Funkcionális programozás
9. előadás

MÁRTON Gyöngyvér

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

2024, tavaszi félév

Miről volt szó?

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

Miről lesz szó?

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

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]
```

Lucky Numbers

Szerencsés számok: hasonló szűrést végzünk az egész számok halmazán, mint az Eratosztenész szitája esetében, csak most a pozíció értékek alapján végezzük a szűrést a megmaradt listában:

```
ls1 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,
ls2 = [1,3,5,7,9,11,13,15,17,19,21,23,25,27,29]
ls3 = [1,3,7,9,13,15,19,21,25,27]
ls4 = [1,3,7,9,15,19,21,25]

mySelect :: Integral a1 => [a2] -> a1 -> [a2]
mySelect ls n = [snd x | x <- filter (fgFilter n) $ zip [1..] ls]
    where
        fgFilter n (i, nr) = mod i n /= 0

> ls2 = mySelect ls1 2
> ls3 = mySelect ls2 3
> ls4 = mySelect ls3 7
```

Lucky Numbers

```
luckyNr :: Integral a => Int -> [a]
luckyNr n = 1 : (take n $ rek 2 [1,3..])

rek :: Integral a => Int -> [a] -> [a]
rek i tls = e : rek (i+1) ls
    where
        ls = mySelect tls e
        e = tls !! (i-1)
```

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.

```
mainHammingRead3 :: IO()
mainHammingRead3 = 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.

```
mainHammingRead4 :: IO ()
mainHammingRead4 = do
    hLista <- readFile "hamming.txt"
    let ls = hammingFilter hLista
    writeFile "hamming3_5.txt" ls

hammingFilter :: String -> String
hammingFilter temp = foldl aux "" $ lines temp
    where
    aux res temp = if even k then res else res ++ show k ++ " "
        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.

```
mainRendez2 = do
    temp <- readFile "szamokSor.txt"
    ls <- myReadLine temp
    let rLs = sortBy (comparing fst) ls
    writeFile "rSzamokSor.txt" $ myShowLine rLs
```

Haskell I/O műveletek, állománykezelés

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

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

myShowLine :: [(Int, String)] -> String
myShowLine ls =
    if null ls then ""
    else show k1 ++ " " ++ k2 ++ "\n" ++ myShowLine ve
        where
            (k1, k2) = head ls
            ve = tail ls
```

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)
```

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) =
    if null ve then tempK
    else 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
- az alkalmazott titkosítási módszer nem okoz megfelelő biztonságot

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
```