

Funkcionális programozás

10. előadás

MÁRTON Gyöngyvér

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

2024, tavaszi félév

Miről volt szó?

- Eratosztenész szitája, szerencsés számok (Lucky Numbers)
- szövegállományok: `hGetContents`, `writeFile`, `readFile`
- Haskell I/O műveletek, bináris állományok, feladatok
 - bináris állomány hexa alakja
 - állomány mérete, bájtban
 - állomány tartalmának titkosítása (xor)

Miről lesz szó?

- hibakezelés: `error`, `catch`, `Maybe`, `Either`
- típusok és adatszerkezetek: rekord típusok

Hibakezelés, *catch*

- a `Control.Exception` könyvtárban van
- két bemeneti értéket vár, az első paraméter típusa `IO a`, ami akcióknak egy olyan sorát jelenti, ahol az utolsó akció `a` típusú értéket ad; ezen akciók valamelyikének a futási hibáját kell *elkapja*
- a második paramétere a kivételt kezelő függvény, kimeneti értékének típusa szintén `IO a`

```
import Control.Exception
osztFg :: Double -> Double -> IO()
osztFg x y = catch (print (fg x y)) kivetelKezelo
  where
    kivetelKezelo :: SomeException -> IO ()
    kivetelKezelo err = do
      let ls = "Hiba jellege: " ++ show err
      putStrLn ls
    fg x y = if y == 0 then error "osztas 0-val" else x / y
```

```
> osztFg 2 3
0.6666666666666666
> osztFg 2 0
```

```
Hiba jellege: Nullával valo osztas!...
```

Hibakezelés, *catch*

A `catch` függvényt most egy file megnyitása esetén használjuk:

```
import Control.Exception
import System.IO

foAll :: IO()
foAll = catch ( do
    inf <- openBinaryFile "valami.jpg" ReadMode
    ls <- hGetContents inf
    print $ length ls
    hClose inf
  ) kivételKezelo
where
  kivételKezelo :: SomeException -> IO ()
  kivételKezelo err = do
    putStrLn $ "Hiba jellege: " ++ show err

> foAll
Hiba jellege: valami.jpg: openFile: does not exist...
```

Hibakezelés, *catch*

A `catch` függvényt most billentyűzetről történő beolvasás-ellenőrzésre használjuk:

```
import Control.Exception
```

```
olvasDouble :: IO Double
olvasDouble = do
    str <- getLine
    return (read str :: Double)
```

```
foOlvas :: IO ()
foOlvas = catch ( do
    putStr "n = "
    n <- olvasDouble
    putStrLn $ "negyzetgyoke: " ++ show (sqrt n)
) kivételKezelo
where
    kivételKezelo :: SomeException -> IO ()
    kivételKezelo err = do
        putStrLn $ "Hiba jellege: " ++ show err
```

```
> foOlvas
```

```
n = 4
```

```
negyzetgyoke: Hiba jellege: Prelude.read: no parse
```



Rekord típusok

- a korábban megismert adattípusok mellett új adattípusokat lehet definiálni, pl. rekord típusokat
- a definiáláshoz a **data** kulcsszót kell használni, ahol a választott név nagybetűvel kell kezdődjön:

```
data DiakT = DiakE String Int [Double]  
  deriving Show
```
- a DiakT lesz az új típusú adatszerkezet neve, amelyet típuskonstruktornak hívunk
- a DiakE azonosító értékkonstruktor lesz, ezt használjuk, amikor egy DiakT típusú adatnak értéket akarunk adni
- a String, Int, [Double] a mezők típusát határozzák meg
- a **deriving**-ban megadott típusosztályok függvényei az újonnan definiált típus esetében is használhatóak lesznek

Rekord típusok

- a gyakorlatban a típus- és értékkonstruktorok gyakran **azonos nevet** kapnak, ez alapján újradefiniáljuk a fenti adatszerkezetet, és a továbbiakban így fogunk dolgozni:

```
data Diak = Diak String Int [Double]
    deriving(Show)
```

- ha értéket szeretnénk adni, akkor választanunk kell kisbetűvel kezdődő nevet:

```
diak = Diak "David" 5 [7.90,9.85,9.95,8.55]
```

- a Diak értékkonstruktor argumentumaként megadott értékek típusa meg kell egyezzen a típus definiálásakor megadott mezők típusával

- a sorrendet is be kell tartani, ellenkező esetben fordítási hibát kapunk:

```
diak = Diak 5 "David" [7.90,9.85,9.95,8.55]
```

```
...
```

- **Couldn't match expected type 'Int' with actual type ...**

Mezőkre való hivatkozás

- a mezőkre való hivatkozást mintaillesztéssel lehet megoldani
- a következő kiírja a diak-ban megadott nevet és a legnagyobb jegyet:

```
myShowDiak :: Diak -> String
myShowDiak k = nev ++ " " ++ show (maximum jegy)
  where
    Diak nev _ jegy = k
```

```
> putStrLn $ myShowDiak diak
David 9.95
```

- a mintaillesztést a következőképpen is meg lehet oldani:

```
myShowDiak_ :: Diak -> String
myShowDiak_ (Diak nev _ jegy) = nev ++ " " ++ show (maximum jegy)
```

Lista inicializálása

- egy Diak elemtípusú lista inicializálása, ahol vesszőt csak a listaelemek közé kell tenni, a mezőértékek között tilos a vesszők használata:

```
lsD = [ Diak "Ferenc" 1 [7.5,9.25],  
        Diak "Katalin" 2 [7.75,6.25,10,7.55],  
        Diak "Maria" 3 [6.5,8.25,9.33],  
        Diak "Zsuzsa" 4 [7.33,8.25,9.75],  
        Diak "David" 5 [7.90,9.85,9.95,8.55]]
```

- a myPrintDiakLs kiírja soronként a diákok neveit és a legnagyobb jegyüket:

```
myPrintDiakLs :: [Diak] -> IO()  
myPrintDiakLs = mapM_ (putStr . auxF)  
  where  
    auxF :: Diak -> String  
    auxF (Diak k1 k2 k3) = k1 ++ " " ++ show (maximum k3) ++ "\n"  
  
> myPrintDiakLs lsD  
Ferenc 9.25  
...
```

Típusszinonimák

- kifejezőbb adatszerkezet-nevek definiálásakor típusszinonimákat lehet létrehozni:

```
type Nev = String
type Kod = Int
type Jegyek = [Double]
```

```
data DiakM = DiakM Nev Kod Jegyek
    deriving Show
```

- a Haskell nem engedi meg, hogy az ugyanolyan szerkezetű, de más nevű adatszerkezetek használatát összekeverjük
- módosítjuk a myShowDiak, korábban megírt függvényünk szignatúráját, a diak-ot azonban ugyanúgy inicializáljuk, ezért futási hibát kapunk:

```
myShowDiak :: DiakM -> String
myShowDiak k = nev ++ " " ++ show (maximum jegy)
    where
        DiakM nev _ jegy = k
```

```
diak = Diak "David" 5 [7.90,9.85,9.95,8.55]
```

```
> myShowDiak diak
```

```
... error:
```

```
Couldn't match expected type 'DiakM' with actual type '...'
```



Rekord típus mezőnevek megadásával

- adatszerkezetek definiálása történhet mezőnevek megadásával, ahol a mezőnevek mindig kisbetűvel kell kezdődjenek:

```
type Nev = String
type Jegy = Double
type Ev = Int
```

```
data Hallgato = Hallgato{
    hNev :: Nev,
    hJegy :: Jegy,
    hEv :: Ev
} deriving (Show)
```

- mező értékének a módosítása:

```
hallgatoA = Hallgato "Mari" 4.5 1
```

```
modosit :: Hallgato -> Double -> Int -> Hallgato
modosit hallgato j e = hallgato {hJegy = j, hEv = e}
```

- a következő lekérdezés után a hallgatoA1 a hallgatoA módosított értékét fogja jelölni.

```
> hallgatoA1 = modosit hallgatoA 8.7 2
```

```
> hallgatoA1
```

```
Hallgato {hNev = "Mari", hJegy = 8.7, hEv = 2}
```

Rekord típusok, feladatok

A következőkben egyszerű algoritmusokat adunk meg, ahol a függvényeknek paraméterként a következő konstans értékekkel inicializált a `hallgatoL` listát adhatjuk meg:

```
hallgatoL :: [Hallgato]
hallgatoL =
    [Hallgato "Sari" 8.75 1, Hallgato "Mari" 4.25 1,
     Hallgato "Feri" 3.5 2, Hallgato "Zsuzsi" 10.0 2,
     Hallgato "Laci" 8.5 2, Hallgato "Lori" 7.5 2]
```

1. feladat

Írjunk egy Haskell-függvényt, amely egy `Hallgato` elemtípusú lista esetében megszámolja, hogy hány diáknak van átmenőjegye.

```
szamol :: [Hallgato] -> Int
szamol ls = length $ filter (\k -> hJegy k > 4.5) ls

> szamol hallgatoL
4
```

Rekord típusok, feladatok

2. feladat

Írjunk egy Haskell-függvényt, amely egy `Hallgato` elemtípusú lista esetében kiválogatja az elsőéves személyeket.

```
valogat :: Ev -> [Hallgato] -> IO ()
valogat e ls = mapM_ (putStrLn . myShow) nLs
  where
    nLs = foldr op [] ls
    op :: Hallgato -> [Hallgato] -> [Hallgato]
    op k rLs = if hEv k == e then k : rLs else rLs
```

```
myShow :: Hallgato -> String
myShow k = hNev k ++ " " ++ show (hJegy k) ++ " "
          ++ show (hEv k)
```

```
> valogat 1 hallgatoL
Sari 8.75 1
Mari 4.25 1
```

Rekord típusok, feladatok

a valogat függvényt halmazkifejezések segítségével is megadjuk:

```
valogatLC :: Ev -> [Hallgato] -> IO()  
valogatLC e ls = mapM_ (putStrLn . myShow) nLs  
  where  
    nLs = [k | k <- ls, hEv k == e]
```

Rekord típusok, feladatok

3. feladat

Írjunk egy Haskell-függvényt, amely egy `Hallgato` elemtípusú lista esetében meghatározza a jegyek átlagát.

```
import Data.List

atlagHFoldL :: [Hallgato] -> Jegy
atlagHFoldL ls = ossz / db
  where
    (ossz, db) = auxAtlag ls
    auxAtlag :: [Hallgato] -> (Double, Double)
    auxAtlag ls = foldl' op (0.0, 0.0) ls
      where
        op (x1, x2) k = (x1 + hJegy k, x2 + 1)

> atlagHFoldL hallgatoL
7.083333333333333
```


Rekord típusok, szövegállományok

4. feladat

Egy cég egy adott alkalmazotról 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 értékke:

```
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

5. feladat

Egy cég egy adott alkalmazotról 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 típuszinonimaként 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 kerülnek, 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.