Funkcionális programozás
2. előadás

MÁRTON Gyöngyvér

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

2025, tavaszi félév

Miről volt szó?

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

Miről lesz szó?

Haskell:

- Az első lépések
- GHC parancsok
- Alaptípusok
- feladatok: faktoriális számítás
- típusosztályok
- típusdefiníciók
- megjegyzések használata
- könyvtármodul importálása

Haskell, az első lépések

Ha a `VSC`-ben történt a kódolás, és ha az állomány neve: `elso.hs`, akkor a következőképpen lehet meghívni a `terulet` függvényt:

- válasszuk ki a `File/Open Folder`-el a munkamappát,
- egy `Terminal` ablakba indítsuk el a `gchi`-vel a Haskell-t,
- írjuk be a prompt után `:l elso.hs`,
- ekkor minden függvény, amely az állományban van kiértékelhetővé válik, hívjuk meg tehát a függvényt, pl: `> terulet 10`
- az állomány betöltéséhez még a következőképpen is eljárhatunk: írjuk be a prompt után `:l`, majd válasszuk ki a `Terminal/Run Active File` menüpontot,
- ha változtatást végzünk a kódsoron, akkor le kell menteni az állományt, majd a `:r` Haskell paranccsal újra be lehet tölteni a módosított file-t.

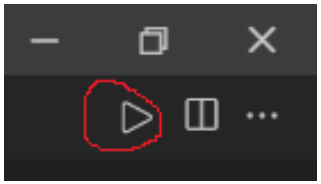
Haskell, az első lépések

A Haskell állományt a következőképpen is futtathatjuk:

- az állományt egészítsük ki egy `main` függvénnyel, pl:

```
main :: IO ()
main = do
    print (terulet 20.75)
```

- ha sikeresen telepítettük a `Code Runner` Haskell kiterjesztést, akkor a programunk futtatható lesz a szerkesztő jobb felső sarkában található gomb lenyomásával:



- egy állomány természetesen nem tartalmazhat két `main` függvényt
- mielőtt futtatnánk az állományt ne felejtsük el lementeni

GHC parancsok

lehet rövidített formában is használni őket, a parancs kezdőbetűjével:

- `:load fnev.hs` az fnev.hs nevű állomány betöltése,
- `:reload fnev.hs` az fnev.hs nevű állomány újrabetöltése, rövidítve: `:r`
- `:type kif` a kif kifejezése típusának a lekérdezése, rövidítve: `:t kif`
- `:?` az összes GHC parancs lekérdezése,
- `:quit` kilépés a GHC-ből,
- `:set +t` a kiértékelés után a kifejezés típusa is megjelenik
- `:unset +t` az előző beállítás visszavonása
- `:set +s` a kiértékelés után megjelenik az eltelt idő és a lefoglalt bájtok száma
- `:!cd` az aktuális könyvtár (directory) állapot lekérdezése
- `:cd C:\Diak` könyvtár (directory) változtatás
- ...

Alaptípusok

- Bool: logikai típus, két értékkel: False, True,
- Char: egyetlen (Unicode) karakter eltárolására alkalmas, idézőjel közé kell írni az értéket:

  `'A', '\n', '+'`
- String: karakterláncok kezelésére alkalmas, macskaköröm közé kell írni az értékeket: `"Hello Haskell", "10 * 0 = 0"`,
- Int: rögzített pontosságú egész számok, -2^{31} közötti 2^{31} értékek kezelésére alkalmas,
- Integer: tetszőleges pontosságú egész számok kezelésére alkalmas. Az Int és Integer közötti választást elsősorban a hatékonyság alapján kell eldönteni,
- Float, Double: valós számok kezelésére alkalmas. A Double 64-bites lebegőpontos ábrázolású, használata ajánlott, ellentétben a Float-tal, aminek használata kevésbé ajánlott.

Típusosztályok

Egy függvény argumentumai nem csak egy adott típushoz tartozó értékek lehetnek. A Haskell bevezeti a "típusváltozó" fogalmát, amit azt jelenti hogy a függvényargumentumok specifikációjakor típusosztályokat is megadhatunk. Jelölésükre az angol ábécé kis betűit használjuk.

A `terulet` függvény szignatúráját a következőképpen módosíthatjuk:

```
terulet :: (Ord t, Floating t) => t -> t
terulet r = if r < 0 then error "Rossz benmenet!" else r * r * pi
```

A `sort` függvény szignatúráját a következőképpen kérdezhetjük le:

```
> import Data.List
> :t sort
sort :: Ord a => [a] -> [a]
```

Ez azt jelenti, hogy a `sort` függvény minden olyan típusú adatra meghívható, amely az `Ord` típusosztályba tartozik. Például egész számok rendezése mellett karakterláncokat is rendezhetünk:

```
> sort ["pinty", "rigo", "cinke", "harkaly"]
```

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
```

- a típusosztályok a `class` kulcsszóval vannak definiálva, de nem ugyanazt jelentik, mint az OOP-ben használt `class`
- a definícióban használt `class` szerepe, az hogy összekapcsol több olyan típust, amelyeken ugyanazok a műveletek hajthatók végre,
- típusosztály: több függvénynek a deklarációját jelenti.

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
```

Típusosztályok

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

- a Real, illetve Enum típusosztályokból van származtatva,
- az Enum: azok a típusok, amelyeknek az értékei felsorolhatók,
- a Real a toRational függvényen keresztül biztosítja a Rational típusra a típuskonverziót,
- 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
> y = toRational 0.75
> y
3 % 4
> :t y
y :: Rational
```

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, pl. 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

Numerikus típusok közötti konverziók:

Source type	Destination type				
	Double, Float		Int, Word	Integer	Rational
Double, Float	fromRational	. toRational	truncate *	truncate *	toRational
Int, Word	fromIntegral		fromIntegral	fromIntegral	fromIntegral
Integer	fromIntegral		fromIntegral	N/A	fromIntegral
Rational	fromRational		truncate *	truncate *	N/A

[a] Instead of truncate, you could also use round, ceiling, or floor.

táblázat forrása: Sullivan, Goerzen and Stewart: Real World Haskell

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,

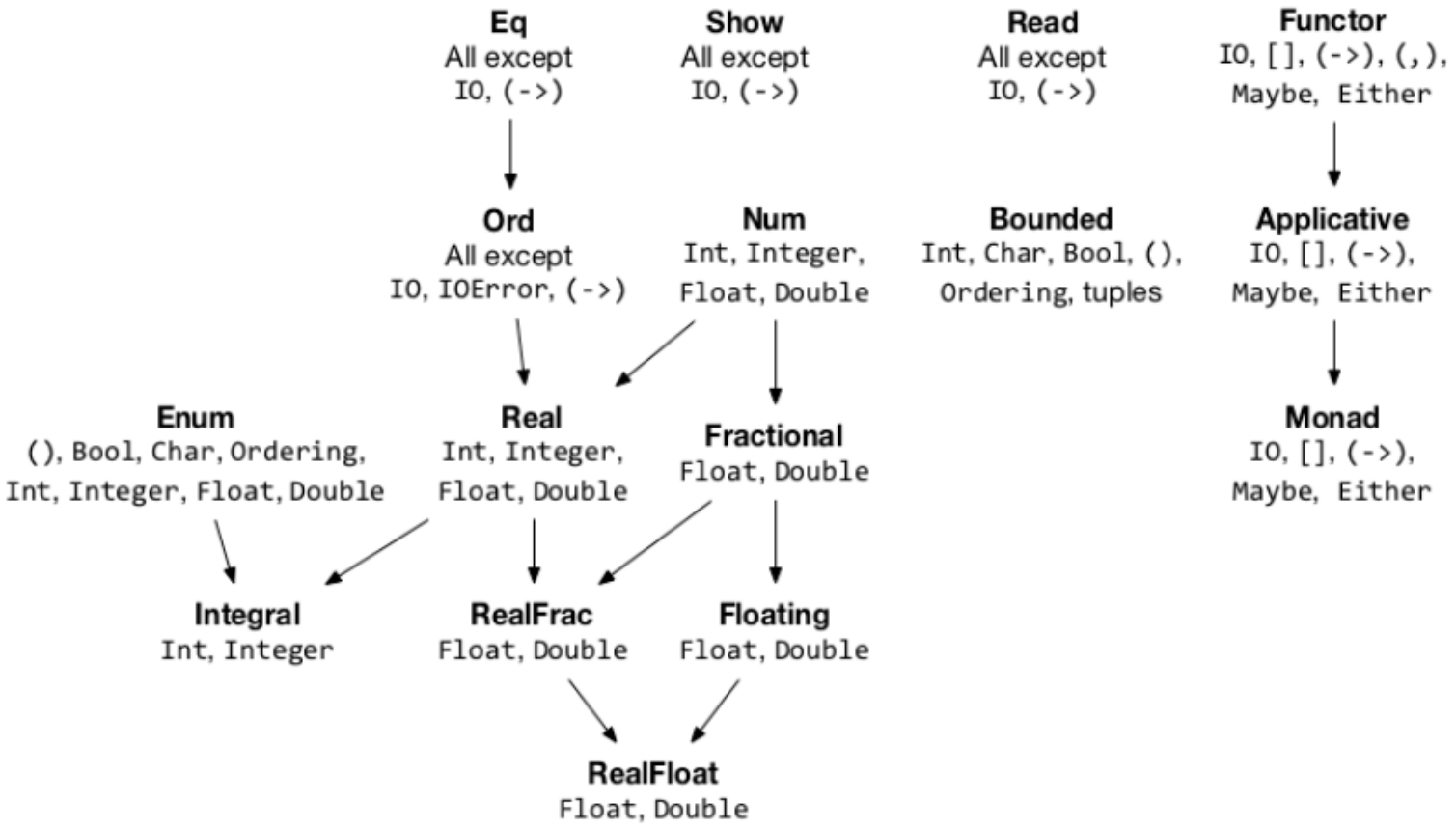
- használatakor meg kell adni, hogy milyen típusra alakítunk, ehhez a :: jelet kell használni,
- magába foglalja az alaptípusokat (Bool, Char, String, Int, Integer, Float, Double, Lista, Tuple):

```
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 lekérdezés 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

> isDigit 'w'
False

> isAlpha 'a'
True

> 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
```