

Funkcionális programozás

8. előadás

MÁRTON Gyöngyvér

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

2024, tavaszi félév

Miről volt szó?

- rendezések:
 - beszúró rendezés (insertion sort),
 - gyorsrendezés (quick sort),
- hajtogatások (fold operations)
 - maximum elem pozíciói
 - prímszámok
 - fibonacci számok
 - könyvtárfüggetlen implementációja: sum, head, last, map, filter, elem, any,

Miről lesz szó?

- hajtogatások (fold operations): `scanl`, `scanr`
- Haskell monádok
- adatbevitel
- szövegállományok

Hajtogatások - `scanl`, `scanr`

- a `scanl` és `scanr` függvények a `foldl`, illetve `foldr` függvények általánosításai
- a `scanl` alkalmazza a `foldl`-t az `inits` által meghatározott listákon
- a `scanr` alkalmazza a `foldr`-t a `tails` által meghatározott listákon
- mindkét függvény esetében három bemeneti paramétert kell megadni:
 - egy bináris operátort
 - egy tetszőleges típusú értéket, a kezdőértéket
 - egy lista típusú adatot, amelynek az elemeit fogja a függvény feldolgozni
- használható a `scanl'` változat is, ami a szigorú kiértékelési stratégiára váltva hatékonyabbá teszi a függvénykiértékelést
- használható a `scanl1` is, ami megengedi a függvény használatát kezdőérték nélkül
- használatukhoz a `Data.List` könyvtárcsomagot kell importálni

Hajtogatások - scanl, scanr

A következő lekérdezésekben a `scanl`-t alkalmazva meghatározzuk a bemeneti `[1..4]` lista minden kezdeti szegmensén az elemek összegét, illetve az elemek szorzatát:

```
> import Data.List
> inits [1..4]
[[], [1], [1,2], [1,2,3], [1,2,3,4]]
> scanl (+) 0 [1..4]
[0, 1, 3, 6, 10]
[0, 0+1, (0+1)+2, ((0+1)+2)+3, (((0+1)+2)+3)+4]

> scanl' (*) 1 [1..4]
[1, 1, 2, 6, 24]
[1, 1*1, (1*1)*2, ((1*1)*2)*3, (((1*1)*2)*3)*4]
```

A `scanr` a listavégeken határozza meg az elemek összegét, illetve szorzatát:

```
> tails [1..4]
[[1,2,3,4], [2,3,4], [3,4], [4], []]
> scanr (+) 0 [1..4]
[10, 9, 7, 4, 0]
[(1+(2+(3+(4+0))))], (2+(3+(4+0))), (3+(4+0)), (4+0), 0]

> scanr (*) 1 [1..4]
[24, 24, 12, 4, 1]
[(1*(2*(3*(4*1))))], (2*(3*(4*1))), (3*(4*1)), (4*1), 1]
```

Hajtogatások - scanl, scanr

A következő lekérdezésekben a `scanl'` alkalmazza a bemeneti lista minden kezdeti szegmensén megadott lambda függvényt:

```
> scanl' (\ x y -> (x + y) / 2) 2 [1,2,3]
[2.0,1.5,1.75,2.375]
foldl' (\ x y -> (x+y)/2) 2 [] -> 2 -> 2.0
foldl' (\ x y -> (x+y)/2) 2 [1] -> (2+1)/2 -> 1.5
foldl' (\ x y -> (x+y)/2) 2 [1,2] -> ((2+1)/2+2)/2 -> 1.75
foldl' (\ x y -> (x+y)/2) 2 [1,2,3] -> (((2+1)/2+2)/2+3)/2 -> 2.375
```

A `scanr` a listavégeken alkalmazza a lambda függvényt:

```
> scanr (\ x y -> (x + y) / 2) 2 [1,2,3]
[1.625,2.25,2.5,2.0]
foldr (\ x y -> (x+y)/2) 2 [1,2,3] -> (((2+3)/2+2)/2+1)/2 -> 1.625
foldr (\ x y -> (x+y)/2) 2 [2,3] -> ((2+3)/2+2)/2 -> 2.25
foldr (\ x y -> (x+y)/2) 2 [3] -> (2+3)/2 -> 2.5
foldr (\ x y -> (x+y)/2) 2 [] -> 2 -> 2.0
```

Hajtogatások - scanl, scanr

1. feladat

Írjunk egy Haskell-függvényt, amely a scanl' függvényt használva kigenerálja egy listába az első n Fibonacci-számot.

Először próbáljuk ki a következő meghívásokat:

```
> scanl' (+) 1 [1,1,2,3]
```

```
> scanl' (+) 1 [1,1,2,3,5]
```

```
> map (foldl' (+) 1) $ inits [1,1,2,3]
```

```
> map (foldl' (+) 1) $ inits [1,1,2,3,5]
```

```
import Data.List (scanl')
fibonacci :: [Integer]
fibonacci = 1 : scanl' (+) 1 fibonacci
```

```
fibonacciLs :: Int -> [Integer]
fibonacciLs n = take n fibonacci
```

```
> fibonacciLs 10
[1,1,2,3,5,8,13,21,34,55]
```

Haskell monádok

- az adatbevitelhez, az adatkíráshoz, a hibakezeléshez, a tömbök kezeléshez szükséges műveleteket *monád*-ok segítségével végezzük
- a monád fogalma a kategóriaelméletből származik
- a monád egy számítási adattípust definiál:
 - megadjuk, hogy egy adattípus értékein milyen számításokat végezhetünk,
 - megadjuk, hogy ezek a számítások hogyan kombinálhatók, azaz milyen sorrendben végezhetők el
- a legegyszerűbb a `Monad m` típusosztály, amely a standard Prelude-ben van definiálva:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```


Haskell monádok

a `Monad m` típusosztályban két művelet, azaz két függvény/operátor típusdeklarációját láthatjuk:

- a `>>=` függvény számítások (akciók) láncolását teszi lehetővé
 - `do` jelölést használva egyszerűbb a szintaxisa
 - `do`-ra átírva:

```
a >>= f = do
    x <- a
    f x
```

- az `<-` operátorral az `a` akció eredményét kérjük le
 - az `<-` operátort most másképp használjuk, mint ahogy a halmazkifejezések vagy a `case` esetében tesszük, ez azonban nem probléma, a kontextusból megállapítható, hogy mikor melyikről van szó
 - tehát: lekérjük az `a` akció eredményét és az eredményen alkalmazzuk az `f` függvényt
- a `return` a paramétereként megadott adatot *becsomagolja* egy `m` típusú monádba

A Maybe monád

- hibák kezelésére alkalmas, eredmény nélküli számításokat tud kezelni
- a Maybe a legegyszerűbb monád, a standard könyvtárban van definiálva:
`data Maybe a = Nothing
 | Just a`
- a data kulcsszóval egy új típust lehet definiálni, jelen esetben egy paraméterezett típus van definiálva: `Maybe Int`, `Maybe String`, `Maybe [Double]` stb. típusokat egyaránt tud kezelni
- egy `Maybe` típusú adat kétféle értéket kaphat, a `Nothing` konstanst, és a `Just a` értéket, amelyeket értékkonstruktoroknak hívunk
- az értékkonstruktorok között kötelező a `|` szimbólum használata

Példa:

```
myHead :: [a] -> Maybe a  
myHead [] = Nothing  
myHead (k : ve) = Just k
```

```
> myHead "Bekas-szoros-Nagyhagymas Nemzeti Park"  
Just 'B'
```

```
> myHead1 ""  
Nothing
```

A Maybe monád

2. feladat

Írjunk egy Haskell-függvényt, amely a foldr segítségével implementálja a könyvtárfüggvény `init`-et, azaz határozzuk meg azt a listát, amely tartalmazza a bemeneti lista elemeit, kivéve az utolsót.

```
myInitAux :: [a] -> Maybe [a]
myInitAux ls = foldr op Nothing ls
  where
    op k Nothing = Just []
    op k (Just res) = Just (k : res)

> myInitAux ["bekas", "nera", "revi", "tordai"]
Just ["bekas", "nera", "revi"]

> myInitAux []
Nothing
```

A Maybe monád

A myInit segítségével kezelni fogjuk a Maybe monád által meghatározott értékeket:

```
myInit :: [a] -> [a]
myInit ls =
  case temp of
    Nothing -> error "Ures lista!"
    Just k -> k
  where
    temp = myInitAux ls
```

```
import Data.Maybe
myInit1 :: [a] -> [a]
myInit1 ls = do
  let temp = myInitAux ls
  if isNothing temp then error "ures"
  else do
    let Just k = temp
    k
```

```
> myInit ["bekas", "nera", "revi", "tordai"]
["bekas","nera","revi"]
```

Az IO monád

- az IO monád a Monad m típusosztály egy másik példánya
- a standard bemenetet/kimenetet, az állományok kezelését az IO monád segítségével végezzük
- a `getLine` a billentyűzetről történő adatbevitelt teszi lehetővé
 - típusdeklarációja: `getLine :: IO String`
 - nem vár bemeneti értéket, kimenetének típusa pedig `IO String`, azaz egy olyan számítást definiál, ahol a végeredmény típusa `String`
- a `putStrLn` a standard kimentre ír, kimenetének típusa `IO ()`
- a függvények/akciók láncolásához használható még a `>>` operátor:

```
mainIO_ = putStr "str: " >> getLine >>= putStrLn
```

- átírva `do` jelölésre:

```
mainIO = do
  putStr "str: "
  temp <- getLine
  putStrLn temp
```

A standard bemenet és kimenet

3. feladat

Írjunk egy Haskell-függvényt, amely meghatározza a billentyűzetről beolvasott n szám páros osztóinak listáját.

```
foParos0 :: IO ()
foParos0 = do
  putStr "n = "
  temp <- getLine
  let n = read temp :: Int
      let res = [i | i <- [2, 4 .. n], mod n i == 0]
  print res
```

```
> foParos0
n: 60
[2,4,6,10,12,20,30,60]
```

- a `getLine` által meghatározott értéket a `<-` művelet segítségével lekértük/kicsomagoltuk,
- a `read` függvényvel `Int` típusú adattá alakítottuk, majd `let ... =` jelölést használva `n`-el jelöltük
- a kért lista előállításához halmazműveletet használtunk

A standard bemenet és kimenet

4. feladat

Olvassunk be egy karakterláncot a billentyűzetről, majd határozzuk meg azt a két karakterláncot, amelyben csak a kis- és nagybetűk, illetve azt, amelyben csak a számjegyek szerepelnek.

```
import Data.Char (isAlpha, isNumber)

olvasStr :: IO ()
olvasStr = do
    putStr "kerek egy karakterláncot: "
    str <- getLine
    x <- return (filter isAlpha str)
    y <- return (filter isNumber str)
    putStrLn $ "Az alfanumerikus karakterek: " ++ x
    putStrLn $ "A számok: " ++ y
```

A return használata nem vonja maga után az olvasStr-ből való kilépést!

A standard bemenet és kimenet

5. feladat

Írjunk egy Haskell-függvényt, amely meghatározza a billentyűzetről beolvasott számok rendezett sorrendjét. Használjuk a sort könyvtárfüggvényt.

```
import Data.List (sort)

foRendez :: IO ()
foRendez = do
    putStr "szamokat kerek: "
    ls <- olvasLista
    let sortLs = sort ls
    putStr "rendezve: "
    print sortLs

olvasLista :: IO [Int]
olvasLista = do
    temp <- getLine
    let ls = read temp :: [Int]
    return ls
```


A standard bemenet és kimenet

Az `olvasLista` függvény számok beolvasását biztosítja, kimenetének típusa `IO [Int]`, ezért vigyáznunk kell arra, hogy helyes formátumban olvassuk be a bemenetet: használjunk szögletes zárójeleket, illetve vesszőket, úgy ahogy a lekérdezésből ez kitűnik:

```
> foRendez  
szamokat kerek: [12, 4, 67, 8, 9]  
rendezve: [4,8,9,12,67]
```

A standard bemenet és kimenet

6. feladat

Írjunk egy Haskell-függvényt, amely valós számokat olvas be a billentyűzetről, majd a számokat az egészrészük alapján különböző csoportokba, azaz listákba teszi.

```
import Data.List (sort, groupBy)

foCsoportosit :: IO ()
foCsoportosit = do
  putStr "szamokat kerek: "
  ls <- olvasSzamok
  let gLs = groupBy (\x y -> truncate x == truncate y) $ sort ls
  putStr "csoportositva: "
  print gLs

olvasSzamok :: IO [Double]
olvasSzamok = do
  temp <- getLine
  let ls = map (read :: String -> Double) $ words temp
  return ls
```

A standard bemenet és kimenet

- a `words` és `map` függvények használata miatt a számok beolvasásakor a számokat egy sorba, szóközöket téve közéjük kell beírni
- az egészrész alapján történő csoportosításhoz először a `sort-al` rendezzük az `ls` listát, majd a `groupBy`-al csoportosítjuk az egymás után következő, azonos egészrészsel rendelkező elemeket
- az egészrészt a `truncate`-el lehet meghatározni

```
> foCsoportosit
```

```
szamokat kerek: 1.77 5.6 1.4 5.34 2.7 2.9 1.9 2.4 1.33
```

```
csoportositva:
```

```
[[1.33,1.4,1.77,1.9],[2.4,2.7,2.9],[5.34,5.6]]
```

A standard bemenet és kimenet

- `myPrintList`, amelyet a `print` helyett kell használni a csoportok kiírásakor minden sorba csak az azonos csoportbeli elemeket írja
- az első `mapM_` a sorok kiírásáért felelős, a második a sorokon belüli értékeket írja ki, szóközt téve az elemek közé

```
myPrintList :: [[Double]] -> IO ()
myPrintList = mapM_ auxF1
  where
    auxF1 :: [Double] -> IO ()
    auxF1 kLs = do
      mapM_ auxF2 kLs
      putStrLn ""

    auxF2 :: Double -> IO ()
    auxF2 k = putStrLn $ show k ++ " "
```

A standard bemenet és kimenet

7. feladat

Írjunk egy Haskell-függvényt, amely beolvas a billentyűzetről több kételemű tuple elemtípusú értéket, úgy hogy minden tuple beolvasása után `enter`-t nyomunk. A tuple első eleme legyen `String`, a második pedig `Int` típusú. A függvénynek adjuk meg paraméterként, hogy hány elemet szeretnénk beolvasni.

```
olvasNTuple :: Int -> IO [(String, Int)]
olvasNTuple n = do
  temp <- getLine
  let (t1, t2) = read temp :: (String, Int)
  if n == 1 then return [(t1, t2)]
  else do
    ve <- olvasNTuple (n-1)
    return ((t1, t2) : ve)
```

A standard bemenet és kimenet

8. feladat

Írjunk egy Haskell-függvényt, amely az előző oldalon megadott olvasNTuple-el beolvasott listát rendezi az első elemek alapján.

Ha nem egy kételemű tuple szintaxisa szerint olvassuk be az elemeket, akkor hibát kapunk!

```
import Data.List (sortBy)
import Data.Ord (comparing)

rendezMain :: IO ()
rendezMain = do
  putStr "n: "
  temp <- getLine
  let n = read temp :: Int
      ls <- olvasNTuple n
      let rLs = sortBy (comparing fst) ls
          print rLs

> rendezMain
n: 3
("mari", 12)
("anna", 20)
("feri", 15)
[("anna",20),("feri",15),("mari",12)]
```

Haskell szövegállományok, a System.IO függvényei

- a Haskell más programozási nyelvekhez hasonlóan definiálja az alapvető I/O funkciókat
- Az I/O függvények esetén szükség van az `import System.IO`-ra
- az állományok kezeléséhez szükséges függvények esetében a függvények nevét `h`-val kell kezdeni
- a fontosabb függvények:
 - A paraméterként megadott állomány megnyitása. Megnyitási módok: `ReadMode`, `WriteMode`, `AppendMode`
`openFile :: FilePath -> IOMode -> IO Handle`
 - A paraméterként megadott állomány bezárása:
`hClose :: Handle -> IO ()`
 - `True`-t térít vissza ha az állomány végén vagyunk, másképp `False`-t:
`hIsEOF :: Handle -> IO Bool`

Haskell I/O, fontosabb függvények

- Beolvassa a paraméterként megadott állomány egy sorát:
`hGetLine :: Handle -> IO String`
- A paraméterként megadott string kiírása az állományba:
`hPutStr :: Handle -> String -> IO ()`
- Hasonló a `hPutStr`-hez, csak újsort tesz a kiírás végére:
`hPutStrLn :: Handle -> String -> IO ()`
- Kíírja a megadott paraméter string alakját egy állományba, újsor jelet tesz a kiírás végére:
`hPrint :: Show a => Handle -> a -> IO ()`

Haskell I/O műveletek, állománykezelés

9. 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 beolvasáskor feltételeztük, hogy a `szamokSor.txt` tartalma a következő struktúrájú:

```
12 Mari
56 Zsuzsa
89 Kata
11 Ferko
```

```
import System.IO
mainRendez = do
  inf <- openFile "szamokSor.txt" ReadMode
  outf <- openFile "rSzamokSor.txt" WriteMode
  ls <- myReadFile inf
  let rLs = sortBy (comparing fst) ls
  myWriteFile outf rLs
  hClose inf
  hClose outf
```

Haskell I/O műveletek, állománykezelés

Az állományból az adatokat soronként fogjuk beolvasni, alkalmazzuk a `hGetLine` függvényt:

```
myReadFile :: Handle -> IO [(Int, String)]
myReadFile inf = do
  feof <- hIsEOF inf
  if feof then return []
  else do
    temp <- hGetLine inf
    let [k1, k2] = words temp
        let k = (read k1 :: Int, k2)
        ve <- myReadFile inf
    return (k : ve)
```

Haskell I/O műveletek, állománykezelés

Az állományba az adatokat soronként írjuk ki, alkalmazzuk a `hPutStrLn` függvényt:

```
myWriteFile :: Handle -> [(Int, String)] -> IO()
myWriteFile outf ls = do
  if null ls then return ()
  else do
    hPutStrLn outf $ show k1 ++ " " ++ k2
    myWriteFile outf ve
      where
        (k1, k2) = head ls
        ve = tail ls
```

Hamming számok

10. feladat

Generáljuk ki az első n Hamming számot növekvő sorrendbe és írjuk ki egy állományba (`hamming.txt`) őket, minden sorba kiírva a szám sorszámát, majd space-el elválasztva a Hamming számot.

- egy Hamming szám általános alakja: $2^n \cdot 3^m \cdot 5^k$
- Az első 10 Hamming szám: [1, 2, 3, 4, 5, 6, 8, 9, 10]

Az algoritmus elvi működése:

- Az első Hamming szám az 1.
- Meghatározzuk azt a **három listát**, melynek elemei a $2 \cdot h$, $3 \cdot h$ illetve $5 \cdot h$ **összefüggéssel** határozhatóak meg, ahol h egy tetszőleges Hamming szám.
- Az így kigenerált három listát összefésüljük.

Hamming számok

A listák meghatározása:

```
hammingF :: [Integer]
hammingF = 1 : listaH
  where
    hamL = hammingF
    lista2h = [ 2 * h | h <- hamL ]
    lista3h = [ 3 * h | h <- hamL ]
    lista5h = [ 5 * h | h <- hamL ]
    listaH = myMerge3 lista2h lista3h lista5h
```

Hamming számok

A kigenerált két lista összefésülése

```
myMerge3 :: (Integral a) => [a] -> [a] -> [a] -> [a]
myMerge3 ls1 ls2 ls3 = myMerge ls1 $ myMerge ls2 ls3
```

```
myMerge :: (Integral a) => [a] -> [a] -> [a]
myMerge [] [] = []
myMerge ls1 [] = ls1
myMerge [] ls2 = ls2
myMerge ls1 ls2
  | k1 < k2 = k1 : myMerge ve1 ls2
  | k1 == k2 = myMerge ls1 ve2
  | otherwise = k2 : myMerge ls1 ve2
  where
    (k1 : ve1) = ls1
    (k2 : ve2) = ls2
```

Haskell I/O műveletek, állománykezelés

A feladat főfüggvénye:

```
import System.IO

mainHammingWrite :: IO()
mainHammingWrite = do
    putStr "n = "
    temp <- getLine
    let n = read temp :: Int
        let lHamming = take n hammingF
            outf <- openFile "hamming.txt" WriteMode
                hammingWrite outf lHamming 1
    hClose outf
```

Haskell I/O műveletek, állománykezelés

```
hammingWrite :: Handle -> [Integer] -> Integer -> IO()
hammingWrite outf ls i =
  if null ls then return ()
  else do
    hPutStrLn outf $ show i ++ " " ++ show k
    hammingWrite outf ve (i + 1)
    where
      k = head ls
      ve = tail ls
```


Haskell I/O műveletek, állománykezelés

A `hammingWrite` függvényt módosítjuk, a kiíratást a `hammingWrite2`-ben a `mapM_` függvénnyel végezzük:

```
hammingWrite2 :: Handle -> [Integer] -> [Integer] -> IO()
hammingWrite2 outf ls lsI = mapM_ aux $ zip ls lsI
  where
    aux (k, i) = hPutStrLn outf $ show i ++ " " ++ show k
```

```
import System.IO
mainHammingWrite2 :: IO ()
mainHammingWrite2 = do
  putStr "n = "
  temp <- getLine
  let n = read temp :: Int
      let lsH = take n hammingF
          outf <- openFile "hamming.txt" WriteMode
              hammingWrite2 outf lsH [1..]
      hClose outf
```

Haskell I/O műveletek, állománykezelés

11. feladat

Olvassuk be majd írjuk ki a képernyőre az előző feladatban létrehozott hamming.txt állomány tartalmát.

```
mainHammingRead1 :: IO ()
mainHammingRead1 = do
    inf <- openFile "hamming.txt" ReadMode
    hammingRead1 inf
    hClose inf
```

```
hammingRead1 :: Handle -> IO ()
hammingRead1 inf = do
    heof <- hIsEOF inf
    if heof then return ()
    else do
        temp <- hGetLine inf
        putStrLn temp
        hammingRead1 inf
```

Haskell I/O műveletek, állománykezelés

12. feladat

Olvassuk be az előző feladatban létrehozott hamming.txt állomány tartalmát, majd tegyük át egy listába a Hamming számokat.

```
mainHammingRead2 :: IO ()
mainHammingRead2 = do
    inf <- openFile "hamming.txt" ReadMode
    hLista <- hammingRead2 inf
    print hLista
    hClose inf

hammingRead2 :: Handle -> IO [Integer]
hammingRead2 inf = do
    heof <- hIsEOF inf
    if heof then return []
    else do
        temp <- hGetLine inf
        let [i, k] = map (read :: String -> Integer) $ words temp
        ve <- hammingRead2 inf
        return (k : ve)
```