Funkcionális programozás
4. előadás

MÁRTON Gyöngyvér

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

2024, tavaszi félév

Miről volt szó?

- lambda kifejezések
- magasabb rendű függvények részleges paraméterezése
- a `list`, `tuple` típusok, operátorok, függvények listákon
- kifejezések:
  - `if ... then ... else`
  - `let ... in`
- feladatok:
  - gyorshatványozás
  - két pont közötti távolság
  - tetszőleges számrendszerben a számjegyek száma
  - másodfokú egyenlet gyökei

Miről lesz 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

A `case ... of` kifejezés

1. feladat

Határozzuk meg egy 16-os számrendszerben megadott számsorozatnak azon alakját, amelyben a számértékeket a 16-os számrendszerben használt szimbólumokkal helyettesítjük.

A case ... of kifejezés

1. feladat

Határozzuk meg egy 16-os számrendszerben megadott számsorozatnak azon alakját, amelyben a számértékeket a 16-os számrendszerben használt szimbólumokkal helyettesítjük.

```
import Data.Char
hexaSz :: Int -> Char
hexaSz c
    | c >= 0 && c < 16 =
```

A case ... of kifejezés

1. feladat

Határozzuk meg egy 16-os számrendszerben megadott számsorozatnak azon alakját, amelyben a számértékeket a 16-os számrendszerben használt szimbólumokkal helyettesítjük.

```
import Data.Char
hexaSz :: Int -> Char
hexaSz c
    | c >= 0 && c < 16 =
        case c of
            10 -> 'A'
            11 -> 'B'
            12 -> 'C'
            13 -> 'D'
            14 -> 'E'
            15 -> 'F'
```

A case ... of kifejezés

1. feladat

Határozzuk meg egy 16-os számrendszerben megadott számsorozatnak azon alakját, amelyben a számértékeket a 16-os számrendszerben használt szimbólumokkal helyettesítjük.

```
import Data.Char
hexaSz :: Int -> Char
hexaSz c
    | c >= 0 && c < 16 =
        case c of
            10 -> 'A'
            11 -> 'B'
            12 -> 'C'
            13 -> 'D'
            14 -> 'E'
            15 -> 'F'
            _ -> chr (c + 48)
    | otherwise = error "rossz bemenet"
```

Feladat

2. feladat

Írjunk egy Haskell-függvényt, amely meghatározza egy 16-os számrendszerben megadott számsorozatnak azon alakját, amelyben a számértékeket a 16-os számrendszerben használt szimbólumokkal helyettesítjük. Használjuk a korábban megírt hexaSz függvényt.

```
hexaLs1 :: [Int] -> [Char]
hexaLs1 [] = ""
hexaLs1 (k : ve) = hexaSz k : hexaLs1 ve

hexaLs2 :: [Int] -> [Char]
hexaLs2 ls = map hexaSz ls

hexaLs3 :: [Int] -> [Char]
hexaLs3 ls = [hexaSz k | k <- ls]

> hexaLs2 [12, 4, 5, 15, 7, 0, 11, 4]
"C45F70B4"
```

Feladat

```
> hexaLs3 [10, 16, 6]
"A*** Exception: rossz bemenet...
```

Feladat

```
> hexaLs3 [10, 16, 6]
"A*** Exception: rossz bemenet...

> import Numeric
> showHex 89348 ""
```

Feladat

```
> hexaLs3 [10, 16, 6]
"A*** Exception: rossz bemenet...

> import Numeric
> showHex 89348 ""

> map (flip showHex "") [10, 16, 6]
> unwords $ map (flip showHex "") [10, 16, 6]
```

Függvénykompozíció

A matematikából ismert művelet megfelelője.

3. feladat

Határozzuk meg a paraméterként megadott számok közül a páratlan számokat.

Függvénykompozíció

A matematikából ismert művelet megfelelője.

3. feladat

Határozzuk meg a paraméterként megadott számok közül a páratlan számokat.

```
paratlanLista :: (Integral a) => [a] -> [a]
paratlanLista ls = filter (not . even) ls
```

Függvénykompozíció

A matematikából ismert művelet megfelelője.

3. feladat

Határozzuk meg a paraméterként megadott számok közül a páratlan számokat.

```
paratlanLista :: (Integral a) => [a] -> [a]
paratlanLista ls = filter (not . even) ls

> paratlanLista [1..20]
[1,3,5,7,9,11,13,15,17,19]
```

Függvénykompozíció

A matematikából ismert művelet megfelelője.

3. feladat

Határozzuk meg a paraméterként megadott számok közül a páratlan számokat.

```
paratlanLista :: (Integral a) => [a] -> [a]
paratlanLista ls = filter (not . even) ls

> paratlanLista [1..20]
[1,3,5,7,9,11,13,15,17,19]
```

Nem kell explicit módon megadni a paramétert:

```
paratlanLista1 :: (Integral a) => [a] -> [a]
paratlanLista1 = filter (not . even)

> paratlanLista1 [1..20]
[1,3,5,7,9,11,13,15,17,19]
```

Függvénykompozíció

A korábbi előadásban megadott `duplaz` függvény is megadható függvénykompozíciót alkalmazva:

```
duplaz :: (a -> a) -> a -> a
duplaz fg = fg . fg

> duplaz (+1) 10
```

Függvénykompozíció

A korábbi előadásban megadott `duplaz` függvény is megadható függvénykompozíciót alkalmazva:

```
duplaz :: (a -> a) -> a -> a
duplaz fg = fg . fg

> duplaz (+1) 10
```

4. feladat

Határozzuk meg a paraméterként megadott természetes számokat tartalmazó listából azokat a számokat, amelyek nem négyzetszámok.

Függvénykompozíció

A korábbi előadásban megadott `duplaz` függvény is megadható függvénykompozíciót alkalmazva:

```
duplaz :: (a -> a) -> a -> a
duplaz fg = fg . fg

> duplaz (+1) 10
```

4. feladat

Határozzuk meg a paraméterként megadott természetes számokat tartalmazó listából azokat a számokat, amelyek nem négyzetszámok.

```
nemNegyzet :: (Integral a) => [a] -> [a]
nemNegyzet = filter (not . negyzetV)

negyzetV :: Integral a => a -> Bool
negyzetV x = temp * temp == x
    where
    temp = truncate (sqrt (fromIntegral x))

> nemNegyzet [12, 121, 34, 49, 625, 133]
[12,34,133]
```

Függvénykompozíció

5. feladat

Vágjuk le a paraméterként megadott lista, karakterlánc első és utolsó karakterét.

Alkalmazzuk az `init, tail` könyvtárfüggvényeket.

```
> init [1..10]
[1,2,3,4,5,6,7,8,9]
> tail [1..10]
[2,3,4,5,6,7,8,9,10]
```

Függvénykompozíció

5. feladat

Vágjuk le a paraméterként megadott lista, karakterlánc első és utolsó karakterét.

Alkalmazzuk az `init, tail` könyvtárfüggvényeket.

```
> init [1..10]
[1,2,3,4,5,6,7,8,9]
> tail [1..10]
[2,3,4,5,6,7,8,9,10]

levag :: [a] -> [a]
levag = init . tail

> levag "gezakekazeg"
"ezakekaze"
```

Függvénykompozíció

6. feladat

Írjunk egy Hakell függvényt, amely a paraméterként megadott 1-nél nagyobb, természetes számokat tartalmazó listából, kiválogatja az összetett számokat.

```
osszetett :: (Integral a) => [a] -> [a]
osszetett ls = filter (not . primT 3) ls

osszetett1 :: (Integral a) => [a] -> [a]
osszetett1 ls = [i | i <- ls, (not . primT 3) i]
```

Függvénykompozíció

6. feladat

Írjunk egy Hakell függvényt, amely a paraméterként megadott 1-nél nagyobb, természetes számokat tartalmazó listából, kiválogatja az összetett számokat.

```
osszetett :: (Integral a) => [a] -> [a]
osszetett ls = filter (not . primT 3) ls

osszetett1 :: (Integral a) => [a] -> [a]
osszetett1 ls = [i | i <- ls, (not . primT 3) i]

import Data.List
osszetett2 :: (Integral a) => [a] -> [a]
osszetett2 ls = ls \\ [i | i <- ls, primT 3 i]

> osszetett [24, 97, 5, 1, 74, 41, 61, 19, 100]
[24,1,74,100]
```

Függvénykompozíció

A primtesztelést végző függvény:

```
primT :: (Integral a) => a -> a -> Bool
primT k nr
    | nr < 0 = error "hibas bemenet"
    | nr == 1 = False
    | nr == 2 = True
    | mod nr 2 == 0 = False
    | nr < k * k = True
    | mod nr k == 0 = False
    | otherwise = primT (k + 2) nr

> primT 3 101
True
```

A $ operátor

- a kifejezések kiértékelési sorrendjét változtatja meg,
- fölöslegessé válik a zárójelezés,
- jobbról asszociatív: előbb a jobb oldalon levő kifejezés értékelődik ki,
- az operátorok között legkisebb a prioritása.

Meghatározza $\sqrt{2}$, majd hozzáadja a 3-t, majd az 5-t:

```
> sqrt 2 + 3 + 5
9.414213562373096
```

A $ operátor

- a kifejezések kiértékelési sorrendjét változtatja meg,
- fölöslegessé válik a zárójelezés,
- jobbról asszociatív: előbb a jobb oldalon levő kifejezés értékelődik ki,
- az operátorok között legkisebb a prioritása.

Meghatározza $\sqrt{2}$, majd hozzáadja a 3-t, majd az 5-t:

```
> sqrt 2 + 3 + 5
9.414213562373096
```

Összeadja a számokat és azután határozza meg $\sqrt{10}$ értékét:

```
> sqrt (2 + 3 + 5)
3.1622776601683795
```

A $ operátor

- a kifejezések kiértékelési sorrendjét változtatja meg,
- fölöslegessé válik a zárójelezés,
- jobbról asszociatív: előbb a jobb oldalon levő kifejezés értékelődik ki,
- az operátorok között legkisebb a prioritása.

Meghatározza $\sqrt{2}$, majd hozzáadja a 3-t, majd az 5-t:

```
> sqrt 2 + 3 + 5
9.414213562373096
```

Összeadja a számokat és azután határozza meg $\sqrt{10}$ értékét:

```
> sqrt (2 + 3 + 5)
3.1622776601683795
```

Összeadja a számokat és azután határozza meg $\sqrt{10}$ értékét:

```
> sqrt $ 2 + 3 + 5
3.1622776601683795
```

Függvénykiértékelés a $ szimbólummal

- Előbb alkalmazza az `abs` függvényt:

```
> sqrt $ abs (-16)
4.0
```

Függvénykiértékelés a $ szimbólummal

- Előbb alkalmazza az `abs` függvényt:

```
> sqrt $ abs (-16)
4.0
```

- Összeadja a számokat, alkalmazza az `abs` függvényt, majd meghatározza a négyzetgyököt:

```
> sqrt $ abs $ (-16) + 9
2.6457513110645907
```

Függvénykiértékelés a $ szimbólummal

- Előbb alkalmazza az `abs` függvényt:

```
> sqrt $ abs (-16)
4.0
```

- Összeadja a számokat, alkalmazza az `abs` függvényt, majd meghatározza a négyzetgyököt:

```
> sqrt $ abs $ (-16) + 9
2.6457513110645907
```

- Alkalmazza az `abs` függvényt, hozzáadja a 9-et, majd meghatározza a négyzetgyököt:

```
> sqrt $ abs (-16) + 9
5.0
```

A $ és a . szimbólumok

A . szimbólummal elsősorban az függvényhívások összeláncolását (kompozicióját) lehet megvalósítani, a lényeg nem a kevesebb zárójelhasználat, bár az eredmény az, hogy kevesebb zárójelt kell használni.

- Előbb alkalmazza az `abs` függvényt:

```
> (sqrt . abs) (-16)
4.0
```

A $ és a . szimbólumok

A . szimbólummal elsősorban az függvényhívások összeláncolását (kompozicióját) lehet megvalósítani, a lényeg nem a kevesebb zárójelhasználat, bár az eredmény az, hogy kevesebb zárójelt kell használni.

- Előbb alkalmazza az `abs` függvényt:

```
> (sqrt . abs) (-16)
4.0
```

- Ugyanazt érem el, mint az előbbi, de kevesebb zárójellel:

```
> sqrt . abs $ -16
4.0
```

A $ és a . szimbólumok

- előbb alkalmazza a `head` függvényt, majd a `toLower` függvényt és utána a `:` konstruktort, azaz épít egy új listát aminek az első elemét kisbetűre cseréli, a többi marad változatlan

```
import Data.Char
fugvKB :: [Char] -> [Char]
fugvKB ls = (toLower . head $ ls) : tail ls
```

A $ és a . szimbólumok

- előbb alkalmazza a `head` függvényt, majd a `toLower` függvényt és utána a `:` konstruktort, azaz épít egy új listát aminek az első elemét kisbetűre cseréli, a többi marad változatlan

```
import Data.Char
fugvKB :: [Char] -> [Char]
fugvKB ls = (toLower . head $ ls) : tail ls

> fugvKB "SAPIENTIA"
"sAPIENTIA"
```

- a karakterlánc első betűjét nagybetűre cseréli, a többi marad változatlan

```
fugvNB :: [Char] -> [Char]
fugvNB ls = (toUpper . head) ls : tail ls
```

A $ és a . szimbólumok

- előbb alkalmazza a `head` függvényt, majd a `toLower` függvényt és utána a `:` konstruktort, azaz épít egy új listát aminek az első elemét kisbetűre cseréli, a többi marad változatlan

```
import Data.Char
fugvKB :: [Char] -> [Char]
fugvKB ls = (toLower . head $ ls) : tail ls

> fugvKB "SAPIENTIA"
"sAPIENTIA"
```

- a karakterlánc első betűjét nagybetűre cseréli, a többi marad változatlan

```
fugvNB :: [Char] -> [Char]
fugvNB ls = (toUpper . head) ls : tail ls

> map fugvNB ["sapientia", "emte", "mvh"]
["Sapientia","Emte","Mvh"]
```

A $ és a . szimbólumok

A `words` a paraméterként megadott stringet szavakra bontja:

```
> words "sapientia hungarian university marosvasarhely"
["sapientia","hungarian","university","marosvasarhely"]
```

A $ és a . szimbólumok

A `words` a paraméterként megadott stringet szavakra bontja:

```
> words "sapientia hungarian university marosvasarhely"
["sapientia","hungarian","university","marosvasarhely"]

> ls = "sapientia hungarian university marosvasarhely"
> map fugvNB $ words ls
["Sapientia","Hungarian","University","Marosvasarhely"]
```

A $ és a . szimbólumok

A `words` a paraméterként megadott stringet szavakra bontja:

```
> words "sapientia hungarian university marosvasarhely"
["sapientia","hungarian","university","marosvasarhely"]

> ls = "sapientia hungarian university marosvasarhely"
> map fugvNB $ words ls
["Sapientia","Hungarian","University","Marosvasarhely"]
```

Másképp:

```
> map (\x -> (toUpper . head) x : tail x) (words ls)
```

A $ és a . szimbólumok

A `words` a paraméterként megadott stringet szavakra bontja:

```
> words "sapientia hungarian university marosvasarhely"
["sapientia","hungarian","university","marosvasarhely"]

> ls = "sapientia hungarian university marosvasarhely"
> map fugvNB $ words ls
["Sapientia","Hungarian","University","Marosvasarhely"]
```

Másképp:

```
> map (\x -> (toUpper . head) x : tail x) (words ls)

> (map (\x -> (toUpper . head) x : tail x) . words) ls
```

A $ és a . szimbólumok

A `words` a paraméterként megadott stringet szavakra bontja:

```
> words "sapientia hungarian university marosvasarhely"
["sapientia","hungarian","university","marosvasarhely"]

> ls = "sapientia hungarian university marosvasarhely"
> map fugvNB $ words ls
["Sapientia","Hungarian","University","Marosvasarhely"]
```

Másképp:

```
> map (\x -> (toUpper . head) x : tail x) (words ls)

> (map (\x -> (toUpper . head) x : tail x) . words) ls

> map (\x -> (toUpper . head) x : tail x) $ words ls
```

A $ és a . szimbólumok

Az `unwords` a fordított műveletet végzi úgy, hogy az összefűzött szavak közé szóközöket tesz:

```
> unwords ["paring","fogaras","kiralyko"]
"paring fogaras kiralyko"
```

A $ és a . szimbólumok

Az `unwords` a fordított műveletet végzi úgy, hogy az összefűzött szavak közé szóközöket tesz:

```
> unwords ["paring","fogaras","kiralyko"]
"paring fogaras kiralyko"
```

A szavak kezdőbetűit átalakítja nagybetűkké, 2 módszerrel:

```
fugvM1 :: String -> String
fugvM1 ls =  unwords $ map aux $ words ls
    where
    aux = \ x -> (toUpper . head) x : tail x
```

A $ és a . szimbólumok

Az `unwords` a fordított műveletet végzi úgy, hogy az összefűzött szavak közé szóközöket tesz:

```
> unwords ["paring","fogaras","kiralyko"]
"paring fogaras kiralyko"
```

A szavak kezdőbetűit átalakítja nagybetűkké, 2 módszerrel:

```
fugvM1 :: String -> String
fugvM1 ls =  unwords $ map aux $ words ls
    where
    aux = \ x -> (toUpper . head) x : tail x

fugvM2 :: String -> String
fugvM2 = unwords . map aux . words
    where
    aux x = (toUpper . head) x : tail x
```

A $ és a . szimbólumok

Az `unwords` a fordított műveletet végzi úgy, hogy az összefűzött szavak közé szóközöket tesz:

```
> unwords ["paring","fogaras","kiralyko"]
"paring fogaras kiralyko"
```

A szavak kezdőbetűit átalakítja nagybetűkké, 2 módszerrel:

```
fugvM1 :: String -> String
fugvM1 ls =  unwords $ map aux $ words ls
    where
    aux = \ x -> (toUpper . head) x : tail x

fugvM2 :: String -> String
fugvM2 = unwords . map aux . words
    where
    aux x = (toUpper . head) x : tail x

> fugvM1 "retyezat kudzsiri bucsecs jezer"
"Retyezat Kudzsiri Bucsecs Jezer"

> fugvM2 "szebeni csernai vulkan"
"Szebeni Csernai Vulkan"
```

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
```

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
```

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
```

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]
```

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
```

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'
```

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'
```

7. feladat

Határozzuk meg egy lista második elemét.

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'
```

7. 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.

Függvények listákon

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

Összeadja a lista elemeit.

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

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)
```

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)
```

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]
```

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 []
```

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"
```