Funkcionális programozás
10. előadás

MÁRTON Gyöngyvér

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

2025, tavaszi félév

Miről volt szó?

- hajtogatások (fold operations): `scanl, scanr`
- Haskell monádok
- a `Maybe` monád
- az `IO` monád
- a standard bemenet és kimenet
- Haskell I/O műveletek, állománykezelés:
  `openFile, hClose, hIsEOF, hGetLine, hPutStr`
- Hamming számok

Miről lesz 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

hGetContents, writeFile, readFile

A Haskell más állomány feldolgozási módszert is lehetővé tesz: lusta kiértékelési módszer szerint egy adott adatot csak akkor dolgoz fel, ha az értékére feltétlenül szüksége van.

- `hGetContents:: Handle -> IO String`

  Meghatároz egy `String`-et, amelyen keresztül elérhető az állományban levő összes karakter, de egyszerre csak annyi karakter kerül beolvasásra amennyit éppen feldolgoz egy adott függvény. Leggyakrabban akkor használjuk, ha beolvasunk *valamennyi* adatot egy fileból, valamit csinálunk velük, majd kiírjuk máshová.

A `writeFile`-t, `ReadFile`-t a `hGetContents` helyett mint *shortcut*-ot szokták használni: kezelik a file megnyítását, bezárását, olvasását, írását, stb.

- `writeFile :: FilePath -> String -> IO ()`

  Kiírja a megadott állományba (első paraméter), a megadott `String` típusú értéket (második paraméter).

- `readFile :: FilePath -> IO String`

  Beolvassa a megadott állomány tartalmát egy `String` típusú értékbe.

Haskell I/O műveletek, állománykezelés

1. feladat

Olvassuk be a korábban létrehozott `hamming.txt` állomány tartalmát, majd tegyük át egy másik fileba a 3-al és 5-el osztható Hamming számokat.

```
mainHammingRead1 :: IO()
mainHammingRead1 = do
    inf <- openFile "hamming.txt" ReadMode
    outf <-openFile "hamming3_5.txt" WriteMode
    temp <- hGetContents inf
    ls <- hammingToList temp
    hPutStr outf $ hammingToStr ls
    hClose inf
    hClose outf
```

A `hammingToList` meghatározza a Hamming számok listáját, a `hammingToStr` pedig a listaelemek alapján egy `String`-et hoz létre, közben meghatározza a 3, 5-el osztható Hamming számokat.

Haskell I/O műveletek, állománykezelés

```
hammingToList :: String -> IO [Int]
hammingToList temp = mapM aux $ lines temp
    where
    aux temp = do
        let [k1, k2] = words temp
        let k = read k2 :: Int
        return k

hammingToStr :: [Int] -> String
hammingToStr [] = ""
hammingToStr (k : ve) = kStr ++ hammingToStr ve
        where
            kStr = if even k then "" else show k ++ "\n"
```

Az állomány soronkénti feldolgozásához a `lines` függvényt használtuk, amely a `String` típusú bemeneti paraméterét az új sor jelek mentén felosztja több `String`-re:

```
lines :: String -> [String]

> lines "12\n123\n1234\n12345"
["12","123","1234","12345"]
```

Haskell I/O műveletek, állománykezelés

A Hamming számokat tartalmazó állomány tartalmát most a `readFile` segítségével olvassuk be, a `hammingFilter` függvénnyel kiszűrjük a 3, 5-el osztható számokat, majd az `writeFile`-al kiírjuk az eredményt egy másik állományba.

```
mainHammingRead2 :: IO ()
mainHammingRead2 = do
    hLista <- readFile "hamming.txt"
    let ls = hammingFilter hLista
    writeFile "hamming3_5.txt" ls

hammingFilter :: String -> String
hammingFilter :: String -> String
hammingFilter temp = tail $ foldr op "" $ lines temp
    where
    op temp res = if even k then res else "\n" ++ show k ++ res
        where
        ls = words temp
        k = read (ls !! 1) :: Int
```

Haskell I/O műveletek, állománykezelés

2. feladat

Olvassuk be az `szamokSor.txt` állomány tartalmát, hozzunk létre egy tuple listát, rendezzük a listát, a tuple-ok első eleme szerint, majd írjuk ki a rendezett adatokat a `rSzamokSor.txt` állományba.

A feladatot korábban megoldottuk, az állománykezeléshez most a `readFile, writeFile` függvényeket fogjuk használni, és feltételeztük, hogy a `szamokSor.txt` tartalma a következő struktúrájú:

```
12 Mari [9.5,10.5]
56 Zsuzsa [8.5,5.5,7.75]
89 Kata [9]
11 Ferko [4.5,7.9,8.5,10]
```

```
mainRendez = do
    temp <- readFile "szamokSor.txt"
    let ls = myReadLine temp
    let rLs = sortBy (comparing myFst) ls
    writeFile "rSzamokSor.txt" $ myShowLine rLs

myFst (t1, t2, t3) = t1
```

Haskell I/O műveletek, állománykezelés

A soronkénti feldolgozást, a `lines`-al végezzük:

```
myReadLine :: String -> [(Int, String, [Double])]
myReadLine temp = map aux $ lines temp
    where
    aux k = (read k1 :: Int, k2, lsK3)
        where
        [k1, k2, k3] = words k
        lsK3 = sort $ read k3 :: [Double]

myShowLine :: [(Int, String, [Double])] -> String
myShowLine [] = ""
myShowLine (k : ve)
    = show k1 ++ " " ++ k2 ++ " " ++ show k3 ++ "\n" ++ myShowLine ve
    where
    (k1, k2, k3) = k
```

Haskel állománykezelés

3. feladat

Eratoszthenész szitájával generáljuk ki az első n prímszámot és az eredményt írjuk ki egy állományba (prim.txt).

```
primFileWrite1 :: IO ()
primFileWrite1 = do
    putStr "n = "
    temp <- getLine
    let n = read temp :: Int
    let ls =  eSzita n
    writeFile "prim.txt" (show ls)
```

Eratosztenész szitája

Eratosztenész szitája, ahol az `eSzita` függvénynek paraméterként meg kell adni, hogy hány prímszámot generáljon. A `szita` függvényt lehet a `takeWhile`-el is használni.

```
eSzita :: Int -> [Int]
eSzita n = take n (2 : szita [3, 5..])

szita :: [Int] -> [Int]
szita (k : ve) = k : szita [x | x <- ve, mod x k /= 0]

> eSzita 10
[2,3,5,7,11,13,17,19,23,29]

> takeWhile ( <= 10 ) $ 2 : szita [3, 5..]
[2,3,5,7]
```

Haskell I/O műveletek, állománykezelés

4. feladat

Olvassuk be és írjuk ki a képernyőre a `prim.txt` *állomány tartalmát, majd írjuk ki az állományban található prímszámok számát.*

```
primFileRead1 :: IO()
primFileRead1 = do
    temp <- readFile "prim.txt"
    let ls = read temp :: [Int]
    putStrLn $ show ls
    let n = length ls
    putStr "primek szama: "
    putStrLn $ show n
```

Haskell I/O műveletek, állománykezelés

A következő kódsorban más formában írunk az állományba, szóközöket teszünk a prímszámok közé:

```
primFileWrite2 :: IO ()
primFileWrite2 = do
    putStr "n = "
    temp <- getLine
    let n = read temp :: Int
    let ls = eSzita n
    writeFile "primSz.txt" $ myShow ls

myShow :: [Int] -> String
myShow = foldr aux ""
    where
        aux k res = show k ++ " " ++ res
```

Haskell I/O műveletek, állománykezelés

Az előző oldalon megadott `myShow` függvény megadható az `intercalate, concat, concatMap` segítségével is:

```
myShow1 :: [Int] -> String
myShow1 ls = intercalate "" $ map (\x -> show x ++ " ") ls

myShow2 :: [Int] -> String
myShow2 = concat $ map (\x -> show x ++ " ")

myShow3 :: [Int] -> String
myShow3 = concatMap (\x -> show x ++ " ")
```

Haskell I/O, bináris állományok

- az operációs rendszer másképp kezeli a bináris fileokat és másképp a szövegállományokat, ezért a bináris állományok megnyítását a következő függvénnyel végezzük:

  ```
  openBinaryFile :: FilePath -> IOMode -> IO Handle
  ```

- a bináris állományok bezárása a már bemutatott `hClose`-al történik: amíg nincs meghíva, addig az adatokat az OP rendszer nem írja ki, ha írásra volt megnyítva egy file
- az alapértelmezett típus az állományok írásakor/olvasásakor a `String`, de ez nem tesz lehetővé hatékony adatkezelést, helyette egy későbbi előadásban bevezetésre fog kerülni a `ByteString` típus

Haskell I/O, állománykezelés

5. feladat

Írjuk ki egy szövegállományba egy tetszőleges állomány bájtjainak, hexa értékét.

```
import Data.Char
import System.IO
import Numeric

mainHexa :: IO ()
mainHexa = do
    infB <- openBinaryFile "file.pdf" ReadMode
    outT <- openFile "fileHexa.txt" WriteMode
    bStr <- hGetContents infB
    let bHexa = alakitHexa bStr
    hPutStr outT bHexa
    hClose infB
    hClose outT
```

Haskell I/O, állománykezelés

```
alakitHexa :: [Char] -> [Char]
alakitHexa [] = []
alakitHexa (k : ve) = newK ++ alakitHexa ve
    where
    tempK = showHex (ord k) ""
    newK = tempK ++ " "
```

A `showHex` a Numeric modulban van, használatát a következő példa szemlélteti:

```
> showHex 1024 ""
"400"

> showHex 10 ""
"a"
```

Haskell I/O, bináris állományok

6. feladat

Határozzuk meg egy állomány bájt-méretét.

```
mainMeret :: IO ()
mainMeret = do
    inf <- openBinaryFile "kep.tif" ReadMode
    size <- hFileSize inf
    putStrLn "fsize: "
    print (fromIntegral size)
    hClose inf
```

A `hFileSize` az állomány bájt-méretét adja meg, szignatúrája a következő:

```
hFileSize :: Handle -> IO Integer
```

Haskell I/O, bináris állományok

7. feladat

Titkosítsuk egy adott állomány tartalmát, alkalmazva a `xor` műveletet (`Data.Bits`), majd fejtsük is vissza a rejtjelzett állományt. A titkosításhoz egy titkos információt, egy `key`-t fogunk használni, amit körkörösen alkalmazunk.

```
import System.IO
import Data.Char
import Data.Bits

mainFileCrypt = do
    let key = [12, 56, 255, 102, 113, 34, 56, 78, 121, 101]
    cryptFunc "file.pdf" "crypt.pdf" key
    putStrLn "vege a titkositasnak"
    cryptFunc "crypt.pdf" "nFile.pdf" key
    putStrLn "vege a visszafejtesnek"

> mainFileCrypt
```

Haskell I/O, bináris állományok

```
cryptFunc :: FilePath -> FilePath -> [Int] -> IO ()
cryptFunc nameIn namOut key = do
    inf <- openBinaryFile nameIn ReadMode
    outf <- openBinaryFile namOut WriteMode
    bLs <- hGetContents inf
    let eLs = encrypt bLs key key
    --let eLs = encrypt2 bLs key
    hPutStr outf eLs
    hClose inf
    hClose outf
```

- a titkosítást és a visszafejtést is a `cryptFunc` függvény végzi
- a titkosítás annyiból áll, hogy a bemeneti file bájtjai és a `key` lista bájtjai között alkalmazzuk az `xor`-t
- a `encrypt` függvényben a `key` elemeit körkörösen vesszük, ha pedig elfogynak az elemek, akkor újból a `key` első elemével folytatjuk, egészen addig, amíg a bemeneti file bájtjain is végig nem mentünk
- a módszer nem biztonságos!!

Haskell I/O, bináris állományok

```
encrypt :: [Char] -> [Int] -> [Int] -> [Char]
encrypt ls key xKey =
    if null ls then []
    else
        if null key then encrypt ls xKey xKey
        else (chr eK): encrypt ve veKey xKey
            where
            eK = xor (ord k) kKey
            k = head ls
            ve = tail ls
            kKey = head key
            veKey = tail key

encrypt2 :: [Char] -> [Int] -> [Char]
encrypt2 ls key = map fg $ zip ls (cycle key)
    where
    fg (t1, t2) = chr $ xor (ord t1) t2
```

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: Nullaval 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
    ) kivetelKezelo
    where
    kivetelKezelo :: SomeException -> IO ()
    kivetelKezelo 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)
    ) kivetelKezelo
    where
    kivetelKezelo :: SomeException -> IO ()
    kivetelKezelo err = do
        putStrLn $ "Hiba jellege: " ++ show err

> foOlvas
n = t
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]
```

8. 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

9. 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

10. 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
```