ([String], Int) -> ([String], Int)
    op kAlk t
        | k == m = (nevA : nevLs, m)
        | k < m = t
        | k > m = ([nevA], k)
            where
            (nevLs, m) = t
            Alkalmazott nevA jovA = kAlk
            temp = FJ.auxEv ev jovA
            k = case temp of
                    Nothing -> -1
                    Just x -> x
```

Egy alkalmazott adott évbeli jövedelemértékének a kiválasztását az `feladatJov.hs` található `auxEv` függvény segítségével végeztük.

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

3. 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

4. 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

Algebrai adattípusok

- több értékkonstruktor is lehetséges
- a legismertebb algebrai adattípus a `Bool`, amely két értékkonstruktorral rendelkezik, a `True`-val és a `False`-szal:

```
data Bool = False | True
```

- az értékkonstruktorokat `|`-al kell elválasztani
- algebrai adattípust mi is definiálhatunk, például létrehozhatunk egy saját `Bool` típust, amelyet egy természetes szám párosságának a vizsgálatához ugyanúgy lehet használni, mint `Bool`-t:

```
data MyBool = Igaz | Hamis
   deriving (Show)

parosT :: Integral a => a -> MyBool
parosT n = if even n then Igaz else Hamis
```

Algebrai adattípusok

- a következő példában a `Szemely` típusnak két értékkonstruktora lesz, az első, a `Diak` három, míg a második, a `Tanár` két mezővel rendelkezik
- az értékkonstruktorokat írhatjuk egymás mellé, de tördelve, külön sorba is meg lehet adni őket

```
data Szemely = Diak String String Double
             | Tanar String String
   deriving (Show)
```

- egy `Szemely` elemtípusú lista kezdőértéke a következő lehet:

```
lsSz :: [Szemely]
lsSz = [Diak "Laci" "ELTE" 4.5, Tanar "Feri" "ELTE",
   Tanar "Mari" "Sapientia", Diak "Lori" "Sapientia" 7.5,
   Diak "Sari" "Sapientia" 8.75, Tanar "Zsuzsi" "ELTE"]
```

Algebrai adattípusok

5. feladat

Írjunk egy Haskell függvényt, amely egy `Szemely` típusú lista esetében meghatározza a 4.5-nél nagyobb jeggyel rendelkező diákok számát:

```
szamolDiak :: [Szemely] -> Int
szamolDiak ls = length $ filter auxF ls
   where
       auxF (Tanar _ _) = False
       auxF (Diak _ _ jegy) = jegy > 4.5

> szamolDiak lsSz
2
```

Mintaillesztéssel választottuk külön a `Tanar`, illetve a `Diak` értékekkel való műveletvégzést.

Algebrai adattípusok

A feladat megoldható a `foldl'` függvény használatával is, a kódsor a következő:

```
import Data.List (foldl')
szamolDiak_ :: [Szemely] -> Int
szamolDiak_ = foldl' op 0
   where
   op res k = case k of
       Tanar _ _ -> res
       Diak _ _ jegy -> if jegy > 4.5 then 1 + res
                        else res
```

Algebrai adattípusok

A következőkben előbb négy típusszinonimát definiálunk, majd egy `BankSzamla` típust, három értékkonstruktorral, amelyekben használjuk a típusszinonimaként megadott típusokat:

```
type KartyaSz = String
type Tulajdonos = String
type Cim = [String]
type FelhasznaloID = Int

data BankSzamla = BankKartya KartyaSz Tulajdonos Cim
                | Keszpenz
                | Szamla FelhasznaloID
                deriving (Show, Eq)
```

Konstans értékeket a következőképpen is létrehozhatunk:

```
sz1 = BankKartya "12321" "Kiss Antal" ["Mvh", "Romania"]
sz2 = Keszpenz
sz3 = Szamla 12
sz4 = BankKartya "54321" "Nagy Antal" ["Kv", "Romania"]
sz5 = BankKartya "98765" "Beres Antal" ["Mvh", "Romania"]
sz6 = Szamla 13

lsBsz :: [BankSzamla]
lsBsz = [sz1, sz2, sz3, sz4, sz5, sz6]
```

Algebrai adattípusok

6. feladat

Írjunk egy Haskell-függvényt, amely kiválogatja egy `BankSzamla` elemtípusú listából azokat a személyeket, akiknek értékkonstruktora `BankKartya`.

A feladatot háromféleképpen is megoldjuk. A `valogatSz` explicit rekurziót használ:

```
valogatSz :: [BankSzamla] -> [Tulajdonos]
valogatSz [] = []
valogatSz (k : ve) = case k of
   BankKartya _ tu _ -> tu : valogatSz ve
   _ -> valogatSz ve
```

Algebrai adattípusok

A `valogatSzF` a `foldr` függvényt alkalmazza. A `foldr` helyett alkalmazhattuk volna valamelyik `foldl` változatot is.

```
valogatSzF :: [BankSzamla] -> [Tulajdonos]
valogatSzF = foldr op []
   where
   op k res = case k of
       BankKartya _ tu _ -> tu : res
       _ -> res
```

A `valogatSzH` halmazműveleteket használ:

```
valogatSzH :: [BankSzamla] -> [Tulajdonos]
valogatSzH ls = [tu | BankKartya _ tu _ <- ls]
```

```
> valogatSz lsBsz
["Kiss Antal","Nagy Antal","Beres Antal"]
```

Algebrai adattípusok

7. feladat

Írjunk egy Haskell-függvényt, amely külön listákba teszi a `BankKartya`, a `Keszpenz` és a `Szamla` értékkonstruktorokra vonatkozó adatokat. A `Keszpenz` esetében csupán számoljuk meg, hogy hány készpénzkifizetés van.

```
valogatBK :: [BankSzamla] -> ([BankSzamla], [Int], [BankSzamla])
valogatBK ls = auxV ls [] [0] []
   where
   auxV [] bLs kLs sLs = (bLs, kLs, sLs)
   auxV (k : ve) bLs kLs sLs = case k of
       BankKartya {} -> auxV ve (k : bLs) kLs sLs
       Keszpenz -> auxV ve bLs [1 + head kLs] sLs
       Szamla _ -> auxV ve bLs kLs (k : sLs)

> valogatBK lsBsz
([BankKartya "98765" "Beres Antal" ["Mvh",...
```

A case-ben, a `BankKartya` értékkonstruktor esetében a `_ _ _` helyett a `{}` szimbólumokat használjuk, mert a `->` jobb oldalán egyetlenegy mezőértékkel sincs műveletvégzés.

Algebrai adattípusok

Az elegánsabb kiíratáshoz `mapM_`-et használunk, amelyet külön-külön alkalmazunk a tuple elemekre, amelyeket a `valogatBK` függvény visszatérési értékeként kaptunk meg:

```
foValogat :: [BankSzamla] -> IO ()
foValogat ls = do
   let (t1, t2, t3) = valogatBK ls
   putStrLn "Bankkartya adatok:"
   mapM_ print t1
   putStrLn "Keszpenzkifizetesek szama: "
   mapM_ print t2
   putStrLn "Szamla adatok:"
   mapM_ print t3

> foValogat lsBsz
Bankkartya adatok:
BankKartya "98765" "Beres Antal" ["Mvh","Romania"]
...
```