Funkcionális programozás
9. előadás

MÁRTON Gyöngyvér

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

2023, tavaszi félév

Miről volt szó?

- hibakezelés: `error`, `catch`, `Maybe`, `Either`
- Haskell I/O műveletek, bináris állományok, feladatok
  - bináris állomány hexa alakja
  - állomány mérte, bájtban
  - állományról való másolatkészítés
  - két állomány tartalmának összehasonlítása
  - állomány tartalmának titkosítása (xor)
- típusok és adatszerekzetek: rekord típusok

Miről lesz szó?

- rekord típusok, tárolás szövegállományban
- algebrai adattípusok
- feladatok (rekord típusok):
  - maximum keresés
  - rendezés, a rendezési feltétel mint függvényparaméter
- feladatok (algebrai adattípusok):
  - összeszámolás, kiválogatás
  - kiíratási lehetőségek

Rekord típusok, tárolás szövegállományban

1. 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 8. előadáson használt típusszinonimaként definiált `Jovedelem` típust fogjuk használni, illetve az `auxEv` függvényt
- feltételezve, hogy ezek az `eload8.hs`-ben vannak megadva, módosítsuk az `eload8.hs` tartalmát, az első sorba írjuk be: `module Eload8 where`
- ezután a feladathoz tartozó definíciókat, illetve függvényeket írjuk az `eload9.hs`-be, amelynek első sorába írjuk: `import qualified Eload8 as E8`
- ekkor megadható a következő típus, amelyet a feladat adatainak a feldolgozásakor fogunk használni
- vegyük észre hogy ez más mint ahogy az `eload8.hs`-ben definiáltuk az Alkalmazott típust:

```
data Alkalmazott = Alkalmazott String [E8.Jovedelem]
    deriving (Show, Read)
```

Rekord típusok, tárolás szövegállományban

- 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 8. előadásban is állományban tároltuk az adatokat, de ott az `alkalmazottData.txt` állomány szerkezete bonyolultabb volt:

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

Rekord típusok, tárolás szövegállományban

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, tárolás szövegállományban

- 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 -> [E8.Jovedelem]) tJovA
```

Rekord típusok, tárolás szövegállományban

```
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, tárolás szövegállományban

```
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 = E8.auxEv ev jovA
            k = case temp of
                    Nothing -> -1
                    Just x -> x
```

Egy alkalmazott adott évbeli jövedelemértékének a kiválasztását az `eload8.hs` található auxEv függvény segítségével végeztük.

Rekord típusok, tárolás szövegállományban

A következő feladatban a quickSort algoritmust tárgyaljuk újra, példákon keresztül mutatjuk be, hogy rekord típusok esetében hogyan lehet használni.

2. feladat

Egy adott telefonról a következő adatok vannak eltárolva: név, eladási adatok. Minden telefon esetében az eladási adatok egy érték-hármasból álló listát jelentenek, ahol az érték-hármas tartalmaz egy évszámot, az adott évhez tartozó telefonárat és az eladott telefonok számát. Írjunk egy Haskell-függvényt, amely rendez egy `Telef` elemtípusú listát, a telefonnevek alapján ábécésorrendbe.

```
data Telef = Telef{
    tNev :: String,
    tEladas:: [(Int, Int, Int)]
    } deriving (Show)
```

Az algoritmusokat a következő konstans listára fogjuk meghívni:

```
lsT = [
    Telef "HuaweiP20" [(2017,1800,10),(2018,1600,20)],
    Telef "SamsungGalaxyS9" [(2018,3500,25)],
    Telef "HuaweiP10" [(2016,1200,20),(2017,1000,15),(2018,950,18)],
    Telef "Lenovo" [(2013,600,5),(2018,450,12)],
    Telef "iPhone" [(2016,2000,9),(2017,2300,15),(2018,2400,6)],
    Telef "SamsungGalaxyJ5" [(2013,700,10),(2018,900,15)]]
```

Rekord típusok, tárolás szövegállományban

név szerinti rendezés, illetve darabszám szerinti rendezés `sortBy` `sortOn`-al:

```
> import Data.List (sortBy)
> import Data.Ord (comparing)
> sortBy compare [8.5, 7.33, 9.25, 8,75, 5.5, 7.33]

> fgMap (Telef nev eladas) = nev ++ "\n" ++ myShow eladas
> mapM_ (putStrLn . fgMap) $ sortBy (comparing tNev) lsT

> sumE x = sum [t3 | (t1, t2, t3) <- tEladas x]
> mapM_ (putStrLn . fgMap) $ sortBy (comparing sumE) lsT

> import Data.List ( sortOn )
> import Data.Ord (Down(Down))

> fgSortN (Telef nev eladas) = Down nev
> mapM_ (putStrLn . fgMap) $ sortOn fgSortN lsT

> fgSortE (Telef nev eladas) = negate $
> mapM_ (putStrLn . fgMap) $ sortOn fgSortE lsT
```

Record típusok

Saját rendező algoritmust írunk:

```
quickT :: [Telef] -> [Telef]
quickT [] = []
quickT (k : ve) = quickT kLs ++ [k] ++ quickT nLs
   where
   kLs = [x | x <- ve, tNev x < tNev k]
   nLs = [x | x <- ve, tNev x >= tNev k]

> quickT lsT
[Telef {tNev = "HuaweiP10", tEladas = [(2016,1200,20)...]}
...
```

Ha elegáns kiíratást szeretnénk, akkor még szükségünk lesz a `showTelef` függvényre:

```
> putStr $ showTelef $ quickT lsT
HuaweiP10
2016 1200 20
2017 1000 15
2018 950 18
...
```

Rekord típusok, tárolás szövegállományban

```
showTelef :: [Telef] -> String
showTelef = concatMap auxT
   where
   auxT :: Telef -> [Char]
   auxT k = tNev k ++ "\n" ++
            showElad (tEladas k) ++ "\n"

   showElad :: [(Int, Int, Int)] -> String
   showElad = concatMap auxE
       where
       auxE k = showTuple k ++ "\n"

   showTuple :: (Int, Int, Int) -> [Char]
   showTuple (k1, k2, k3) = show k1 ++ " " ++
                            show k2 ++ " " ++
                            show k3
```

Rekord típusok, tárolás szövegállományban

A concatMap az első paraméterként megadott függvényt alkalmazza a listaelemekre, majd a kapott elemeket egymás után fűzi:

```
> ls = ["Nera-szurdok", "Nemzeti", "Park"]
> concatMap (++ " ") ls
"Nera-szurdok Nemzeti Park "

> ls = ["nera-szurdok", "nemzeti", "park"]
> auxC xLs = ' ' : (toUpper . head) xLs : tail xLs
> concatMap auxC ls
" Nera-szurdok Nemzeti Park"
```

Rekord típusok, tárolás szövegállományban

3. feladat

Írjunk egy Haskell-függvényt, amely rendez egy `Telef` elemtípusú listát, a telefonok összeladási darabszáma alapján.

Az elemek összehasonlításakor, a `sumE` függvényt alkalmazzuk, amelynek az a szerepe, hogy egy adott telefon esetében meghatározza, hogy hány darab telefont adtak el:

```
quickTa :: [Telef] -> [Telef]
quickTa [] = []
quickTa (k : ve) = quickTa kLs ++ [k] ++ quickTa nLs
    where
    kLs = [x | x <- ve, sumE x < sumE k]
    nLs = [x | x <- ve, sumE x >= sumE k]

 sumE :: Telef -> Int
 sumE x = sum [thd y | y <- tEladas x]
    where
    thd (y1, y2, y3) = y3
```

Rekord típusok, tárolás szövegállományban

A `showTelefSum` formázva írja ki a rendezett adatokat, a lista elemeinek a feldolgozását a `foldl'` segítségével oldjuk meg, illetve a telefonnevek mellett az összeladási értéket is fetüntetjük:

```
import Data.List (foldl')
showTelefSum :: [Telef] -> String
showTelefSum = foldl' op ""
   where
   op res k = res ++ tNev k ++ "\nOssz ertek: " ++
              showElad (tEladas k) ++ "\n\n"

   showElad :: [(Int, Int, Int)] -> String
   showElad ls = show $ sum $ map thd ls
       where
       thd (y1, y2, y3) = y3

> putStr $ showTelefSum $ quickTa lsT
Lenovo
Ossz ertek: 17

SamsungGalaxyS9
Ossz ertek: 25
...
```

Rekord típusok, tárolás szövegállományban

- a következő kódsorokban az összehasonlítási feltételt függvényparaméterként adjuk meg
- a `quickSort` alkalmas lesz tetszőleges típusú adatszerkezetek adatainak a rendezésére

```
quickSort :: (a -> a -> Bool) -> [a] -> [a]
quickSort fg [] = []
quickSort fg (k : ve) =
   quickSort fg kLs ++ [k] ++ quickSort fg nLs
   where
   kLs = [x | x <- ve, fg x k]
   nLs = [x | x <- ve, not $ fg x k]
```

Rekord típusok, tárolás szövegállományban

Egy `Telef` típus esetében, ha név szerinti rendezést akarunk, akkor egy `compareNev` függvényt kell írni, amelyet paraméterként kell alkalmazni a `quickSort`-ben:

```
compareNev :: Telef -> Telef -> Bool
compareNev x y = tNev x <= tNev y

> putStr $ showTelef $ quickSort compareNev lsT
HuaweiP10
2016 1200 20
2017 1000 15
2018 950 18

HuaweiP20
...
```

Rekord típusok, tárolás szövegállományban

darabszám szerinti rendezés:

```
compareSum :: Telef -> Telef -> Bool
compareSum x y = sumE x <= sumE y

> putStr$ showTelefSum $ quickSort compareSum lsT
Lenovo
Ossz ertek: 17

SamsungGalaxyJ5
Ossz ertek: 25
...
```

valós számok rendezése:

```
compareR :: Ord a => a -> a -> Bool
compareR x y = x < y

> quickSort compareR [8.5, 7.33, 9.25, 8,75, 5.5, 7.33]
[5.5,7.33,7.33,8.0,8.5,9.25,75.0]
```

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
```

- a `Maybe` és `Either` típusok tárgyalásánál említettük, hogy a típusdeklarációban 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

4. 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

5. 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

6. 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"]
...
```