Funkcionális programozás
8. előadás

MÁRTON Gyöngyvér

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

2025, tavaszi félév

Miről volt szó?

- operátorok, függvények listákon:
  `splitAt, notElem, concat, repeat, replicate, cycle, iterate, any, all`
- feladatok: madárnyelv, maximum elem és pozíciói, válogatások, statisztikai számítások
- Haskell projektek: Benford törvénye

Miről lesz szó?

- a @ minta (as-pattern)
- rendezések:
  - `sortOn`,
  - összefésülő rendezés (merge sort)
  - beszúró rendezés (insertion sort),
  - gyorsrendezés (quick sort),
- hajtogatások (fold operations)
- Haskell monádok
- adatbevitel
- szövegállományok

A @ minta (as-pattern)

Egyszerű jelölést tesz lehetővé, ha egy lista egészére, illetve, ha csak egy részére (első elemére, végére) szeretnénk hivatkozni.

1. feladat

Határozzuk meg két rendezett lista összefésült értékét.

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ls = ls
merge ls [] = ls
merge ls1@(k1 : ve1) ls2@(k2 : ve2)
    | k1 < k2 = k1 : merge ve1 ls2
    | otherwise = k2 : merge ls1 ve2

> merge [1, 5, 10, 15] [2, 4, 6, 8, 20, 22, 24]
[1,2,4,5,6,8,10,15,20,22,24]
```

A függvényben a harmadik feltétel megadásakor a @ mintát használva elértük, hogy az egyenlőség jobb oldalán egyaránt tudunk hivatkozni a listák egészére, az `ls1`-re vagy az `ls2`-re, illetve a listák első elemére, a `k1`-re vagy `k2`-re és a listák végére, a `ve1`-re vagy a `ve2`-re.

Összefésülő rendezés

- az összefésülő rendezésben (merge sort) a `merge` két rendezett lista összefésülését végzi, amelyet korábban adtunk meg
- az eredeti listát két részlistára osztjuk, a `bLs` fogja tartalmazni a bemeneti lista első felét, a `jLs` pedig a második felét
- mindkét részlistát rekurzívan, a `mergeS` függvénnyel rendezzük, majd a `merge` függvénnyel összefésüljük

```
mergeS :: (Ord a) => [a] -> [a]
mergeS  [] = []
mergeS [k] = [k]
mergeS ls = merge bLista jLista
    where
    db = div (length ls) 2
    bLista = mergeS (take db ls)
    jLista = mergeS (drop db ls)

> mergeS [3,12,6,7,5,9,10,2,10,1]
[1,2,3,5,6,7,9,10,10,12]
```

Beszúró rendezés

A beszúró rendezést (insertion sort) két függvénnyel valósítjuk meg, az első az `ins` függvény, amely beszúr egy elemet egy rendezett listába, a második az `insertionS`, amely az `ins` függvényt alkalmazva meghatározza a rendezett sorrendet.

Adott elem beszúrása rendezett listába:

```
ins :: (Ord a) => a -> [a] -> [a]
ins x [] = [x]
ins x (k : ve)
    | x > k = k : ins x ve
    | otherwise = x : k : ve

> ins 'h' "abcdeklmno"
"abcdehklmno"

> ins "racionalis" ["egesz", "komplex", "valos"]
["egesz","komplex","racionalis","valos"]
```

Beszúró rendezés

Az insertionS úgy rendezi a lista elemeit, hogy az első elemet, a `k`-t beszúrja a rendezett `ve` listába, ahol a `ve` lista rendezését rekurzívan az `insertionS` függvény végzi.

```
insertionS :: (Ord a) => [a] -> [a]
insertionS [] = []
insertionS (k : ve) = ins k (insertionS ve)

> insertionS [3,12,7,8,4,5,11]
[3,4,5,7,8,11,12]

> insertionS "acbdeklhmno"
"abcdehklmno"
```

Gyorsrendezés

- a gyorsrendezést (quick sort) halmazkifejezésekkel végezzük
- kiválogatjuk a `k`-nál kisebb elemeket a `kLs`-be, illetve a `k`-nál nagyobb elemeket az `nLs`-be
- mindkét részlistát a `quickS` függvénnyel rekurzívan rendezzük
- a kapott részlistákat összefűzzük, a két rendezett lista közé helyezve a `k` elemet

```
quickS :: (Ord a) => [a] -> [a]
quickS [] = []
quickS (k : ve) = quickS kLista ++ [k] ++ quickS nLista
    where
    kLista = [x | x <- ve, x < k]
    nLista = [x | x <- ve, x >= k]

> quickS [7,5,9,10,13,2,3,11]
[2,3,5,7,9,10,11,13]
```

Hajtogatások (fold operations)

- a *hajtogatásokat*, a *folding* műveleteket végző függvények magasabb rendű függvények
- listaelemeken végeznek kiértékeléseket explicit rekurzió nélkül
- a `foldl` függvény háromparaméteres, az első egy bináris operátor, a második egy kezdőérték, a harmadik pedig egy lista, típusdeklarációja a következő:

```
foldl :: Foldable t => (a -> b -> a) -> a -> t b -> a
```

  - balra asszociatív: balról jobbra haladva dolgozza fel a listaelemeket
  - rendre alkalmazza a bináris operátort úgy, hogy a bal oldali operandusa a `foldl` függvény második paramétere, jobb oldali operandusa az aktuális listaelem lesz
  - a második paraméter minden listaelem feldolgozása után felülíródik a bináris művelet eredményével
  - a függvény által meghatározott érték a második argumentumban kiszámolt érték lesz
  - akkor számol, amikor megy be a rekurzióba, ezért a rekurzió legalsó szintjén már meg van határozva a végső eredmény.

Hajtogatások (fold operations)

A lista elemeinek összege:

```
> foldl (+) 0 [1..10]
55
```

10 faktoriális:

```
> foldl (*) 1 [1..10]
3628800
```

A következő kiértékelés esetében nem csak az eredményt tüntetjük fel, hanem a számítási sorozatot is:

```
> foldl (-) (-2) [34,6,12,65,8,11,23]
-161

(((((((-2 - 34) - 6) - 12) - 65) - 8) - 11) - 23)
```

Hajtogatások (fold operations)

A `foldl` függvény implementációja, ahol a mintaillesztés alkalmazása miatt módosítottuk a típusdeklarációt, a harmadik paraméter típusát lista típusra cseréltük:

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl op res [] = res
myFoldl op res (k : ve) = myFoldl op (op res k) ve

> ls = [9,6,12,65,8,11,23]
> myFoldl max (head ls) ls
65
```

Ha a listaelemeket k_1, k_2, k_3,..., k_n-nel jelöljük, és a `res` a kezdőérték, akkor általánosan a `foldl` hívásai során a következőképpen kerülnek sorra a műveletek, ahol az operátort infix formába írtuk:

$$(\ldots(((res\ op\ k_1)\ op\ k_2)\ op\ k_3)\ \ldots\ op\ k_n).$$

Hajtogatások (fold operations)

Módosítjuk a `myFoldl` függvény kódsorát, kiíratjuk a részeredményeket:

```
myFoldlIr op res [] = return res
myFoldlIr op res (k : ve) = do
    putStr $ show (op res k) ++ " "
    temp <- myFoldlIr op (op res k) ve
    return temp

> ls = [9,6,12,65,8,11,23]
> myFoldlIr max (head ls) ls
9 9 12 65 65 65 65 65
```

Megjegyzések:

- a `do` blokk keretén belül az `<-` operátort használva lehetőség van arra, hogy a `myFoldlIr` függvény által meghatározott értéket lekérjük, amely *nem tiszta* függvény
- az `<-` operátort most másképp használjuk, mint a halmazkifejezések során, a Haskellnek azonban a kontextusból meg tudja állapítani, hogyan járjon el

Hajtogatások (fold operations)

A `foldr` a `foldl` függvény párja, típusdeklarációja a következő:

```
foldr :: Foldable t => (b -> a -> a) -> a -> t b -> a
```

- jobbról asszociatív: a listaelemeket jobbról balra dolgozza fel
- egy bináris operátort alkalmaz egy kezdeti érték és a megfelelő listaelemek között
- a függvény első paramétere a bináris operátor, második paramétere pedig a kezdeti érték lesz
- a megfelelő listaelemeket a harmadik paraméterként megadott listából veszi
- a bináris operátor bal oldali operandusa az aktuális listaelem, jobb oldali operandusa pedig a `foldr` függvény második paramétere lesz
- a kezdőérték csak a legutolsó rekurzív híváskor kerül feldolgozásra
- a részértékek akkor kerülnek meghatározásra, amikor a függvény jön vissza a rekurzióból
- a végső kifejezés meghatározására csak akkor kerülhet sor, amikor minden rekurzív függvényhívás kiértékelődött

Hajtogatások (fold operations)

A lista elemeinek összege:

```
> foldr (+) 0 [1..10]
55
```

Az első 5 páros szám szorzata:

```
> foldr (*) 1 [2,4..10]
3840
```

A következő kiértékelés esetében nem csak az eredményt tüntetjük fel, hanem a számítási sorozatot is:

```
> foldr (-) (-2) [34,6,12,65,8,11,23]
-3

34 - (6 - (12 - (65 - (8 - (11 - (23 - (-2)))))))
```

Hajtogatások (fold operations)

A `foldr` függvény implementációja, ahol a mintaillesztés alkalmazása miatt módosítottuk a típusdeklarációt, a harmadik paraméter típusát lista típusra cseréltük:

```
myFoldr :: (b -> a -> a) -> a -> [b] -> a
myFoldr op res [] = res
myFoldr op res (k : ve) = op k $ myFoldr op res ve
```

```
> ls = [9,6,12,65,8,11,23]
> myFoldr max (head ls) ls
65
```

Ha a listaelemeket k_1, k_2, k_3,..., k_n-nel jelöljük, és a `res` a kezdőérték, akkor általános esetben a `foldr` a következőképpen értékeli ki a műveleteket:

$$k_1 \text{ op } (k_2 \text{ op } (k_3 \text{ op } (\ldots (k_n \text{ op res}) \ldots))).$$

Hajtogatások (fold operations)

Módosítjuk a `myFoldr` függvény kódsorát, kiíratjuk a részeredményeket:

```
myFoldrIr op res [] = return res
myFoldrIr op res (k : ve) = do
    temp <- myFoldrIr op res ve
    putStr $ show temp ++ " "
    return $ op k temp
```

A következő lekérdezések eredményeit hasonlítsunk össze a `myFoldlIr` hasonló paraméterezésével kapott eredményeivel:

```
> myFoldrIr (+) 0 [1..10]
0 10 19 27 34 40 45 49 52 54 55

> myFoldrIr (*) 1 [2,4..10]
1 10 80 480 1920 3840

> ls = [9,6,12,65,8,11,23]
> myFoldrIr max (head ls) ls
9 23 23 23 65 65 65 65
```

Hajtogatások (fold operations)

A lusta kiértékelési stratégia miatt a `foldl` hatékonysága nem elfogadható, egy 1 millió elemszámú lista elemei összegének a meghatározása több másodpercig is eltarthat:

```
> :set +s
> foldl (+) 0 [1..10000000]
50000005000000
(3.15 secs, 1,612,359,432 bytes)
```

Valamivel jobb időt kapunk, ha a `foldr` változattal dolgozunk:

```
> foldr (+) 0 [1..10000000]
50000005000000
(2.32 secs, 1,615,360,800 bytes)
```

- A fenti hatékonysági problémák kiküszöbölésére a Haskell több `foldl` implementációt vezet be, ezekben szigorú (strict) kiértékelési stratégiát tesz lehetővé: `foldl'`, `foldl1`, `foldl1'`
- a `foldl'`, illetve `foldl1'` a `Data.List`-ben találhatók
- a `foldl1`, illetve `foldl1'` változatok esetében nem kell megadni kezdőértéket, ez mindig a lista első eleme lesz

Hajtogatások (fold operations)

Összehasonlításképpen időket mérhetünk:

```
> import Data.List (foldl', foldl1')

> foldl' (+) 0 [1..10000000]
50000005000000
(0.23 secs, 880,062,880 bytes)

> foldl1 (+) [1..10000000]
50000005000000
(2.75 secs, 1,612,359,336 bytes)

> foldl1' (+) [1..10000000]
50000005000000
(0.43 secs, 880,059,088 bytes)
```

Következésképpen a `foldl` helyett mindig a `foldl'`, vagy amikor lehetséges a `foldl1'` változatokat használjuk.

Hajtogatások (fold operations)

2. feladat

Írjunk egy Haskell-függvényt, amely a `foldl'`-t alkalmazva meghatározza a paraméterként megadott lista legnagyobb elemét, illetve a legnagyobb elem előfordulási pozícióit.

```
import Data.List (foldl')
myMaximum :: (Num b, Ord a) => [a] -> (a, [b])
myMaximum ls = (max, mLs)
    where
    (mLs, _, max) = foldl' op res ls
    res = ([], 0, head ls)
    op tRes k
        | k == m = (p : pLs, p + 1, m)
        | k < m = (pLs, p + 1, m)
        | k > m = ([p], p + 1, k)
            where
            (pLs, p, m) = tRes

> myMaximum [3, 5, 6, 10, 3, 10, 7, 6, 10, 4, -10, 10]
(10,[11,8,5,3])
```

Megjegyzések

- a `foldl'` függvény második paraméterének típusa egy háromelemű tuple
- a meghívásra kerülő bináris operátor egyik operandusa egy háromelemű tuple a másik az aktuális listaelem
- a háromelemű tuple elemei rendre a következő értékeket jelölik: a maximum elem pozíciói, az aktuális pozíció, illetve az aktuális maximum elem

Hajtogatások (fold operations)

A következő myMaximum_ függvény is a legnagyobb elemet, illetve a legnagyobb elem pozícióit határozza meg, csak explicit rekurziót használ:

```
myMaximum_ :: (Num b, Ord a) => [a] -> (a, [b])
myMaximum_ ls = (max, mLs)
    where
    (mLs, _, max) = myMaxAux ([], 0, head ls) ls
    myMaxAux tRes [] = tRes
    myMaxAux tRes (k : ve)
        | k == m = myMaxAux (p : pLs, p + 1, m) ve
        | k < m = myMaxAux (pLs, p + 1, m) ve
        | k > m = myMaxAux ([p], p + 1, k) ve
            where
            (pLs, p, m) = tRes
```

Hajtogatások (fold operations)

3. feladat

Írjunk egy Haskell-függvényt, amely megvizsgálja, hogy egy adott szám prímszám-e vagy sem, majd alkalmazzuk a függvényt a prímszámok listájának kigenerálására. Az implementáció során alkalmazzuk a `foldr` függvényt.

```
primTeszt_ :: Integer -> Bool
primTeszt_ n = n > 1 && foldr op True [3,5..]
    where
    op k resB = k * k > n || (mod n k /= 0 && resB)

> primTeszt_ 1789
True
```

Megjegyzések:

- a `foldr` harmadik paramétere a páratlan számok végtelen listája lesz, de ez nem eredményez végtelen számítási folyamatot, mert az `op`-ben a `||` operátort használtuk
- amikor a `k * k > n` feltétel eredménye `True` lesz, nincs már szükség a `||` jobb oldalán álló kifejezés kiértékelésére
- az oszthatóságok vizsgálata a `k-2`, `k-4` stb. értékekkel kezdődik (visszafelé haladva a páratlan számok listájában)

Hajtogatások (fold operations)

4. feladat

Írjunk egy Haskell-függvényt, amely kigenerálja az első n prímszám listáját. Az implementáció során alkalmazzuk a `foldr` függvényt.

A prímszámok listáját a `primLista` függvény generálja ki, amely a hatékonyabb `primTeszt`-el végzi a prímek tesztelését:

```
primTeszt :: Integer -> Bool
primTeszt n = n > 1 && foldr op True primLista
    where
    op k resB = k * k > n || (mod n k /= 0 && resB)

primLista :: [Integer]
primLista = 2 : filter primTeszt [3,5..]

> take 10 primLista
[2,3,5,7,11,13,17,19,23,29]
> last $ take 10000 primLista
104729
(1.61 secs, 253,024,968 bytes)
```