Funkcionális programozás
5. előadás

MÁRTON Gyöngyvér

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

2024, tavaszi félév

Miről volt szó?

- `case ... of`
- függvénykompozíció
- függvénykiértékelés a $ szimbólummal
- a $ és a . szimbólumok
- a Haskell kiértékelési stratégiája
- operátorok, függvények listákon: null, init, last, sum, map,
- feladatok:
  - hexa szimbólumok
  - négyzetszámok, összetett számok, prímszámok tesztelése

Miről lesz szó?

- operátorok, függvények listákon: filter, reverse, take, takeWhile, drop, dropWhile, elem, zip, splitAt, notElem, concat, repeat, replicate, cycle, iterate, any, all
- Haskell, a mellékhatások kezelése (side effects)
- kiíratási műveletek, a `mapM_` használata
- feladatok:
  - magánhangzók, mássalhangzók szűrése

Függvények listákon

1. feladat

Írjunk Haskell-függvényt, amely meghatározza $g^x \pmod p$ értékét minden `x = 1,2,..., p-1` értékre.

```
hatvSz :: (Integral a) => a -> a -> [a]
hatvSz g p = map (aux g p) [1..p - 1]
    where
    aux :: (Integral a) => a -> a -> a -> a
    aux g p x = mod (g ^ x) p

> hatvSz 2 11
[2,4,8,5,10,9,7,3,6,1]
```

Függvények listákon

`filter :: (a -> Bool) -> [a] -> [a]`
Kiválasztja a lista azon elemeit melyek eleget tesznek egy adott feltételnek.

```
myFilter1 :: (a -> Bool) -> [a] -> [a]
myFilter1 fg [] = []
myFilter1 fg (k : ve)
    | fg k = k : myFilter1 fg ve
    | otherwise = myFilter1 fg ve

myFilter2 :: (a -> Bool) -> [a] -> [a]
myFilter2 fg ls =
    let
        k = head ls
        ve = tail ls
    in
        if null ls then [] else
            if fg k then k : myFilter2 fg ve
            else myFilter2 fg ve

> import Data.Char
> myFilter2 isUpper "Erdelyi Karpat Egyesulet"
"EKE"
```

Függvények listákon

`reverse :: [a] -> [a]`
Megfordítja a lista elemeit.

```
-- nem hatékony
myReverse1 :: [a] -> [a]
myReverse1 [] = []
myReverse1 (k : ve) = myReverse1 ve ++ [k]

-- nem hatékony
myReverse2 :: [a] -> [a]
myReverse2 ls =
    if null ls then []
    else myReverse2 (tail ls) ++ [head ls]

-- ez a hatékony!!
myReverse3 :: [a] -> [a]
myReverse3 ls = auxRev ls []
    where
    auxRev [] res = res
    auxRev (k : ve) res = auxRev ve (k : res)
```

Függvények listákon

- a `reverse3` kódsorát úgy módosítjuk, hogy az épülő listát minden egyes rekurzív hívás előtt kiíratjuk
- a `do` blokk keretén belül két műveletsort adunk meg,
- az elsőnek az lesz a szerepe, hogy a `print` függvényt alkalmazva kiíratást végezzen
- a másodikban rekurzív függvényhívásra kerül sor,
- a triviális esetben a `return` segítségével jelezzük, hogy mi a függvény kimeneti értéke.

```
myReverseIr ls = auxReverse ls []
    where
    auxReverse [] res = return res
    auxReverse (k : ve) res = do
        putStrLn (k : res)
        auxReverse ve (k : res)
```

Függvények listákon

`take :: Int -> [a] -> [a]`
Visszatéríti a második argumentumként megadott lista első `n` elemét, ahol `n` a függvény első argumentuma.

```
myTake :: Int -> [a] -> [a]
myTake n [] = []
myTake n (k : ve)
    | n == 0 = []
    | otherwise = k : myTake (n-1) ve

> myTake 3 ["abc", "efgh", "ijklmn", "op", "qrst"]
["abc","efgh","ijklmn"]

> myTake 10 ["abc", "efgh", "ijklmn", "op", "qrst"]
["abc","efgh","ijklmn","op","qrst"]
```

Függvények listákon

`takeWhile :: (a -> Bool) -> [a] -> [a]`
Visszatéríti a második argumentumként megadott lista azon prefixét, amelyben az elemek eleget tesznek a feltételnek.

```
> takeWhile even [2,4,6,8,9,10,12,14]
[2,4,6,8]

myTakeWhile1 :: (a -> Bool) -> [a] -> [a]
myTakeWhile1 fg [] = []
myTakeWhile1 fg (k : ve)
    | fg k = k : myTakeWhile1 fg ve
    | otherwise = []

> import Data.Char
> myTakeWhile1 isDigit "1234aedbcde567fgh"
"1234"

> length $ myTakeWhile (/= 0) [1 / (2 ^ i) | i <-[1..]]
```

Függvények listákon

`drop :: Int -> [a] -> [a]`
Kitörli a második argumentumként megadott lista első `n` elemét, ahol `n` a függvény első argumentuma.

```
> drop 3 ["abc", "efgh", "ijklmn", "op", "qrst"]
["op","qrst"]
```

```
myDrop :: Int -> [a] -> [a]
myDrop n [] = []
myDrop n (k : ve)
    | n == 0 = (k : ve)
    | otherwise = myDrop (n-1) ve
```

Függvények listákon

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

Kitörli a második argumentumként megadott lista azon prefixét, amelyben az elemek eleget tesznek a feltételnek.

```
> dropWhile even [2,4,6,8,9,10,12,14]
[9,10,12,14]
```

```
myDropWhile :: (a -> Bool) -> [a] -> [a]
myDropWhile fg [] = []
myDropWhile fg (k : ve)
    | fg k = myDropWhile fg ve
    | otherwise = k : ve
```

Függvények listákon

```
elem ::(Eq a) => a -> [a] -> Bool
```

Megvizsgálja, hogy egy adott elem szerepel-e a listaelemek között.

```
> elem 'a' "Sapientia University"
True
```

```
myElem :: (Eq a) => a -> [a] -> Bool
myElem x [] = False
myElem x (k : ve)
    | x == k = True
    | otherwise = myElem x ve
```

- annak a megállapítása, hogy egy elem *nem* szerepel a listaelemek között, csak akkor lehetséges, ha megvizsgáltuk az összes listaelemet
- a *nem* válasz meghatározása időigényesebb, mint az *igen* válasz megállapítása.

Függvények listákon

```
zip :: [a] -> [b] -> [(a, b)]
```

A bemeneti két lista alapján elempárokból álló listát hoz létre. Az új lista elemszámát a rövidebb lista elemszáma határozza meg. A bemeneti két lista típusa nem kell megegyezzen.

```
> zip ["abc", "efg"] [1,2,3,5,6,7]
[("abc",1),("efg",2)]
```

```
myZip :: [a] -> [b] -> [(a, b)]
myZip [] ls = []
myZip ls [] = []
myZip (k1 : ve1) (k2 : ve2) = (k1, k2) : myZip ve1 ve2
```

Mellékhatások -side effects

- Haskell-ben csak bizonyos körülmények között megengedett azoknak műveleteknek az elvégzése amelyek *mellékhatással* járnak
- mellékhatást okoz ha:
  - egy változó értékét módosítjuk,
  - beolvasást, kiíratást végzünk
  - hibakezelést végzünk, stb.
- a funkcionális paradigma nem engedi meg a mellékhatások előfordulását
- olvasás, írás, hibakezelés esetében nem lehet elkerülni a mellékhatásokkal járó műveleteket
- a Haskell külön kiértékelési stratégiát, illetve jelölésmódot alkalmaz, hogy megkülönböztesse a funkcionális paradigma szerint megírt *tiszta/pure* függvényt a mellékhatásokat okozó függvénytől, így lehetővé teszi, hogy kezelhetőek legyenek a mellékhatások
- mellékhatást eredményező műveleteket külön blokkban, egy `do` blokkban tudunk megadni,
- a do blokkban a `let` kulcsszó használatával pedig jelezni tudjuk, ha tiszta funkcionális stílusban írt függvény kiértékelése következik,
- az előadásban megadott függvények közül, kivéve a `myReverseIr` függvényt, mindegyik pure function volt,
- a Haskell `main` függvénye nem pure function

Haskell előnyök/hátrányok

Előnyök:

- a tiszta függvényeket könnyű megérteni, a típusszignatúra alapján pedig könnyedén lehet a működésükre is következtetni,
- a tiszta függvények nem végeznek állapotmódosítást,
- a funkcionális nyelvek a függvényeket értékként kezelik és paraméterként is használják

Hátrányok:

- a tiszta és mellékhatással járó függvények együttes használata csökkenti a kód olvashatóságát
- a rekurzió a hatékonyság csökkentését eredményezheti

!!!!!!

- A Facebook a levélszemét-szűrő rendszerét Haskell-ben implementálta.
- A Whatsapp 50 számítógépes szakember segítségével szolgálja ki a 900 millió felhasználóját, mert az Erlang funkcionális programozási nyelvet használja a párhuzamosítható műveletek megvalósításakor.

Kiíratási műveletek

2. feladat

Írjunk Haskell-függvényt, amely külön sorokba írja egy `String` típusú elemekből álló lista elemeit.

```
ls1 = ["Dell", "HP", "ASUS", "Lenovo", "Toshiba"]

myShow1 :: [String] -> String
myShow1 [] = ""
myShow1 (k : ve) = k ++ "\n" ++ myShow1 ve

fugvKiir1 :: [String] -> IO ()
fugvKiir1 ls = do putStr $ myShow1 ls

fugvKiir2 :: [String] -> IO ()
fugvKiir2 ls = mapM_ putStrLn ls
-- > fugvKiir2 ls1
```

Kiíratási műveletek

- az alkalmazott `mapM_` függvény működése hasonló a `map`-hez
- `mapM_` és `map` közötti különbség:
  - a `map` csak olyan függvényt kaphat paraméterként, amely függvénykiértékelést végez, azaz *tiszta* függvényt,
  - a `mapM_`-nek olyan függvényt kell paraméterként megadni, amely *utasításokat* hajt végre

```
> ls = [12, 45, 67, 89]
> mapM_ print ls
> map (^ 2) ls
```

Kiíratási műveletek

Használni fogjuk a `sortOn` függvényt megadott kritérium szerinti rendezéshez:

```
> import Data.List
>  sortOn fst [("Mari", 7), ("Kata", 8), ("Anti", 10)]
[("Anti",10),("Kata",8),("Mari",7)]
```

Alkalmas komplex számok rendezéséhez is:

```
import Data.Complex

> lsC = [(-3.5) :+ 1.2, 3.2 :+ 1.1, 0.75 :+ 2.3]
> sortOn (realPart.abs) lsC
[0.75 :+ 2.3,3.2 :+ 1.1,(-3.5) :+ 1.2]
```

Kiíratási műveletek

3. feladat

Írjunk egy Haskell-függvényt, amely egymás alá írja a bemeneti listában található számokat, illetve a számok négyzetgyökét.

```
foNegyzet :: (Show a, Floating a) => [a] -> IO ()
foNegyzet ls = do
    let gyLs = [(i, sqrt i) | i <- ls]
    mapM_ myPrint gyLs

myPrint :: (Show a, Show b) => (a, b) -> IO ()
myPrint (t1, t2) = do
    putStrLn $ show t1 ++ " negyzetgyoke: " ++ show t2

> foNegyzet [12, 4, 56, 112]
```

Kiíratási műveletek

4. feladat

Modósítsuk az előző Haskell-függvényt, úgy hogy rendezzük az eredményt a négyzetgyökök alapján.

```
import Data.List
foNegyzetR :: (Show b, Floating b, Ord b) => [b] -> IO ()
foNegyzetR ls = do
    let gyLs = [(i, sqrt i) | i <- ls]
    let rLs = sortOn fst gyLs
    mapM_ myPrint rLs
```

Függvények listákon

A következők további listákat kezelő függvények:

`splitAt :: Int -> [a] -> ([a], [a])`
A megadott indexérték alapján a bemeneti listát két listára osztja. Az eredmény egy tuple típusú érték lesz.

```
> splitAt 4 [1,2,3,4,5,6,7,8,9]
([1,2,3,4],[5,6,7,8,9])

> splitAt 10 "Sapientia University"
("Sapientia ","University")
```

`notElem :: (Foldable t, Eq a) => a -> t a -> Bool`
Kimeneti értéke `True`, ha az első paraméterként megadott elem nincs benne a második paraméterként megadott listában, ellenkező esetben `False`.

```
> notElem 'O' "Maros-volgyi fatorzsbarlangok"
True
> notElem 'o' "Maros-volgyi fatorzsbarlangok"
False
```

Függvények listákon

```
concat :: Foldable t => t [a] -> [a]
```

A függvény sajátos esetben egy olyan listát vár, amelynek elemei szintén lista típusúak, eredményként pedig a bemeneti listákból egyetlenegy listát épít.

```
> concat ["torjai", " Budos", "-barlang"]
"torjai Budos-barlang"
> concat [[2, 4, 6], [1, 3, 5, 7, 9]]
[2,4,6,1,3,5,7,9]
```

```
repeat :: a -> [a]
```

A megadott elemmel egy végtelen listát határoz meg. Ajánlott a `take` vagy a `takeWhile` függvényekkel együtt használni.

```
> take 5 (repeat "hello")
["hello","hello","hello","hello","hello"]
```

Függvények listákon

```
replicate :: Int -> a -> [a]
```
A megadott elemmel egy `n` elemű listát határoz meg.

```
 > replicate 5 "hello"
 ["hello","hello","hello","hello","hello"]
```

```
cycle :: [a] -> [a]
```
A paraméterként megadott lista elemeit végtelenszer fűzi egymás után.

```
> take 10 $ cycle "barlang"
"barlangbar"
```

```
iterate :: (a -> a) -> a -> [a]
```
Alkalmazza a megadott függvényt, kiindulva a kezdeti értékként megadott paraméterből.

```
> take 10 (iterate (\x -> 2 * x ) 1)
[1,2,4,8,16,32,64,128,256,512]
```

Függvények listákon

```
any :: (a -> Bool) -> [a] -> Bool
```

Megvizsgálja, hogy a megadott feltételt teljesíti-e valamelyik listabeli elem.

```
> any isUpper "Sapientia University"
True
```

```
all :: (a -> Bool) -> [a] -> Bool
```

Megvizsgálja, hogy a megadott feltételt teljesíti-e minden listabeli elem.

```
> all isUpper "Sapientia University"
False
```

Feladatok

5. feladat

Írjunk két függvényt, amelyek rendre meghatározzák a magánhangzókat, illetve a mássalhangzókat egy karakterláncban.

```
import Data.List
abc = ['A'..'Z'] ++ ['a'..'z']
maganH = "aeiouAEIOU"
massalH = abc \\ maganH

maganHFilter :: [Char] -> [Char]
maganHFilter ls = filter (\c -> elem c maganH) ls

massalHFilter :: [Char] -> [Char]
massalHFilter ls = filter (flip elem massalH) ls

main = do
    let ls = "sapientia 2023, erdelyi magyar tud egyetem"
    let (mgH, msH) = (maganHFilter ls, massalHFilter ls)
    putStrLn $ "a maganhangzok: " ++ show mgH
    putStrLn $ "a massalhangzok: " ++ show msH
```