

Funkcionális programozás

11. előadás

MÁRTON Gyöngyvér

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

2024, tavaszi félév

Miről volt szó?

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

Miről lesz szó?

- Algebrai adattípusok
- Kombinatorikai feladatok

Algebrai adattípusok

- több értékkonstruktor is lehetséges
- a legismertebb algebrai adattípus a `Bool`, amely két értékkonstruktorral rendelkezik, a `True`-val és a `False`-szal:
`data Bool = False | True`
- az értékkonstruktorokat `|`-al kell elválasztani
- algebrai adattípust mi is definiálhatunk, például létrehozhatunk egy saját `Bool` típust, amelyet egy természetes szám párosságának a vizsgálatához ugyanúgy lehet használni, mint `Bool`-t:

```
data MyBool = Igaz | Hamis
  deriving (Show)
```

```
parosT :: Integral a => a -> MyBool
parosT n = if even n then Igaz else Hamis
```

Algebrai adattípusok

- a következő példában a Szemely típusnak két értékkonstruktor lesz, az első, a Diak három, míg a második, a Tanár két mezővel rendelkezik
- az értékkonstruktorokat írhatjuk egymás mellé, de tördelve, külön sorba is meg lehet adni őket

```
data Szemely = Diak String String Double
              | Tanar String String
  deriving (Show)
```

- egy Szemely elemtípusú lista kezdőértéke a következő lehet:

```
lsSz :: [Szemely]
lsSz = [Diak "Laci" "ELTE" 4.5, Tanar "Feri" "ELTE",
        Tanar "Mari" "Sapientia", Diak "Lori" "Sapientia" 7.5,
        Diak "Sari" "Sapientia" 8.75, Tanar "Zsuzsi" "ELTE"]
```

Algebrai adattípusok

1. feladat

Írjunk egy Haskell függvényt, amely egy `Szemely` típusú lista esetében meghatározza a 4.5-nél nagyobb jeggyel rendelkező diákok számát:

```
szamolDiak :: [Szemely] -> Int
szamolDiak ls = length $ filter auxF ls
  where
    auxF (Tanar _ _) = False
    auxF (Diak _ _ jegy) = jegy > 4.5
```

```
> szamolDiak lsSz
2
```

Mintaillesztéssel választottuk külön a Tanar, illetve a Diak értékekkel való műveletvégzést.

Algebrai adattípusok

A feladat megoldható a `foldl'` függvény használatával is, a kódsor a következő:

```
import Data.List (foldl')
szamolDiak_ :: [Szemely] -> Int
szamolDiak_ = foldl' op 0
  where
    op res k = case k of
      Tanar _ _ -> res
      Diak _ _ jegy -> if jegy > 4.5 then 1 + res
                       else res
```

Algebrai adattípusok

A következőkben előbb négy típuszinonimát definiálunk, majd egy BankSzamla típust, három értékkonstruktorral, amelyekben használjuk a típuszinonimaként megadott típusokat:

```
type KartyaSz = String
type Tulajdonos = String
type Cim = [String]
type FelhasznaloID = Int

data BankSzamla = BankKartya KartyaSz Tulajdonos Cim
                | Keszpenz
                | Szamla FelhasznaloID
                deriving (Show, Eq)
```

Konstans értékeket a következőképpen is létrehozhatunk:

```
sz1 = BankKartya "12321" "Kiss Antal" ["Mvh", "Romania"]
sz2 = Keszpenz
sz3 = Szamla 12
sz4 = BankKartya "54321" "Nagy Antal" ["Kv", "Romania"]
sz5 = BankKartya "98765" "Beres Antal" ["Mvh", "Romania"]
sz6 = Szamla 13

lsBsz :: [BankSzamla]
lsBsz = [sz1, sz2, sz3, sz4, sz5, sz6]
```


Algebrai adattípusok

2. feladat

Írjunk egy Haskell-függvényt, amely kiválogatja egy `BankSzamla` elemtípusú listából azokat a személyeket, akiknek értékkonstruktora `BankKartya`.

A feladatot háromféleképpen is megoldjuk. A `valogatSz` explicit rekurziót használ:

```
valogatSz :: [BankSzamla] -> [Tulajdonos]
valogatSz [] = []
valogatSz (k : ve) = case k of
    BankKartya _ tu _ -> tu : valogatSz ve
    _ -> valogatSz ve
```

Algebrai adattípusok

A `valogatSzF` a `foldr` függvényt alkalmazza. A `foldr` helyett alkalmazhattuk volna valamelyik `foldl` változatot is.

```
valogatSzF :: [BankSzamla] -> [Tulajdonos]
valogatSzF = foldr op []
  where
    op k res = case k of
      BankKartya _ tu _ -> tu : res
      _ -> res
```

A `valogatSzH` halmazműveleteket használ:

```
valogatSzH :: [BankSzamla] -> [Tulajdonos]
valogatSzH ls = [tu | BankKartya _ tu _ <- ls]

> valogatSz lsBsz
["Kiss Antal", "Nagy Antal", "Beres Antal"]
```

Algebrai adattípusok

3. feladat

Írjunk egy Haskell-függvényt, amely külön listákba teszi a BankKartya, a Keszpenz és a Szamla értékkonstruktorokra vonatkozó adatokat. A Keszpenz esetében csupán számoljuk meg, hogy hány készpénzkifizetés van.

```
valogatBK :: [BankSzamla] -> ([BankSzamla], [Int], [BankSzamla])
valogatBK ls = auxV ls [] [0] []
  where
    auxV [] bLs kLs sLs = (bLs, kLs, sLs)
    auxV (k : ve) bLs kLs sLs = case k of
      BankKartya {} -> auxV ve (k : bLs) kLs sLs
      Keszpenz -> auxV ve bLs [1 + head kLs] sLs
      Szamla _ -> auxV ve bLs kLs (k : sLs)
```

```
> valogatBK lsBsz
([BankKartya "98765" "Beres Antal" ["Mvh",...
```

A case-ben, a BankKartya értékkonstruktor esetében a `_ _ _` helyett a `{}` szimbólumokat használjuk, mert a `->` jobb oldalán egyetlenegy mezőértékkel sincs műveletvégzés.

Algebrai adattípusok

Az elegánsabb kiíratáshoz `mapM_-et` használunk, amelyet külön-külön alkalmazunk a tuple elemekre, amelyeket a `valogatBK` függvény visszatérési értékeként kaptunk meg:

```
foValogat :: [BankSzamla] -> IO ()
foValogat ls = do
  let (t1, t2, t3) = valogatBK ls
  putStrLn "Bankkartya adatok:"
  mapM_ print t1
  putStrLn "Keszpenzkifizetesek szama: "
  mapM_ print t2
  putStrLn "Szamla adatok:"
  mapM_ print t3

> foValogat lsBsz
Bankkartya adatok:
BankKartya "98765" "Beres Antal" ["Mvh","Romania"]
...
```

Kombinatorikai feladatok

4. feladat

Írjunk egy Haskell-függvényt, amely meghatározza az összes olyan m hosszúságú listát, amely egy bemeneti lista elemeiből képezhető.

- az `lGen_` halmazműveleteket alkalmaz, létrehozza azokat a listákat, amelyek első elemei rendre a bemeneti lista elemei lesznek, a többi elemet pedig rekurzívan generálja
- a rekurzív hívásban a függvény első paramétere az eredeti lista lesz, a második paramétert pedig minden egyes alkalommal csökkentjük eggyel

```
lGen_ :: (Eq a) => [a] -> Int -> [[a]]
```

```
lGen_ ls 0 = [[]]
```

```
lGen_ ls m = [k : ve | k <- ls, ve <- lGen_ ls (m-1)]
```

```
> lGen_ [0, 1] 4
```

```
[[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],[0,1,0,0],[0,1,0,1],[0,1,1,0],  
[0,1,1,1],[1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1],[1,1,0,0],[1,1,0,1],  
[1,1,1,0],[1,1,1,1]]
```

```
> lGen_ "ab" 3
```

```
["aaa","aab","aba","abb","baa","bab","bba","bbb"]
```

Kombinatorikai feladatok

- a következő `lGen`-ben módosítjuk a listaelemek generálási módjának a sorrendjét
- először generáljuk a `ve` listaelemeket, és csak ezután adjuk meg a `k` értékének a generálási módját:

```
lGen :: (Eq a) => [a] -> Int -> [[a]]  
lGen ls 0 = [[]]  
lGen ls m = [k : ve | ve <- lGen ls (m-1), k <- ls]
```

- a módosítás a listaelemek más sorrendjét fogja eredményezni:

```
> lGen "ab" 3  
["aaa", "baa", "aba", "bba", "aab", "bab", "abb", "bbb"]
```

- az algoritmus hatékonyságát is javítottuk, amelyet a következő oldalon megadott `foGen` és a `foGen_` időigényei közötti lényeges különbség jelez

Kombinatorikai feladatok

```
foGen :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen ls m = do
    let rLs = lGen ls m
    print $ last rLs
```

```
> :set +s
> foGen [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(0.67 secs, 352,405,392 bytes)
```

```
foGen_ :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen_ ls m = do
    let rLs = lGen_ ls m
    print $ last rLs
```

```
> foGen_ [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(3.93 secs, 3,120,641,320 bytes)
```

- a foGen, illetve foGen_ csak a kigenerált lista utolsó elemét írja ki, így a függvények időigényének értékét nem befolyásolta a listaelemek kiírásának időigénye

n elem m -ed rendű kombinációi

5. feladat

Írjunk egy Haskell-függvényt, amely meghatározza n elem m -ed rendű kombinációit.

```
komb :: (Ord a) => [a] -> Int -> [[a]]
komb ls 0 = [[]]
komb ls m = [k : ve |
              ve <- komb ls (m-1),
              k <- ls, feltKomb k ve]

feltKomb :: (Ord a) => a -> [a] -> Bool
feltKomb x [] = True
feltKomb x (k : ve)
  | k <= x = False
  | otherwise = feltKomb x ve

> komb [4, 1, 7, 9] 3
[[1,4,7],[1,4,9],[4,7,9],[1,7,9]]

> komb "wxyz" 2
["wx","wy","xy","wz","xz","yz"]
```


n elem m -ed rendű kombinációi

- a komb az lGen-en alapszik
- a komb paraméterként egy listát és az m értéket kapja meg, ahol a lista elemszáma adja a feladat megfogalmazásában szereplő n értéket
- mivel a kigenerált listák nem mindegyike felel meg a feladat kritériumának, ezért egy tesztlő feltKomb függvényt is alkalmazunk
- a feltKomb kimenete akkor lesz True ha a bemeneti lista elemei szigorúan növekvő sorrendben vannak
- például a fenti bemenet esetében az eredmény lista nem tartalmazhatja az [1, 4, 7], [1, 7, 4], [7, 4, 1] stb. mindegyikét, csak az [1, 4, 7]-t fogja tartalmazni
- a feltKomb-ban, ha módosítjuk a feltételt $k \geq x = \text{False}$ -ra, akkor a [7, 4, 1] fog szerepelni az eredménylistában
- a következő kódsor a feltKomb egy kompaktabb változata:

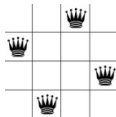
```
feltKomb_ :: (Ord a) => a -> [a] -> Bool
feltKomb_ x = all (aux x)
  where
    aux x k = k > x
```

Az n királynő feladat

6. feladat

Írjunk egy Haskell-függvényt, amely elhelyez egy sakktáblán 8 királynőt úgy, hogy azok ne üssék egymást. Általánosan is oldjuk meg a feladatot, azaz helyezzünk el egy $n \times n$ -es sakktáblán n királynőt úgy, hogy azok ne üssék egymást.

- a gyakorlatban egy $n \times n$ -es sakktáblán, ha el akarunk helyezni n királynőt úgy, hogy azok ne üssék egymást, ez azt fogja jelenteni, hogy nem tehetjük őket ugyanabba a sorba, oszlopba, illetve átlóra
- a megoldás listák kigenerálását fogja jelenti:
 - a listaelemek egész számok lesznek
 - az első listaelem az első sorban levő királynő oszlop-pozícióját fogja jelenteni, a második listaelem a második sorban levő királynő oszlop-pozícióját jelenti, és így tovább
- egy 4×4 -es sakktáblán **[3, 1, 4, 2]** helyes megoldás azt fogja jelenteni, hogy a királynőt az első sorban a harmadik, a második sorban az első, a harmadik sorban a negyedik és a negyedik sorban a második oszlopba tettük:



Az n királynő feladat

```
kiralyno :: Int -> [[Int]]
kiralyno n = auxK n n
  where
    auxK :: Int -> Int -> [[Int]]
    auxK n 0 = [[]]
    auxK n m = [k : ve |
                  ve <- auxK n (m-1),
                  k <- [1..n], feltKir k ve ]

feltKir :: Int -> [Int] -> Bool
feltKir x ls = auxFeltK 1 x ls
  where
    auxFeltK :: Int -> Int -> [Int] -> Bool
    auxFeltK i x [] = True
    auxFeltK i x (k : ve)
      | k == x = False
      | abs (x - k) == i = False
      | otherwise = auxFeltK (i + 1) x ve

> kiralyno 5
[[4,2,5,3,1],[3,5,2,4,1],[5,3,1,4,2],[4,1,3,5,2],[5,2,4,1,3],
 [1,4,2,5,3],[2,5,3,1,4],[1,3,5,2,4],[3,1,4,2,5],[2,4,1,3,5]]
```

Az n királynő feladat

- az algoritmus a kombinációkat meghatározó függvény egy változata lesz
- a feltételek teljesülését vizsgáló függvény a `feltKir` lesz
- ennél a feladatnál a listaelemek nem kell növekvő sorrendben legyenek, elég ha különböznek
- az eredménylisták elemeit akkor határozzuk meg, amikor *jövünk vissza* a rekurzióból, például az 5x5-ös sakktábla esetében a `[4,2,5,3,1]` lista elsőként meghatározott eleme az 1 lesz, majd eléje kerül a 3, majd az elé az 5-ös, és így tovább
- az átlók menti ütközés kizárását a `abs(x - k) == i = False` feltétel biztosítja
- ha igaz a fenti feltétel az azt jelenti, hogy az `x` elem a már kigenerált listabeli elemek közül a `k`-val, amely `i` pozíciónyira helyezkedik el az `x`-től, az átlók mentén, ütközni fog, például a 6x6-os sakktábla esetében a `[2,6,3]` lista elejére nem lehet 1-est tenni, mert az átlós ütközést eredményezne a 2-vel (`abs(2 - 1) == 1`), de 4-est sem lehet tenni, mert az a 6-tal okozna átlós ütközést (`abs(6 - 4) == 2`)

Az n királynő feladat

a feltKir tömörebb változata a következő:

```
feltKir_ :: Int -> [Int] -> Bool
feltKir_ x ls = auxFeltK x ls
  where
    auxFeltK :: Int -> [Int] -> Bool
    auxFeltK x ls = all (aux x) $ zip [1..] ls
      where
        aux x (i, k) = (k /= x) && (abs(x - k) /= i)
```

Az n királynő feladat

A feladat megoldásait elegánsabban is meg tudjuk jeleníteni a képernyőn, a `foKiralyno` meghívásával:

```
kiirSor :: Int -> Int -> IO()
kiirSor n k = do
    mapM_ (auxF k) [1..n]
    putStrLn ""
    where
        auxF k x =
            if k == x then putStr "Q " else putStr ". "

kiirTabla :: Int -> [Int] -> IO()
kiirTabla n ls = do
    mapM_ (kiirSor n) ls
    putStrLn ""

foKiralyno :: Int -> IO()
foKiralyno n = mapM_ (kiirTabla n) $ kiralyno n

> foKiralyno 5
```

Kombinatorikai feladatok

- észrevehető az algoritmikai hasonlóság a 10. előadáson vett lGen, komb, illetve kiralyno függvények között
- ezek megadhatóak általánosan: különböző feltételeket definiáló függvényt írunk a megfelelő listák kiszűrésére, a generálást pedig a rekurzio fogja végezni:

```
rekurzio :: Int -> Int -> (Int -> [Int] -> Bool) -> [[Int]]
rekurzio n 0 fg = [[]]
rekurzio n m fg = [k : ve | ve <- rekurzio n (m-1) fg, k <- [1..n], fg k ve]
```

```
lGenR :: Int -> Int -> [[Int]]
lGenR n m = rekurzio n m (\k ve -> True)
```

```
> lGenR 2 3
[[1,1,1],[2,1,1],[1,2,1],[2,2,1],[1,1,2],[2,1,2],[1,2,2],[2,2,2]]
```

```
permutacioR :: Int -> [[Int]]
--permutacioR n = rekurzio n n (\k ve -> notElem k ve)
permutacioR n = rekurzio n n notElem
```

```
> permutacioR 3
[[3,2,1],[2,3,1],[3,1,2],[1,3,2],[2,1,3],[1,2,3]]
```

Kombinatorikai feladatok

```
feltKomb :: (Ord a) => a -> [a] -> Bool
feltKomb x = all (aux x)
```

```
  where
    aux x k = k > x
```

```
kombR :: Int -> Int -> [[Int]]
```

```
kombR n m = rekurzio n m feltKomb
```

```
> kombR 3 2
[[1,2],[1,3],[2,3]]
```

```
feltKir :: Int -> [Int] -> Bool
```

```
feltKir x ls = auxFeltK 1 x ls
```

```
  where
```

```
    auxFeltK :: Int -> Int -> [Int] -> Bool
```

```
    auxFeltK i x ls = all (aux x) $ zip [1..] ls
```

```
      where
```

```
        aux x (i, k) = k /= x && (abs(x - k) /= i)
```

```
kiralynoR :: Int -> [[Int]]
```

```
kiralynoR n = rekurzio n n feltKir
```

```
> kiralynoR 6
[[5,3,1,6,4,2],[4,1,5,2,6,3],[3,6,2,5,1,4],[2,4,6,1,3,5]]
```