Funkcionális programozás
4. előadás

MÁRTON Gyöngyvér

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

2023, tavaszi félév

Miről volt szó?

- a `list`, `tuple` típusok, operátorok, függvények listákon
- kifejezések:
  - `if ... then ... else`
  - `case ... of`
  - `let ... in`
- függvénykompozíció
- függvénykiértékelés a $ szimbólummal
- a $ és a . szimbólumok
- feladatok:
  - két pont közötti távolság
  - tetszőleges számrendszerben a számjegyek száma
  - gyorshatványozás
  - hexa szimbólumok
  - másodfokú egyenlet gyökei
  - négyzetszámok, összetett számok, prímszámok tesztelése

Miről lesz szó?

- a Haskell kiértékelési stratégiája
- operátorok, függvények listákon: null, init, last, sum, map, filter, reverse, take, takeWhile, drop, dropWhile, elem, zip
- Haskell, a mellékhatások kezelése (side effects)
- kiíratási műveletek, a `mapM_` használata
- feladatok:
  - (mod p) szerinti hatványértékek
  - számok és a négyzetgyökök kiíratása

A Haskell kiértékelési stratégiája

- funkcionális programozási nyelvek esetén kétféle kiértékelési stratégiát ismerünk: lusta (lazy), mohó (eager)
- a Haskell kiértékelési stratégiája lusta,
- Lusta kiértékelési stratégia:
  - a legbaloldalibb, legkülső redex (redukálható kifejezés) helyettesítése történik először,
  - egy alkifejezés csak akkor értékelődik ki, ha szükség van az értékére (ha a kifejezés függvénymegadással kezdődik előbb a függvénydefiníció lesz alkalmazva)
  - ez a stratégia mindig megtalálja a normál formát, ha az létezik Pl. Clean, Haskell, Miranda
  - lehetségessé válik a függvények kötetlen definiálása: egy függvény akkor is képes értéket visszaadni, ha egyik argumentuma nem definiált,
  - lehetségessé válik a végtelen adatszerkezetek létrehozása,
  - a mohó kiértékelési stratégiához képest kevésbé hatékony.

A mohó (eager) kiértékelési stratégia

- a legbaloldalibb, legbelső redex, az argumentumok helyettesítése történik meg először,
- nem mindig ér véget a kiértékelési folyamat,
- hatékonyabb mint a lusta rendszer,
- Pl. Lisp, SML, Hope,
- a lusta kiértékelési stratégia hatékonyságát oly módon lehet javítani, hogy az azonos részkifejezéseket megjelöljük, az eredményt megjegyezzük és ahányszor szükség van rá mindig a megjegyzett eredményt használjuk.

Példa, kiértékelési stratégiákra

```
myInc :: Num a => a -> a
myInc x = x + 1

negyzet :: Num a => a -> a
negyzet x = x * x

negyzet_inc :: Num a => a -> a
negyzet_inc x = negyzet (myInc x)

> negyzet_inc 6
```

A lusta kiértékelési stratégia:

```
negyzet_inc 6
-> negyzet (myInc 6)
-> (myInc 6) * (myInc 6)
-> (6 + 1) * (6 + 1)
-> 7 * 7 -> 49
```

A mohó kiértékelési stratégia:

```
negyzet_inc 6
-> negyzet (myInc 6)
-> negyzet (6 + 1)
-> negyzet 7
-> 7 * 7 -> 49
```

Függvények listákon

```
null :: [a] -> Bool
```

Megvizsgálja hogy egy lista üres lista-e vagy tartalmaz elemeket.

```
myNull :: [a] -> Bool

myNull [] = True
myNull (k : ve) = False
--myNull (_ : _) = False
```

```
init :: [a] -> [a]
```

Egy listát térít vissza, amelyben nem szerepel az eredeti lista utolsó eleme, nem alkalmazható üres listákra.

```
myInit :: [a] -> [a]

myInit [] = error "ures lista"
myInit [k] = []
myInit (k : ve) = k : myInit ve

> myInit [54, 67, 23, 89, 102, 110, 225]
[54, 67, 23, 89, 102, 110]
```

Függvények listákon

```
last :: [a] -> a
```
Meghatározza egy lista utolsó elemét.

```
myLast :: [a] -> a

myLast [] = error "ures lista"
myLast [k] = k
myLast (_ : ve) = myLast ve

> myLast "hello"
'o'
```

1. feladat

Határozzuk meg egy lista második elemét

```
masodikE :: [a] -> a
masodikE [] = error "ures lista"
masodikE [k1] = error "egy elemu"
masodikE (k1 : k2 : ve) = k2
--masodikE (_ : k2 : _) = k2

> masodikE [[3, 3, 3], [2, 2], [4, 4, 4, 4], [5]]
[2, 2]
```

Függvények listákon

```
sum :: Num a => [a] -> a
```

Összeadja a lista elemeit.

```
mySum1 :: Num a => [a] -> a
mySum1 [] = 0
mySum1 (k : ve) = k + mySum1 ve

mySum2 :: Num a => [a] -> a
mySum2 [] = 0
mySum2 ls = head ls + mySum2 (tail ls)

mySum3 :: Num a => [a] -> a -- a hatékony megoldas
mySum3 ls = auxSum ls 0
    where
    auxSum [] res = res
    auxSum (k : ve) res = auxSum ve (k + res)
```

Komplex számokat is össze lehet adni:

```
> import Data.Complex
> mySum3 [3 :+ (-2.3), 3 :+ 2.1, 8.54 :+ 1.3]
14.54 :+ 1.1000000000000003
```

Függvények listákon

`map :: (a -> b) -> [a] -> [b]`
A paraméterként megadott függvényt alkalmazza a második paraméterként megadott lista minden elemére.

```
myMap1 :: (a -> b) -> [a] -> [b]
myMap1 fg [] = []
myMap1 fg (k : ve) = fg k : myMap1 fg ve
```

A null függvény segítségével a map függvény 2. verziója:

```
myMap2 :: (a -> b) -> [a] -> [b]
myMap2 fg ls =
    if null ls then []
    else fg (head ls) : myMap2 fg (tail ls)
```

```
> import Data.Char
> myMap1 toUpper "abcdefghijkLMNop"
"ABCDEFGHIJKLMNOP"
```

Függvények listákon

2. 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 = myMap2 (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
        print (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

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

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

-- > fugvKiir3 [12, 45, 67, 89]
fugvKiir3 :: Show a => [a] -> IO ()
fugvKiir3 ls = mapM_ print $ sort ls
```

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

Komplex számok rendezése:

```
import Data.Complex
import Data.List

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

Kiíratási műveletek

4. 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
    where
    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

5. feladat

Modósítsuk az előző Haskell-függvényt, hogy az eredményeket a négyzetgyökértékek alapján rendezve írja ki a képernyőre.

```
import Data.List
import Data.Ord

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