Funkcionális programozás
8. előadás

MÁRTON Gyöngyvér

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

2023, tavaszi félév

Miről volt szó?

- hajtogatások (fold operations): polinom hellyettesítési értéke
- `scanl, scanr`
- Haskell monádok
- adatbevitel
- szövegállományok
- szövegállományok: `hGetContents, writeFile, readFile`

Miről lesz 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

Hibakezelés, *error*

Kisebb, informális programkódok esetében használják: nullával való osztáskor jelez hibát az alábbi kódsor.

```
osztF :: Double -> Double -> Double
osztF _ 0 = error "Nullaval valo osztas!"
osztF x y = x / y

> osztF 2 3
0.6666666666666666

> osztF 3 0
*** Exception: Nullaval valo osztas!...
```

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`, szignatúrája a következő:
  `catch :: Exception e => IO a -> (e -> IO a) -> IO a`

```
import Control.Exception
foOszt :: Double -> Double -> IO()
foOszt x y = catch (print $ osztF x y) kivetelKezelo
    where
    kivetelKezelo :: SomeException -> IO ()
    kivetelKezelo err = do
        let ls = "Hiba jellege: " ++ show err
        putStrLn ls

>  foOszt 2 3
0.6666666666666666
>  foOszt 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
```

Hibakezelés, a *Maybe*

Nullával való osztás esetében a visszatérési érték *Nothing*:

```
osztF1 :: Double -> Double -> Maybe Double
osztF1 _ 0 = Nothing
osztF1 x y = Just (x / y)

> osztF1 3 0
Nothing
```

A *case* kifejezés segítségével letesztelhetjük, hogy mi a `osztF1` visszatérési értéke:

```
foOszt1 :: Double -> Double -> IO()
foOszt1 x y =
    case osztF1 x y of
        Nothing -> putStrLn "Nullaval valo osztas!"
        Just k -> print k

> foOszt1 2 3
0.6666666666666666

> foOszt1 3 0
Nullaval valo osztas!
```

Hibakezelés, *Maybe*

1. feladat

Határozzuk meg a a következő összeget: $\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}$, *ahol az* $x_1, x_2, \ldots x_n$ *értékeket az* `ls` *listán keresztül adjuk meg*

```
recipOsszeg :: [Double] -> Maybe Double
recipOsszeg ls =
    if null ls then return 0.0
    else do
        temp <- osztF1 1.0 $ head ls
        res <- recipOsszeg $ tail ls
        return (temp + res)

> recipOsszeg [1,2,6,5]
Just 1.8666666666666667

> recipOsszeg [1,2,0,5,7]
Nothing
```

Hibakezelés, *Maybe*

```
foOsszeg :: [Double] -> IO ()
foOsszeg ls =
    case recipOsszeg ls of
        Nothing -> putStrLn "Osztasi hiba!"
        Just k -> print k
```

```
> foOsszeg [1, 2, 3, 0, 4]
Osztasi hiba!
> foOsszeg [1, 2, 3,  4]
2.083333333333333
```

```
foOsszeg1 :: [Double] -> IO ()
foOsszeg1 ls
    | temp == Nothing = putStrLn "Osztasi hiba!"
    | otherwise = print k
    where
        temp = recipOsszeg ls
        Just k = temp

import Data.Maybe
foOsszeg2 :: [Double] -> IO ()
foOsszeg2 ls
    | isNothing temp = putStrLn "Osztasi hiba!"
    | otherwise = print k
    where
        temp = recipOsszeg ls
        Just k = temp
```

Hibakezelés, *Either*

- hasonlóan a *Maybe*-hez a *Prelude*-ben van definiálva és két paramétere van:
  ```
  data Either a b = Left a | Right b
  ```
- A Left esete a hibát jelzi, és hibaüzenetre is van lehetőség
- a Right esetében sikeres a kód végrehajtása

Egy lista első elemének a meghatározása:

```
myHead :: [a] -> Either String a
myHead [] = Left "ures lista"
myHead (k : ve) = Right k

> myHead "Sapientia"
Right 'S'

> myHead ""
Left "ures lista"
```

Hibakezelés, a *Either* érték

A nullával való osztás hibakezelése:

```
osztF2 :: Double -> Double -> Either String Double
osztF2 _ 0 = Left "Nullaval valo osztas!"
osztF2 x y = Right (x / y)

> osztF2 2 3
Right 0.6666666666666666

foOszt2 :: Double -> Double -> IO()
foOszt2 x y =
    case osztF2 x y of
        Left r -> putStrLn r
        Right r -> print r

> foOszt2 2 3
0.6666666666666666

> foOszt2 3 0
Nullaval valo osztas!
```

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 a `ByteString` típust foguk használni

Haskell I/O, állománykezelés

2. 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

3. 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

4. feladat

Készítsünk másolatot egy tetszőleges állományról, 1. módszer

```
mainMasol1 :: IO ()
mainMasol1 = do
    inf <- openBinaryFile "kep.jpg" ReadMode
    outf <- openBinaryFile "nkep.jpg" WriteMode
    blista <- hGetContents inf
    hPutStr outf blista
    hClose inf
    hClose outf
```

Haskell I/O, bináris állományok

5. feladat

Készítsünk másolatot egy tetszőleges állományról, 2. módszer.

```
mainMasol2 :: IO ()
mainMasol2 = do
    inf <- openBinaryFile "kep.jpg" ReadMode
    outf <- openBinaryFile "nkep.jpg" WriteMode
    blista <- beolvasBajtok inf
    kiirBajtok outf blista
    hClose inf
    hClose outf
```

Haskell I/O, bináris állományok

Az író/olvasó segédfüggvények, ahol az állományból karakterenként olvasunk be, illetve írunk ki:

```
beolvasBajtok :: Handle -> IO [Char]
beolvasBajtok inf = do
    heof <- hIsEOF inf
    if heof then return []
    else do
        k <- hGetChar inf
        ve <- beolvasBajtok inf
        return (k: ve)

kiirBajtok :: Handle -> [Char] -> IO ()
kiirBajtok outf ls =
    if null ls then return ()
    else do
        hPutChar outf k
        kiirBajtok outf ve
        where
        k = head ls
        ve = tail ls
```

Haskell I/O, bináris állományok

6. feladat

Hasonlítsuk össze két állomány tartalmát, 1. módszer

```
mainHasonlit1 :: IO ()
mainHasonlit1 = do
    inf1 <- openBinaryFile "kep.jpg" ReadMode
    inf2 <- openBinaryFile "nkep.jpg" ReadMode
    bstr1 <- hGetContents inf1
    bstr2 <- hGetContents inf2
    let bInd = hasonlitBajtok bstr1 bstr2 1
    if bInd == 0 then
        putStrLn "OK, identical files"
    else do
        putStr "different files on poz: "
        putStrLn $ show bInd
    hClose inf1
    hClose inf2
```

Haskell I/O, bináris állományok

Az alkalmazott segédfüggvény::

```
hasonlitBajtok :: (Eq a1, Num a) => [a1] -> [a1] -> a -> a
hasonlitBajtok [] [] bInd = 0
hasonlitBajtok [] ve bInd = bInd
hasonlitBajtok ve [] bInd = bInd
hasonlitBajtok ls1 ls2 bInd
    | (k1 /= k2) = bInd
    | otherwise = hasonlitBajtok ve1 ve2 (bInd + 1)
      where
      k1 = head ls1
      ve1 = tail ls1
      k2 = head ls2
      ve2 = tail ls2
```

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
```

Rekord típusok

- a korábban megismert adattípusok 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 ...
```

Rekord típusok

- 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)
```

Rekord típusok

- 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
...
```

Rekord típusok

- 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 típusok

- 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

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

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

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

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
```

Rekord típusok

11. feladat

Egy cég egy adott alkalmazottról 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

Az állomány szerkezete lehetővé teszi az egyszerű adatkiovasást: a `temp`-be kerülő `String` típusú adatot egyetlen `read` segítségével át tudjuk alakítani `[Alkalmazott]` típusú értékké:

```
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

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

```
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
...
```