Funkcionális programozás
2. előadás

MÁRTON Gyöngyvér

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

2023, tavaszi félév

Miről volt szó?

- követelmények
- könyvészet
- programozási módszerek összehasonlítása: imperatív, funkcionális, logikai
- Haskell, történelmi háttér, telepítés, használat, bevezető fogalmak, az első program
- Haskell alaptípusok
- feladatok: faktoriális számítás

Miről lesz szó?

- típusosztályok
- típusdefiníciók
- megjegyzések használata
- könyvtármodul importálása
- feltételek megadása
- rekurzió, margószabály, mintaillesztés
- halmazkifejezések, lambda kifejezések
- magasabb rendű függvények, részleges paraméterezés
- a `list` típus, operátorok, függvények listákon
- feladatok:
  - területszámítás
  - abszolút érték
  - aritmetikai műveletek
  - tuple elemek megegyeznek-e?
  - másodfokú egyenlet gyökei
  - legnagyobb közös osztó
  - számjegyek összege, szorzata
  - szám osztóinak listája
  - gyorshatványozás

Típusosztályok

Az `Eq` azokat a típusokat tartalmazza, amelyek esetében az értékek *egyenlőség* és *nem egyenlőség* operátorokkal összehasonlíthatóak, definíciója a következő:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

```
> "hello" /= "Hello"
True

> 13.5 == 3.4
False
```

Annak ellenére, hogy egy típusosztály a `class` kulcsszóval van definiálva, nem ugyanazt jelenti, mint az OOP-ben használt `class`. A Haskellben a tulajdonképpeni szerepe, hogy összekapcsol több típust.

Típusosztályok

Az `Ord` az `Eq` típusosztályból van származtatva, és olyan típusváltozók esetében használjuk, amikor az értékek között rendezettségi kapcsolat áll fenn. Definíciója a következő:

```
class Ord a where
    (<) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    min :: a -> a -> Bool
    max :: a -> a -> Bool
    compare :: Ord a => a -> a -> Ordering
```

A `compare` a paraméterként megkapott két értéket hasonlítja össze, kimenete az `LT`, `GT` vagy `EQ` konstans lesz, aszerint, hogy az első érték a kisebb, az első érték a nagyobb, vagy a két érték megegyezik.

```
> compare 4 5
LT

> compare "hello" "hello"
EQ

> compare 'a' 'Z'
GT
```

Típusosztályok

A `Num` típusosztályt akkor használjuk, amikor numerikus értékekkel dolgozunk:

```
class Num a where
     (+), (-), (*) :: a -> a -> a
     negate :: a -> a
     abs, signum :: a -> a
     fromInteger :: Integer -> a
```

A `negate` megváltoztatja a bemenet előjelét, az `abs` meghatározza a bemenet abszolút értékét, míg a `signum` kimenete `(-1)`, ha a bemenet negatív, `1`, ha a bemenet pozitív szám, és `0`-t határoz meg, ha a bemenet `0`.

```
> negate (-7.8)
7.8
> negate 4
-4
```

```
> signum (-5.4)
-1.0
> signum 5
1
```

A `fromInteger` explicit típuskonverziót tesz lehetővé, `Integer` típusról alakít,

Típuskonverziók

- Ha két `Integer` típusú értéken akarjuk az osztás `(/)` műveletét alkalmazni, akkor típuskonverziót kell alkalmazni,
- az `osztas_` esetében fordítási hiba lép fel, míg az `osztas` függvény fordításakor nem lesz hiba

```
osztas_ :: Integer -> Integer -> Double
osztas_ x y = x / y

... error:
Couldn't match expected type 'Double' with...
```

```
osztas :: Integer -> Integer -> Double
osztas x y = fromInteger x / fromInteger y

> osztas 758375832 21171189
35.82112615403887
```

További típuskonverziót végezhetünk a következő függvényekkel:

`toInteger, fromRational, toRational`

Típusosztályok

Az `Integral` típusosztályt akkor használjuk, amikor egész számokkal szeretnénk műveleteket végezni.

- a `Real`, illetve `Enum` típusosztályokból van származtatva,
- az `Enum` típusosztályba azok a típusok tartoznak, amelyeknek az értékei felsorolhatók,
- a `Real` típuskonverziót biztosít a `Rational` típusról az `Integer` típusra,
- az `Integral` magába foglalja az `Int` és az `Integer` típust, ugyanakkor benne van a `Num` típusosztályba is.

Definíciója a következő:

```
class (Real a, Enum a) => Integral a where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod :: a -> a -> (a,a)
    toInteger :: a -> Integer
```

```
> mod (-3) 4
1
> rem (-3) 4
-3
```

Típusosztályok

A Fractional a `Num` alosztálya:

- tulajdonképpen a racionális számokat kezeli, azaz azokat a típusokat amelyeken osztás, és reciprok érték határozható meg,
- a `fromRational`-al `Rational` típusú értékről lehet alakítani

```
class Num a => Fractional a where
    (/) :: a -> a -> a
    recip :: a -> a
    fromRational :: Rational -> a
```

```
> recip 2.0
0.5
```

```
> import Data.Ratio
> fromRational (3 % 10)
0.3
```

Típusosztályok

A RealFrac a `Fractional` és `Real` alosztálya, a következő függvények tartoznak hozzá:

```
properFraction  :: (Fractional a, Integral b) => a -> (b,a)
truncate, round,
floor, ceiling :: (Fractional a, Integral b) => a -> b
```

```
> properFraction(3.4)
(3,0.3999999999999999)

> round(3.4)
3
> round(3.9)
4

> truncate(3.9)
3
> truncate(3.2)
3
```

Típusosztályok

A Floating elsősorban a valós értékek kezeléséért felelős, hozzá ide tartoznak a trigonometrikus függvények:

```
class (Fractional a) => Floating a where
    pi :: Floating a => a
    (**) :: Floating a => a -> a -> a
    sqrt, exp, log :: Floating a => a -> a
    sin, cos, tan :: Floating a => a -> a
    ⋮
```

Típusosztályok

A Show azokat a típusokat tartalmazza, amelyek értékei átalakíthatóak karakterlánccá (`String`-é).

- magába foglalja az alaptípusokat (Bool, Char, String, Int, Integer, Float, Double, Lista, Tuple):

```
show :: a -> String
```

```
> show 579
"579"

> show ('b', True)
"('b', True)"
```

Típusosztályok

A Read azokat a típusokat tartalmazza, amelyek értékét meg lehet határozni karakterláncból,

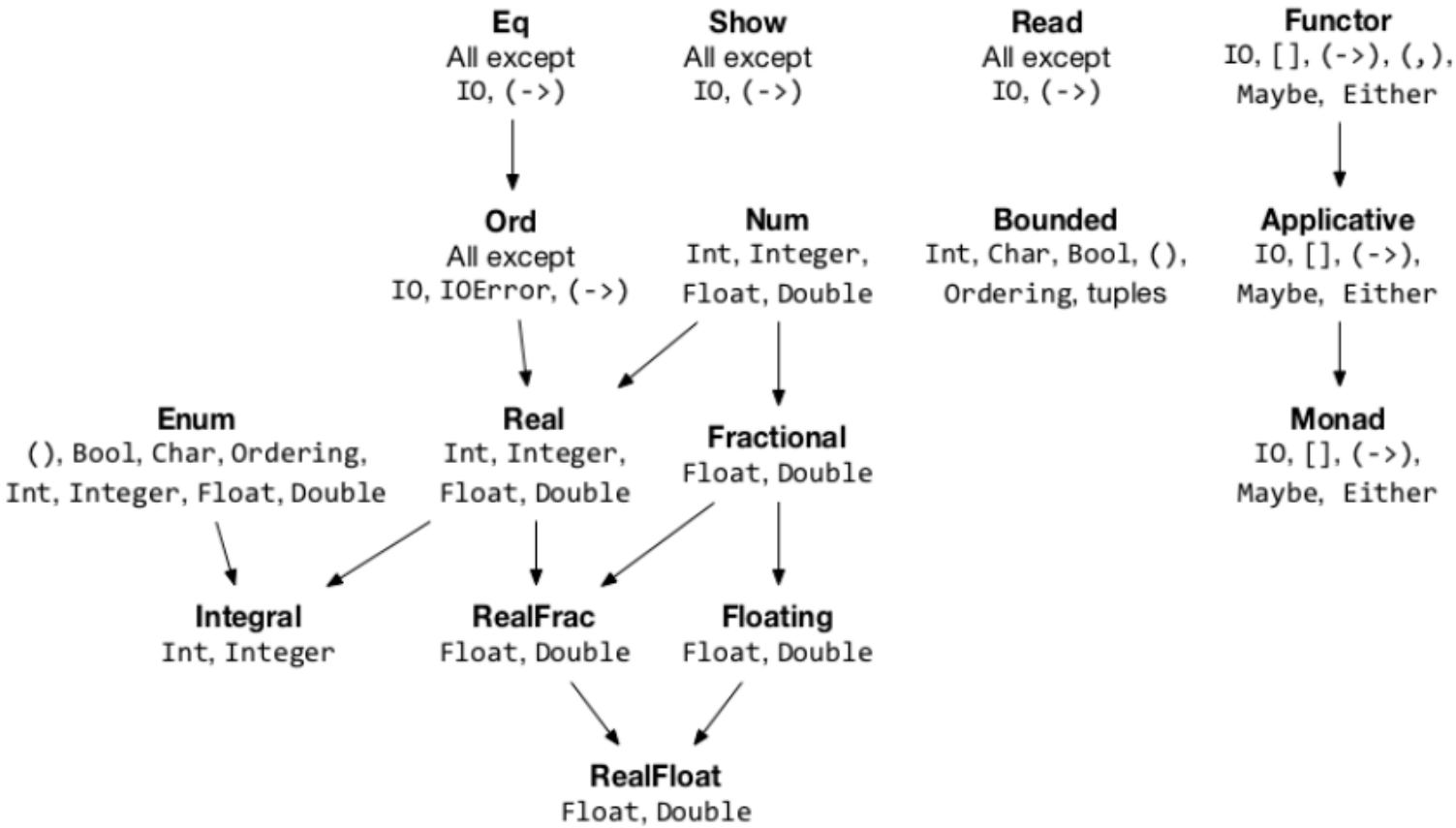
- használatakor meg kell adni, hogy milyen típusra alakítunk, ehhez a :: jelet kell használni,
- magába foglalja az alaptípusokat (Bool, Char, String, Int, Integer, Float, Double, Lista, Tuple):

```
read :: String -> a
```

```
> read "False" :: Bool
False

> read "[10, 11, 12, 13]" :: [Int]
[10, 11, 12, 13]
```

A standard Haskell osztálydiagramm



Típusdefiníciók

A kifejezések típusát a fordítóprogram meghatározza úgy is, ha azok nincsenek feltüntetve.

1. feladat

Határozzuk meg egy szám abszolút értékét, 1. változat.

```
abszolut1 x
    | x < 0 = -x
    |otherwise = x
```

a függvény kiértékelhető, egész és valós típusú bemenetre is:

```
> abszolut1 (-10.0)
10.0
```

leszűkítjük a paraméterek típusát, csak Int típusúak lehetnek:

```
abszolut2 :: Int -> Int
abszolut2 x
    | x < 0 = -x
    |otherwise = x

> abszolut2 (-10.0)
No instance for (Fractional Int) arising from the literal '10' ...
```

Típusdefiníciók

Az általános típusdefiníció:

```
abszolut :: (Num a, Ord a) => a -> a
abszolut x
    | x < 0 = -x
    |otherwise = x
```

a függvény kiértékelhető, egész és valós típusú bemenetre is:

```
> abszolut (-10.0)
10.0
> abszolut (-10)
10
```

Típusdefiníciók

2. feladat

Definiáljunk egy függvényt, amely meghatározza két szám összegét.

```
--osszeg :: Double -> Double -> Double
osszeg :: (Num a) => a -> a -> a
osszeg x y = x + y
```

3. feladat

Definiáljunk egy függvényt, amely egy 4 elemű tuple típusba meghatározza két szám összegét, különbségét, szorzatát, hányadosát.

```
--aritM :: Float -> Float -> (Float, Float, Float, Float)
--aritM :: Double -> Double -> (Double, Double, Double, Double)
aritM :: (Fractional a) => a -> a -> (a, a, a, a)
aritM x y = (r1, r2, r3, r4)
    where
    r1 = x + y
    r2 = x * y
    r3 = x - y
    r4 = x / y
```

Típusdefiníciók

4. feladat

Definiáljunk egy függvényt, amely egy 6 elemű tuple típusba meghatározza két szám összegét, különbségét, szorzatát, hányadosát, osztási egészrészét, osztási maradékát.

```
aritM_ ::
    (Integral a, Fractional a1) => a -> a -> (a, a, a, a1, a, a)
aritM_ x y = (r1, r2, r3, r4, r5, r6)
    where
    r1 = x + y
    r2 = x * y
    r3 = x - y
    r4 = fromIntegral x / fromIntegral y
    r5 = div x y
    r6 = mod x y
```

A fenti függvény nem hívható meg valós bemenetre:

```
> aritM_ 4.3 5
<interactive>:5:1: error: ...
```

A `fromIntegral` explicit típuskonverziót tesz lehetővé, `Integer`, `Int` típusról alakít:

```
fromIntegral :: (Integral a, Num b) => a -> b
```

Típusdefiníciók

A következő változat meghívható egész és valós bemenetre egyaránt:

```
aritM__ ::
    (RealFrac a, Integral a1) =>
    a -> a -> (a, a, a, a, a1, a1)
aritM__ x y = (r1, r2, r3, r4, r5, r6)
    where
    r1 = x + y
    r2 = x * y
    r3 = x - y
    r4 = x / y
    r5 = div (truncate x) (truncate y)
    r6 = mod (truncate x) (truncate y)
```

Az alábbi lekérdezése explicit típuskonverzió hiányában futási hibát ad:

```
> ls = [8.50, 9.75, 8.75, 7.50, 10, 8.25]
> sum ls / length ls
```

Helyesen:

```
> sum ls / fromIntegral (length ls)
```

Típusdefiníciók

Feltételezve, hogy az `aritM_` függvény, az `eload2.hs` állományban van, futtatható állomány létrehozása érdekében, tegyük még hozzá a következőket:

```
main = do
    print (aritM_ 13 14)
    print "Press any key to continue ..."
    return getLine()
```

majd adjuk ki a következő Haskell parancsot:

```
> :! ghc --make "eload2.hs"
```

vagy

```
> :! ghc "eload2.hs"
```

Típusdefiníciók

5. feladat

Definiáljunk egy olyan függvényt, amely megvizsgálja, hogy két elempár értékei "majdnem" megegyeznek-e: akkor térít vissza True értéket a függvény, ha a két pár ugyanazokat az értékeket tartalmazza függetlenül az elemek sorrendjétől.

```
--elemparF :: (Int, Int) -> (Int, Int) -> Bool
--elemparF :: (String, String) -> (String, String) -> Bool
elemparF :: (Eq a) => (a, a) -> (a, a) -> Bool
elemparF t1 t2
    | (a == c && b == d ) || (a == d && b == c) = True
    | otherwise = False
    where
    (a, b) = t1
    (c, d) = t2

> elemparF ("aaa", "ddd") ("ddd", "aaa")
True

> elemparF (3.5, 4) (4, 3.5)
True
```

Megjegyzések használata, fenntartott szavak

- egysoros megjegyzés:

```
--ez egy egysoros megjegyzés
```

- több soros megjegyzés:

```
{-
ez egy több soros
megjegyzés
-}
```

- fenntartott szavak

```
case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where
```

Könyvtármodul importálása

```
import Data.Char
> isDigit '3'
True

> isAlpha 'a'
True
```

```
> isDigit 'w'
False

> isAlpha '?'
False
```

```
my_isDigit :: Char -> Bool
my_isDigit x = x >= '0' && x <= '9'
```

```
> my_isDigit '3'
True
```

```
> import Data.List
> tails "hello"
["hello","ello","llo","lo","o",""]

> nub "sapientia egyetem"
> "sapient gym"

> import Data.Complex
> sum[3 :+ (-2.6), 11 :+ 3.4, 1 :+ (-2.41), (-8) :+ 4.11]
7.0 :+ 2.5
```

Feltételek megadása (definition by cases)

Feltételek megadása: a függvény értékét az első, igaz feltételhez tartozó kifejezés adja, amely kifejezést az aktuális paraméter értéke alapján értékeljük ki.

6. feladat

Határozzuk meg egy szám előjelét.

```
elojel :: Int -> Int
elojel x
   | x < 0 = -1
   | x > 0 = 1
   | x == 0 = 0
```

```
elojel1 :: Int -> String
elojel1 0 = "nulla"
elojel1 x
   | x < 0 = "negatív"
   | x > 0 = "pozitív"
```

a függvény kiértékelése:

```
> elojel (-10)
```

Rekurzió (recursion)

A rekurzió a funkcionális nyelvek alapvezérlési szerkezete, a függvények hivatkozhatnak önmagukra és kölcsönösen egymásra.

7. feladat

Határozzuk meg két egész szám legnagyobb közös osztóját.

Kivonásos módszer

```
lnko :: Int -> Int -> Int
lnko a b
       | a > b = lnko (a-b) b
       | a < b = lnko a (b-a)
       | otherwise = a
```

Eukleidész módszere

```
euklid :: Integral a => a -> a -> a
euklid a b
   | b == 0 = a
   | otherwise = euklid b (mod a b)
```

a függvény kiértékelése:

```
> lnko 24 204
```

Margószabály (layout rule)

Az összetartozó kifejezéseket a baloldali margó alapján lehet megállapítani.

8. feladat

Határozzuk meg egy másodfokú egyenlet valós gyökeit.

```
masodE :: (Fractional a, Floating a, Ord a) => a -> a -> a -> (a, a)
masodE a b c
    | delta < 0 = error "Komplex gyokok"
    | otherwise  = (x1, x2)
    where
        x1 = (-b + sqrt delta) / n
        x2 = (-b - sqrt delta) / n
        delta = b * b - 4 * a * c
        n = 2 * a

delta a = 3 * a
```

a függvény kiértékelése:

```
> masodE 2 3 1
```

Melyik `delta` kifejezés értékelődik ki?

```
> delta 2
```

Mintaillesztés (pattern matching)

Az argumentumok mintaillesztése: a függvényértékét az a függvénytörzs határozza meg, amelyre a formális paraméter egy megadott minta alapján illeszkedik. A mintaillesztést és a feltételek megadását lehet együttesen is alkalmazni.

9. feladat

Határozzuk meg egy szám számjegyeinek összegét, szorzatát.

```
szOsszeg :: Int -> Int
szOsszeg 0 = 0
szOsszeg x = mod x 10 + szOsszeg (div x 10)

szSzorzat :: Int -> Int
szSzorzat 0 = 0
szSzorzat x
   | x < 10 = x
   | otherwise = mod x 10 * szSzorzat (div x 10)
```

Mintaillesztés

10. feladat

Határozzuk meg egy 3 elemű lista elemeinek összegét.

```
osszeg :: [Int] -> Int
osszeg [] = 0
osszeg [x] = x
osszeg [x,y] = x + y
osszeg [x,y,z] = x + y + z
```

a függvény kiértékelése:

```
> osszeg [10, 4, 2]
16
```

Mi történik a 4, 5 stb elemű listák esetében? Futási hiba adódik, ezek az esetek nincsenek letárgyalva:

```
> osszeg [10, 4, 2, 8]
*** Exception: ...: Non-exhaustive patterns in function osszeg
```

Halmazkifejezések (list comprehension)

Iteratív adatszerkezetek (listák, halmazok, sorozatok) elemeinek megadására alkalmazott jelölésrendszer.

11. feladat

Határozzuk meg a paraméterként megadott szám osztóit.

```
osztok :: Int -> [ Int ]
osztok n = [ i | i <- [1..n], mod n i == 0 ]
```

a függvény kiértékelése:

```
> osztok 60
[1,2,3,4,5,6,10,12,15,20,30,60]

> length (osztok 60)
12
```

Magasabb rendű függvények (high order function)

- argumentumuk lehet függvény, és visszatérítési értékük is lehet függvény,
- könyvtárfüggvények: map, filter, foldr, foldl, stb.
- map: két argumentuma van, az első egy függvény, amelyet alkalmaz a listaként megadott második argumentumára.

12. feladat

Határozzuk meg a paraméterként megadott számok négyzetgyökét.

```
> map sqrt [ 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 ]

[1.7320508075688772,2.0,2.23606797749979,2.449489742783178,
2.6457513110645907,2.8284271247461903,3.0]
```

Magasabb rendű függvények

13. feladat

Határozzuk meg a paraméterként megadott számok közül a páros számokat.

```
parosLista :: (Integral a) => [a] -> [a]
parosLista ls = filter even ls

> paroslista [1..20]

[2,4,6,8,10,12,14,16,18,20]
```

Az `even` könyvtárfüggvény, amelynek `True` a visszatérési értéke, ha a bemenet páros szám, ellenkező esetben `False`.

```
> even 10
True
```

Magasabb rendű függvények

14. feladat

Növeljük a paraméterként megadott szám értékét kettővel.

```
duplaz :: (a -> a) -> a -> a
duplaz f x = f (f x)

myInc :: (Num a) => a -> a
myInc x = x + 1

> duplaz myInc 10
12

> duplaz (* 10) 10
1000
```

A `duplaz` első paramétere egy függvény, amit kétszer alkalmaz a második paraméterére, a `my_inc` a megadott paraméter értékét 1-el növeli.

Lambda kifejezések, névtelen függvények

- a függvények egy alternatív definiálási módja, akkor használjuk, ha nem akarunk nevet választani egy adott segédfüggvénynek,
- nem tartalmazhatnak őrfeltételeket, mintaillesztéseket,
- vigyázni kell hogy a függvénydefiníció minden lehetséges helyzetre kiértékelődjön.

15. feladat

Növeljük a paraméterként megadott szám értékét 1-el.

```
myIncL :: (Num a) => a -> a
myIncL = \x -> x + 1
```

16. feladat

Növeljük egy listában megadott számok értékét 1-el.

```
listInc1 :: (Num a) => [a] -> [a]
listInc1 ls = map (\x -> x + 1) ls

listInc2 :: (Num a) => [a] -> [a]
listInc2 ls = [( x -> x + 1) k | k <- ls]

> listInc1 [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

Magasabb rendű függvények, részleges paraméterezés

- partial parameterization, curry-zés, Haskell Curry matematikus után,
- a függvényhívás megengedett kevesebb paraméterrel is.

17. feladat

Írjunk egy Haskell-függvényt, amely az `x` *és* `k` *bemenetekre, ahol* `k` *egész szám, meghatározza az* x^0, x^1, ..., x^k *értékeket.*

```
fugv1 :: (Integral a, Num b) => b -> a -> [b]
fugv1 x k = map (x ^) [0..k]

> fugv1 7 6
[1,7,49,343,2401,16807,117649]
```

A `map` függvény első paraméterét a hatványozó operátort, infix formában, részlegesen paraméterezve adtuk meg.

Magasabb rendű függvények, részleges paraméterezés

18. feladat

Írjunk egy Haskell-függvényt, amely az `x` *és* `k` *bemenetekre meghatározza az* 0^k, 1^k, ..., x^k *értékeket.*

A `^` operátort infix formában hívjuk úgy, hogy a `^` operátor első argumentuma rendre a `[0..x]` lista elemei legyen.

```
fugv2 :: (Integral a, Num b, Enum b) => b -> a -> [b]
fugv2 x k = map (^ k) [0..x]

> fugv2 7 6
[0,1,64,729,4096,15625,46656,117649]
```

Módosítjuk a zárójelezést, a `^` operátor prefix formában kerül meghívásra, más lesz az eredmény: k^0, k^1, ..., k^x

```
fugv2_ :: (Integral a, Num b) => a -> b -> [b]
fugv2_ x k = map ((^) k) [0..x]

> fugv2_ 7 6
[1,6,36,216,1296,7776,46656,279936]
```

Magasabb rendű függvények, részleges paraméterezés

A beépített operátor helyett megírjuk a saját hatványozó függvényünket, a `map` függvénynek ezt adjuk meg paraméternek. A `myPow1` függvény első paramétere az alap, második a hatványkitevő lesz.

```
myPow1 :: (Integral a) => a -> a -> a
myPow1 x n
    | n < 0 = error "Negativ kitevo"
    | n == 0 = 1
    | even n = temp * temp
    | otherwise = x * temp * temp
        where
        temp = myPow1 x (div n 2)
```

Magasabb rendű függvények, részleges paraméterezés

A `fugvA`-ban prefix formában használjuk `myPow1` függvényt, azért hogy az x^0, x^1, ..., x^k értékeket határozza meg:

```
fugvA :: Integral a => a -> a -> [a]
fugvA x k = map (myPow1 x) [0..k]

> fugvA 7 6
[1,7,49,343,2401,16807,117649]
```

A `fugvB`-ben infix formában, azért hogy a 0^k, 1^k, ..., x^k értékeket határozza meg:

```
fugvB :: Integral a => a -> a -> [a]
fugvB x k = map (`myPow1` k) [0..x]

> fugvB 7 6
[0,1,64,729,4096,15625,46656,117649]
```

Magasabb rendű függvények, részleges paraméterezés

A `fugvC`-ben alkalmazásra kerül a beépített `flip` függvény, amely segítségével a paraméterek sorrendjét lehet megváltoztatni, így most is a 0^k, 1^k, ..., x^k értékeket határozza meg a kiértékelés.

```
fugvC :: Integral a => a -> a -> [a]
fugvC x k = map (flip myPow1 k) [0..x]
```

```
> fugvC 7 6
[0,1,64,729,4096,15625,46656,117649]
```

A `flip` függvény megértéséhez figyeljük meg a következő kiértékelések eredményeit.

```
> myPow1 2 10
1024
> flip myPow1 2 10
100
```

Magasabb rendű függvények, részleges paraméterezés

19. feladat

Válasszuk ki egy adott listából az x-el osztható számokat, használjuk a `filter` *könyvtárfüggvényt:*

```
oszthato :: (Integral a) => a -> a -> Bool
oszthato x y
    | mod y x == 0 = True
    | otherwise = False

oszthato_ :: (Integral a) => a -> a -> Bool
oszthato_ x y = mod y x == 0

fugv :: (Integral a) => a -> [a] -> [a]
fugv x ls = filter (oszthato x) ls

> fugv 7 [1..100]
[7,14,21,28,35,42,49,56,63,70,77,84,91,98]
```

A lista típus

- ugyanolyan típusú elemek sorozata, ahol az elemek száma változó. Jelölésére a szögletes zárójelt használjuk: `[]`, sorszámozásuk nullától kezdődik.
  - `[]` - egy üres listát mintáz,
  - `[x]` - egy egyelemű listát mintáz,
  - `[x, y]` - egy kételemű listát mintáz,
  - `(k : ve)` - egy olyan listát mintáz, melynek első eleme `k`, `ve` pedig a lista vége, ahol `k` elem, `ve` lista típusú,
- Függvények listákon, lista elemeinek összege (`sum`), lista hossza (`length`), lista elemeinek megfordítása (`reverse`), stb:

```
> sum [3, 2, 10, 7, 5]
27
> length [3, 2, 10, 7, 5]
5
> reverse [3, 2, 10, 7, 5]
[5,7,10,2,3]
```

Operátorok listákon

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

- hozzáad egy új elemet a listához, amelyet a lista elejére tesz, a típusdefiníció általános lista feldolgozását teszi lehetővé,
- hozzárendeli a lista első elemét egy azonosítóhoz, a lista többi elemét, pedig egy másik nevű azonosítóhoz rendeli hozzá

```
> ls1 = [1, 2, 3, 4]
> ls1_ = 0 : ls1
> print ls1_
[0, 1, 2, 3, 4]
```

```
> k : ve = "Hello Vilag"
> print k
'H'
> print ve
"ello Vilag"
```

```
> ls2 = "apientia"
> ls2_ = 'S': ls2
> print ls2_
"Sapientia"
```

Operátorok listákon

```
> ls3 = [[1,2,3,4], [1,2,3], [1,2]]
> ls3_ = [1..5] : ls3
> print ls3_
[[1,2,3,4,5], [1,2,3,4], [1,2,3], [1,2]]

> ls4 = [0,5..40]
> print ls4
[0, 5, 10, 15, 20, 25, 30, 35, 40]

> ls5 = [-3, -6.. -20]
> print ls5
[-3, -6, -9, -12, -15, -18]
```

Operátorok listákon

```
> ls6 x = [2^x, length (show (2 ^ x))]
> print ls6 10
[1024,4]

> ls7 x = 3^x : length (show (3^x)): (ls6 x)
> print ls7 10
[59049, 5, 1024, 4]

> ls8 = ['a'..'z']

> ls9 = ['A' ..'Z'] ++ ls8
> print ls9
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

> ls9 !! 2
'C'
```

Függvények listákon

`head :: [a] -> a`
visszatéríti egy lista első elemét, nem alkalmazható üres listára, a típusdefiníció általános lista feldolgozását teszi lehetővé.

```
myHead :: [a] -> a
myHead [] = error "ures lista"
myHead (k : ve) = k

> myHead [1..10]
1

> myHead ["alma", "korte", "barack", "szilva"]
"alma"

> head [1..10] -- a konyvtarfuggveny
1
```

Függvények listákon

`tail :: [a] -> [a]`
egy listát térít vissza, amelyben nem szerepel az eredeti lista első eleme, nem alkalmazható üres listákra.

```
myTail :: [a] -> [a]
myTail [] = error "ures lista"
myTail (k : ve) = ve

> myTail [1..10]
[2,3,4,5,6,7,8,9,10]

> myTail ["alma", "korte", "barack", "szilva"]
["korte", "barack", "szilva"]

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