

HASKELL



Tartalom – 1

1. előadás

- Bevezetés
- A Haskell mint funkcionális nyelv
 - típusok és értékek
 - függvények és operátorok
 - adatkonstruktorok tulajdonságai
 - mintaillesztés, örök
 - vezérlési szerkezetek
 - a forráskód beosztása
- A Haskell mint lusta nyelv
 - végtelen adatszerkezetek
 - listák építése

Tartalom – 2

2. előadás

- A Haskell mint lusta nyelv (folyt.)
 - fenékérték
 - szigorú kiértékelés kikényszerítése
- A típusnyelv kiterjesztése
 - típusosztályok, többszörös terhelés
 - beépített típusosztályok
 - származtatás

Tartalom – 3

3. előadás

- A típusnyelv kiterjesztése (folyt.)
 - számok kezelése
 - Peano-számok megvalósítása
 - többargumentumú típusosztályok
- A Haskell modulnyelve

4. előadás

- „Imperatív” elemek a Haskellben
 - hibakezelés
 - állapotkezelés
 - állapotkezelés hibakezeléssel kombinálva

Tartalom – 4

5.-6. előadás

- „Imperatív” elemek a Haskellben (folyt.)
 - monádok
 - a `do` jelölés
 - imperatív stílusú programozás
 - monádok aritmetikája
 - a lista mint monád
 - a Monad könyvtár
 - ki- és bevitel

Ajánlott olvasmány

- „A Haskell programozó evolúciója” (Fritz Ruehr)

BEVEZETÉS



Bevezetés

A Haskell eredete

- *Haskell* Brooks Curry matematikus tiszteletére (v.ö. *curry* és *uncurry*)
- 1987: az első Haskell – cél: egy szabványos, lusta kiértékelésű, tisztán funkcionális nyelv
- 1999: Haskell 98 – lényegében azonos a Haskell 1.4-gyel

Interaktív környezetek

- Hugs – kicsi és gyorsan fordít, tanuláshoz ideális
- GHC – nagyobb, sok kiegészítővel, nagyon gyors kódot fordít

Források, irodalom

- Haskell.org
- A Gentle Introduction to Haskell (Paul Hudak, John Peterson, Joseph H. Fasel)
<http://www.haskell.org/tutorial>
- Haskell 98 Report & Library Report
<http://www.haskell.org/onlinereport>

A HASKELL MINT FUNKCIONÁLIS NYELV



Típusok és értékek – 1

Szintaktika

- névadás egyenlet formájában: $név = érték$
- Church-féle típusmegkötés: $név/érték [, név/érték...] :: típus$
- a típus- és adatkonstruktorok nagybetűvel kezdődnek, és prefix pozícióban állnak

Egyszerű típusok

- logikai értékek: $True :: Bool$
- egész számok
 - korlátos: $5 :: Int$
 - „BigNum”: $908789326459032645987326495819280921 :: Integer$
- lebegőpontos számok
 - egyszeres pontosságú: $2.71828 :: Float$
 - dupla pontosságú: $3.14159 :: Double$
- karakterek: $'c' :: Char$

Típusok és értékek – 2

Összetett (polimorf) típusok

- listák: `3:6:1:2:[] == ([3,6,1,2] :: [Integer])`
- füzérek: `"haskell" == (['h','a','s','k','e','l','l'] :: [Char])`
- ennesekek: `('a', 1.23, True) :: (Char, Double, Bool)`
- függvények: `take :: Int -> [a] -> [a]`

Felhasználói típusok

- típuszinonima
 - meglévő (összetett) típusok rövid neve
 - a típusnév bárhol felcserélhető a definíciójával
 - `type String = [Char]`
- típuskonstruktor
 - új típust hoz létre
 - mintaként illeszthető
 - `data Bool = True | False`
 - `data Tree a = Leaf | Node a (Tree a) (Tree a)`

Típusok és értékek – 3

Beépített típuskonstruktorok

- rendezettség: `data Ordering = LT | EQ | GT`
- feltételes típus: `data Maybe a = Nothing | Just a`
- diszjunktív unió: `data Either a b = Left a | Right b`

Felhasználói típusok (folyt)

- fedőtípus
 - a típuszinonimához hasonlóan egy meglévő típus átnevezése
 - új típust hoz létre (a típuszinonimától eltérően, de a típuskonstruktorhoz hasonlóan), illeszthető mintaként
 - *nincs plusz memóriaigény, csak a típusellenőrzéskor van szerepe*
 - `newtype Natural = Natural Integer`
 - `Natural 15 :: Natural`

Típusok és értékek – 4

Felhasználói típusok (folyt)

- mezőnevek
 - egy adatkonstruktor argumentumai elnevezhetők (v.ö. rekord)
 - ```
data Tree a = L | N { value :: a,
 left :: Tree a,
 right :: Tree a }
```
  - adatstruktúra megadása, mintaillesztés:
    - ```
N 1 L L
```
 - ```
N { value = 1, left = L, right = L }
```
  - kiválasztó függvények:
    - ```
value :: Tree a -> a
```
 - ```
left, right :: Tree a -> Tree a
```
    - ```
value (N 1 L L) == 1
```
 - a hibás használat csak futási időben derül ki:

```
value L
```
 - értékfelülírás:

```
(N 1 L L) { value = 2 } == N 2 L L
```

Függvények és operátorok – 1

Lambda függvények

- `\arg1 [arg2 ...] -> törzs`
- `\x -> \y -> x+y`
- `\x y -> x+y`

Nevesített függvények

- a típusmegkötés explicit módon megadható
- általában *curried* (kaszkádosított, részlegesen alkalmazható) alakúak (ld. operátorok)
- `-- add x y = x és y összege`
`add :: Integer -> Integer -> Integer`
`add x y = x+y`
`-- ugyanaz, mint: add = \x y -> x+y`

Operátorok

- kétargumentumú, *curried* (kaszkádosított, részlegesen alkalmazható) függvények
- ha a függvény neve *szimbólumokból* áll, akkor operátor, ha *alfanumerikus*, akkor prefix függvény

Függvények és operátorok – 2

Operátorok (folyt)

- az operátorok prefix alakja: $(+) = \lambda x y \rightarrow x+y$
- $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
 $f . g = \lambda x \rightarrow f (g x)$
- $f . g x = f (g x)$ szintaktikusan hibás! (v.ö. *kétargumentumú*)
- szeletek (sections): az operátorok is részlegesen alkalmazhatók
 - $(x+) \equiv \lambda y \rightarrow x+y$ és $(x-) \equiv \lambda y \rightarrow x-y$
 - $(+y) \equiv \lambda x \rightarrow x+y$ de $((-)y) \equiv \lambda x \rightarrow x-y$
- ``f`` a függvények infix alakja, pl. `3 `add` 4 == 5`
- kötés megadása
 - balra kötő: `infixl 6 *`
 - jobbra kötő: `infixr 3 &&`
 - nem kötő: `infix 4 /=`
 - alapértelmezés: `infixl 9`

Adatkonstruktorok tulajdonságai

Infix adatkonstruktorok

- a nevük csak szimbólumokból áll, és kettősponttal kezdődik
- ugyanúgy van precedenciájuk, mint az infix függvényeknek
- pl. tört definiálása:
 - `data Fraction = Integer :/ Integer`
 - `3 :/ 5 :: Fraction`

Adatkonstruktorok mint függvények

- az adatkonstruktorok függvényként is használhatók
- `map Just [1,2] == ([Just 1, Just 2] :: [Maybe Integer])`
- ez igaz az ennesre is!
 - `(,) True 'x' == (True, 'x')`
 - `(,,) "ok" 2 :: a -> (String, Integer, a)`

Mintaillesztés, örök

Mintaillesztés

- bármely adatkonstruktor használható mintában
- alternatív mintákat több egyenlettel adunk meg
- `_`: univerzális minta, mindenesjel, mindenre illeszkedik
- réteges minta: `név @ minta`
- `take`

```

take      :: Int -> [a] -> [a]
take 0 _  = []
take _ []  = []
take n (x:xs) = x : take (n-1) xs

```

Örök

- nem minden eset választható szét mintákkal
- `ór` = a klóztörzs kiértékelhetőségének feltétele
- `compare x y`

<code>x == y</code>	<code>= EQ</code>
<code>x <= y</code>	<code>= LT</code>
<code>otherwise</code>	<code>= GT</code>

Vezérlési szerkezetek – 1

Esetszétválasztás

- mintaillesztéses esetszétválasztás

```
take      :: Int -> [a] -> [a]
take m xs = case (m,xs) of
    (0,_)      -> []
    (_,[])     -> []
    (m,x:xs)   -> x : take (m-1) xs
```

- feltételes kifejezés

- $\max x y = \text{if } x \geq y \text{ then } x \text{ else } y$

- szintaktikus édesítőszer: $\text{if } e \text{ then } e1 \text{ else } e2$ ekvivalens alakja:

```
case e of
    True  -> e1
    False -> e2
```

Vezérlési szerkezetek – 2

Lokális érvényű deklarációk

- let-kifejezés

- a deklarációk egy *kifejezésen* belül érvényesek

- ```
distance (x1,y1) (x2,y2) = let xd = x1-x2
 yd = y1-y2
 in sqrt(xd*xd + yd*yd)
```

- where-deklaráció

- a deklarációk egy (esetleg több őrzött esetből álló) *deklaráción* belül érvényesek

- tipikusan segédfüggvény definiálásakor használjuk

- ```
gcd x y = gcd' (abs x) (abs y)
  where gcd' x 0 = x
        gcd' x y = gcd' y (x `rem` y)
```

A forráskód beosztása (layout)

Mi választja el az egyes deklarációkat, kifejezéseket egymástól? Ekvivalens-e a következő két kifejezés:

```
let y    = a*b
    f x = (x+y)/y
in f c + f d
```

```
let y    = a*b f
    x    = (x+y)/y
in f c + f d
```

A válasz: nem. A jelentés a *beosztástól* (layout) függ. A forráskód kétdimenziós elrendezésű:

- alapvetően intuitív, könnyen olvasható;
- a *where*, *let*, *of* kulcsszók utáni első nemszóköz karakter határozza meg a deklarációk, ill. minták kezdőoszlopát;
- egy *beágyazott* blokk kezdőoszlopa mindig beljebb legyen, mint a *beágyazó* blokké;
- egy deklarációnak, kifejezésnek vége, ha valami a blokk kezdőoszlopától balra kezdődik.

A tagolás *explicit* módon is megadható: { ; }, pl. ha több deklarációt szeretnénk egy sorba írni.

```
let { y    = a*b
    ; f x = (x+y)/y
    }
in f c + f d
```

```
let y = a*b; f x = (x+y)/y
in f c + f d
```

A HASKELL MINT LUSTA NYELV



Kérdés

Mi a különbség a *függvényértékek* és az *egyéb értékek* között?

Válasz

Semmi!

egyéb érték = argumentum nélküli függvény

Végtelen adatszerkezetek

Deklaráció

- egyesek végtelen listája: `ones = 1 : ones`
- egészek n-től felfelé: `upFrom n = n : upFrom (n+1)`
- négyzetszámok: `squares = map (^2) (upFrom 0)`
- Fibonacci sorozat – 1. változat

```
fib = 1 : 1 : fib +: (tail fib)
      where (x:xs) +: (y:ys) = x+y : xs +: ys
```

- Fibonacci sorozat – 2. változat

```
fib @ (_:tfib) = 1 : 1 : zipWith (+) fib tfib
```

Felhasználás

- `take 5 ones == [1,1,1,1,1]`
- `take 7 squares == [0,1,4,9,16,25,36]`
- `take 10 fib == [1,1,2,3,5,8,13,21,34,55]`

Listák építése – 1

Listanézet (List Comprehension)

- a listaépítés és -transzformálás tömör, kifejező formája
- lefedi a `map` és a `filter` függvényeket, és még sokkal többet
- általános alak: `[elemkif | minta <- listakif, örkkif, ...]`
- `map f xs = [f x | x <- xs]`
- `filter p xs = [x | x <- xs, p x]`
- összes lehetséges pár (Descartes-szorzat):

```
cartesian :: [a] -> [b] -> [(a,b)]
cartesian xs ys = [ (x,y) | x <- xs, y <- ys ]
```

```
cartesian [1,2] ['a','b'] == [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

- gyorsrendezés – a lehető legtömörebben

```
quicksort [] = []
quicksort (x:xs) =
  quicksort [ y | y <- xs, y < x ] ++
  x : quicksort [ y | y <- xs, y >= x ]
```


Listák építése – 2

Listanézet (folyt)

- Fibonacci sorozat – 3. változat

```
fib = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
      where _:tfib = fib
```

Számtani sorozatok

- számtani sorozat függvényvel

```
fromThenTo n n' m = nn'm
      where nn'm = takeWhile p (n : map ((n'-n) +) nn'm)
            p | n' >= n = (m >=)
              | otherwise = (m <=)
```

```
fromThenTo 1 3 10 == [1,3,5,7,9]
```

- számtani sorozat szintaktikai édesítőszerrel

- $[3..15] == [3,4,5,6,7,8,9,10,11,12,13,14,15]$

- $['a', 'c'..'f'] == "ace"$

- $[0.0, 1.1..] \implies [0.0, 1.1, 2.2, 3.3, \dots]$ végtelen lista

Fenekérték – 1

1. próbálkozás

- `bot = bot`
- Mi a típusa? `bot :: a`
- ha kiértékeljük, végtelen ciklusba esünk

2. próbálkozás

- `bot | False = bot`
- Mi a típusa? `bot :: a`
- ha kiértékeljük, futási hibát kapunk
- jelölése: \perp (ejtsd: *fenékérték* vagy bottom)
- az okozott hiba *fatális*, a program leáll
- ha nem értékeljük ki, nem okoz gondot: `(\x -> 1) bot == 1`
- a Standard Prelude-ben `undefined`-nak hívják

Fenekérték – 2

Hibajelzés

- \perp visszaadása: jó, de nem túl „bőbeszédű”
- `error` hibajelző függvény
 - `error :: String -> a`
 - `head :: [a] -> a`
`head (x:_) = x`
`head [] = error "head{PreludeList}: head []"`
 - szemantikai értelemben az `error` függvény értéke \perp

Szigorú kiértékelés kikényszerítése

Szigorú adatkonstruktorok

- általában előfordulhat, hogy egy adatszerkezet egy részét soha nem értékeljük ki

```
fst ("ok", undefined) == "ok"
```

- időnként szemantikailag indokolt lehet csak teljesen kiértékelhető struktúrák megengedése

```
data Fraction = !Integer :/ !Integer
(\(x :/ y) -> x) (1 :/ undefined) ==> undefined
```

- növeli a hatékonyságot

Szigorú kiértékelés

- $f \ \$! \ x$ hívás x *legfelső szintjét* kiértékeli, és alkalmazza rá f -et
- $(\lambda x \rightarrow 1) \ undefined == 1$
- $(\lambda x \rightarrow 1) \ \$! \ undefined \implies undefined$
- $(\lambda x \rightarrow 1) \ \$! \ (undefined, undefined) == 1$

A TÍPUSNYELV KITERJESZTÉSE

Típusosztályok, többszörös terhelés – 1

A polimorfizmus változatai

- **Paraméteres ~:** ez a „megszokott”, típusváltozót használó. Az elvégzendő művelet mindig *ugyanaz*, nem (teljesen) használja ki az argumentum típusát.
- **Ad-hoc ~:** közismertebb néven többszörös terhelés vagy overloading. Itt csak a szintaktika azonos, a számítás teljesen különböző lehet minden típusra. Például
 - az 1, 2, ... állandók jelenthetnek egész és lebegőpontos számokat is
 - az aritmetikai operátorok, pl. a + vagy a * sokféle számtípuson működnek
 - az egyenlőségvizsgáló operátorok, pl. az == és a /= nagyon sokféle típusra működnek
- Jó hír:
 - a Haskellben a felhasználó is definiálhat többszörösen terhelt függvényeket, sőt,
 - a meglévő, többszörösen terhelt függvényeket *kiterjesztheti újabb típusokra*.
- Mi a `member_of` függvény típusa?

```
x `member_of` [] = False
```

```
x `member_of` (y:ys) = x == y || x `member_of` ys
```

Nem egyszerűen `a -> [a] -> Bool!`

Típusosztályok, többszörös terhelés – 2

Típusosztály, példány, kontextus

- egy típus *példánya* egy *típusosztálynak*, ha a típushoz tartozó értékekre alkalmazható a típusosztályba tartozó összes függvény

- például: egyenlőségi osztály

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

- `Eq a` fejezi ki azt a *kényszert*, hogy az `a` típusnak az `Eq` osztály egy példányának kell lennie

- egy típuskifejezésre vonatkozó kényszert (pl. `Eq a`) a kifejezés *kontextusának* nevezünk

- `(==) :: Eq a => a -> a -> Bool`

- ez alapján: `member_of :: Eq a => a -> [a] -> Bool`

- példányosítás:

```
data Fraction = !Integer :/ !Integer
```

```
instance Eq Fraction where
```

```
    (a:/b) == (x:/y) = a*y == x*b
```

```
(3 :/ 5 == 6 :/ 10) == True
```

Típusosztályok, többszörös terhelés – 3

További kontextusok

- lehet (kell, hogy legyen) kontextusa a példányosításnak:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
instance Eq a => Eq (Tree a) where
    Leaf == Leaf = True
    Node v1 l1 r1 == Node v2 l2 r2 = (v1,l1,r1) == (v2,l2,r2)
    _ == _ = False
```

- lehet saját kontextusa az egyes *metódusoknak*:

```
class MyClass a where method :: Eq b => b -> b -> a
```

Alapértelmezett metódusmegvalósítás

- class Eq a where


```
(==), (/=) :: a -> a -> Bool

-- Minimal complete definition: (==) or (/=)
x == y = not (x/=y)
x /= y = not (x==y)
```


Típusosztályok, többszörös terhelés – 4

Öröklődés

- a típusosztályok *öröklődés* útján kiterjeszthetők

- `class Eq a => Ord a where`

```

compare                :: a -> a -> Ordering
(<), (<=), (>=), (>)   :: a -> a -> Bool
max, min                :: a -> a -> a

```

```

compare x y | x==y      = EQ
            | x<=y      = LT
            | otherwise = GT

```

- a kontextusban elegendő az *alosztályt* megadni, az *ősosztály* kiírása redundáns

```
quicksort :: Ord a => [a] -> [a]
```

- a többszörös öröklődés megengedett, de a szokásos problémák nem jönnek elő, mivel egy név csak egy osztályba tartozhat, azaz átfedés eleve nem lehet

```
class (A a, B a) => C a where ...
```

Típusosztályok, többszörös terhelés – 5

Típuskonstruktorok osztályai

- egy típusosztály nemcsak típusállandók, hanem típuskonstruktorok osztálya is lehet

- `class Functor f where`
`fmap :: (a -> b) -> (f a -> f b)`

Itt `f` egy típuskonstruktor.

- `instance Functor Tree where`
`fmap f Leaf = Leaf`
`fmap f (Node v l r) = Node (f v) (fmap f l) (fmap f r)`

```
instance Functor [] where
  fmap = map
```

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Beépített típusosztályok – 1

- `Eq a`, `Eq a => Ord a` és `Functor f` már ismert
- korlátossági osztály

```
class Bounded a where
    minBound, maxBound :: a
```

- enumerációs osztály számtani sorozatok létrehozásához

```
class Enum a where
    succ, pred          :: a -> a
    toEnum              :: Int -> a
    fromEnum           :: a -> Int
    enumFrom           :: a -> [a]           -- [n..]
    enumFromThen       :: a -> a -> [a]      -- [n,m..]
    enumFromTo         :: a -> a -> [a]      -- [n..m]
    enumFromThenTo     :: a -> a -> a -> [a] -- [n,n'..m]
```

- *monadikus* osztály: `Monad`, lásd később
- számosztályok, lásd később

Beépített típusosztályok – 2

A Show típusosztály

- értékek füzérré alakítására szolgál (kiíráshoz)
- `show (2, 'a') == "(2, 'a')"`
- fa kiírása:

```
showTree           :: Show a => Tree a -> String
showTree Leaf     = "<>"
showTree (Node v l r) = "<" ++ showTree l ++ "|" ++
                        show v      ++ "|" ++
                        showTree r ++ ">"
```

Ezzel az a gond, hogy a költsége négyzetes, mivel ++ költsége arányos a lista hosszával.

- fa kiírása gyűjtőargumentummal:

```
showsTree          :: Show a => Tree a -> String -> String
showsTree Leaf    = ("<>")
showsTree (Node v l r) = ('<':) . showsTree l . ('|':) .
                        shows v      . ('|':) .
                        showsTree r . ('>':)
```

Beépített típusosztályok – 3

A Show típusosztály (folyt)

- *hozzáíró függvény* (showing function): `type ShowS = String -> String`

- primitív hozzáíró függvények: `(' | ' :)`, `("<>"++)`

- `showsTree` fából hozzáíró függvényt állít elő

```
showsTree :: Show a => Tree a -> ShowS
```

- `class Show a where`

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

```
    showsPrec :: Int -> a -> ShowS
```

```
    showList  :: [a] -> ShowS
```

- `showsPrec` első argumentuma a különböző precedenciaszintek kezelésére használható

- `showList` azért, hogy a lista a szokásostól eltérő alakban is megjelenhessen, ld. `String`

- `instance Show a => Show (Tree a) where`

```
    showsPrec _ = showsTree
```

- `instance Show a => Show [a] where`

```
    showsPrec _ = showList
```

Származtatás

Típusosztály példányainak automatikus származtatása

- bizonyos típusosztályok példányainak megírása unalmas, mechanikus munka
- az ilyen osztályok példányai automatikusan előállíthatók
- ```
data Tree a = Leaf | Node a (Tree a) (Tree a)
 deriving (Eq, Ord, Show)
```
- Eq származtatása: az intuíciónak megfelelő
- Ord származtatása: lexikografikus sorrend, balról jobbra
- Show származtatása: a Haskell szintaktikának megfelelő kiírás

## Számok kezelése – 1

---

### A Haskell által ismert, beépített számtípusok

- véges és korlátlan egészek
- egész típusokból képzett *arányok*, racionális számok
- egyszeres és dupla pontosságú, valós és komplex lebegőpontos számok

Ezek a típusok az átjárhatóság kedvéért *típusosztályok hierarchiájába* vannak szervezve.

### A Num osztály

- minden számosztály őse
- azokat a műveleteket adja, amelyeknek minden számra értelmesnek kell lennie
- ha egy típus a példánya, akkor alapvető aritmetikai műveletek már végezhetőek az értékein

```
class (Eq a, Show a) => Num a where
 (+), (-), (*) :: a -> a -> a
 negate :: a -> a -- the '-' prefix operator
 abs, signum :: a -> a
 fromInteger :: Integer -> a
 fromInt :: Int -> a
```

## Számok kezelése – 2

---

### Az Integral osztály

- egész számok ábrázolására
- példányai az Int és Integer típusok
- ```
class (Real a, Enum a) => Integral a where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod    :: a -> a -> (a,a)
    even, odd          :: a -> Bool
    toInteger          :: a -> Integer
    toInt              :: a -> Int
```

A Fractional osztály

- törtek és lebegőpontos számok ábrázolására szolgáló őosztály
- ```
class (Num a) => Fractional a where
 (/) :: a -> a -> a
 recip :: a -> a
 fromRational :: Rational -> a
 fromDouble :: Double -> a
```



## Számok kezelése – 3

---

### A Floating osztály

- a `Fractional` osztály leszármazottja
- lebegőpontos számok ábrázolására
- példányai a `Float` és `Double` típusok
- metódusai szögfüggvények és -konstansok

### Arányok ábrázolása

- a `Fractional` osztály példánya a `Ratio` típus
- az `Integral` osztály példányaiból képes arányokat létrehozni
- *absztrakt adattípus* az arányok ábrázolásához:

```
data Integral a => Ratio a = !a :% !a deriving (Eq)
```

- típuszinonima a racionális számokhoz: `type Rational = Ratio Integer`
- absztrakt adattípus, ezért a `:%` adatkonstruktor nem látszik ki

```
(%) :: Integral a => a -> a -> Ratio a
```

```
3 % 6 ==> 1 % 2
```

## Számok kezelése – 4

---

### Kényszerítők és többszörösen terhelt konstansok

- számok konverziójára több többszörösen terhelt *kényszerítő* (coercion) függvény szolgál

```

fromInteger :: (Num a) => Integer -> a
fromRational :: (Fractional a) => Rational -> a
toInteger :: (Integral a) => a -> Integer
toRational :: (RealFrac a) => a -> Rational
fromIntegral :: (Integral a, Num b) => a -> b
fromRealFrac :: (RealFrac a, Fractional b) => a -> b

```

ahol

```

fromIntegral = fromInteger . toInteger
fromRealFrac = fromRational . toRational

```

- a Haskell kettőt közülük implicit konverzióra használ a számkonstansok polimorffá tételéhez
  - Egy egész szám (tizedespont nélkül) ekvivalens a `fromInteger` implicit alkalmazásával
  - Ezért pl. a 3 típusa `(Num a) => a` (v.ö. `fromInteger` eredményével)
  - Egy decimális szám (tizedesponttal) ekvivalens a `fromRational` implicit alkalmazásával
  - Ezért pl. a 3.0 típusa `(Fractional a) => a` (v.ö. `fromRational` eredményével)

## Számok kezelése – 5

---

### Kényszerítők és többszörösen terhelt konstansok (folyt.)

- típusmegkötéssel megadhatjuk egy polimorf számkonstans típusát

- `3 :: Int ==> 3`
- `3 :: Integer ==> 3`
- `3 :: Double ==> 3.0`
- `3 :: Float ==> 3.0`
- `3 :: Rational ==> 3 % 1`
- `3.0 :: Double ==> 3.0`
- `3.0 :: Float ==> 3.0`
- `3.0 :: Rational ==> 3 % 1`

- polimorf típus csak kontextussal adható meg számkonstanshoz

- `3 :: Num a => a`
- `3.0 :: Fractional a => a`
- `3 % 5 :: Integral a => Ratio a`

## Számok kezelése – 6

---

### Konverziós függvények

- decimális szám konverziós függvényekkel alakítható át egészé

- `round`     `3.5`  $\implies$  `4`

- `truncate` `3.5`  $\implies$  `3`

- `floor`     `3.5`  $\implies$  `3`

- `ceiling`   `3.5`  $\implies$  `4`

## Peano-számok megvalósítása – 1

---

### Az adattípus deklarációja

```
data Peano = Zero | Succ Peano
 deriving (Eq, Ord, Show)
```

### A Num osztályba tartozás

```
instance Num Peano where
 Zero + m = m
 Succ n + m = n + Succ m

 n - Zero = n
 Succ n - Succ m = n - m
 Zero - m = error "Peano.(-): negative number"

 abs = id
 signum Zero = 0 -- a Haskell válasza mégis Zero lesz!
 signum n = 1 -- erre meg Succ Zero! Magyarázd meg!

 fromInteger 0 = Zero
 fromInteger n | n > 0 = Succ (fromInteger (n-1))
```

## Peano-számok megvalósítása – 2

---

### Az Integral osztályba tartozás előkészítése

```
instance Real Peano where
 toRational = toRational . toInteger
```

A `toInteger` függvényt az `Integral` osztály specifikálja. A Haskell lusta kiértékelése miatt használhatjuk fel előre `toInteger Peano`-példányát, amelyet majd később definiálunk.

```
toInteger :: Integral a => a -> Integer
toRational :: Real a => a -> Rational
```

A fenti definícióban `toInteger` egy, az `Integral` osztályba tartozó (`Int`, `Integer` vagy `Peano` !) típusú értékből egy `Integer` típusú értéket állít elő.

A fenti definíció jobb oldalán `toRational` ebből az értékből, amelynek típusa egyúttal a `Real` osztályba is beletartozik, egy `Rational`  $\equiv$  `Ratio Integer` típusú értéket állít elő, azaz olyat, amelynek a számlálója és a nevezője is `Integer` típusú.

## Peano-számok megvalósítása – 3

---

Az `Integral` osztályba tartozás előkészítése (folyt.)

```
instance Enum Peano where
 succ n = Succ n

 pred Zero = error "Peano.pred: negative number"
 pred (Succ n) = n

 toEnum = fromInteger . toInteger
 fromEnum = fromInteger . toInteger
```

A két definíció azonos, de ez csak a látszat! Azt, hogy `fromInteger` és `toInteger` melyik példányát kell itt alkalmazni, `toEnum` és `fromEnum` `Enum` osztálybeli specifikációjából vezethető le:

```
toEnum :: Int -> a
fromEnum :: a -> Int

toInteger :: Integral a => a -> Integer
fromInteger :: Num a => Integer -> a
```

Az `Enum` osztály többi függvényének van alapértelmezett megvalósítása. Ha a hatékonyság cél lenne, külön meg kellene őket valósítani.

## Peano-számok megvalósítása – 4

---

### Az Integral osztályba tartozás

```
instance Integral Peano where
```

```
 n `quotRem` Zero = error "Peano.quotRem: division by zero"
```

```
 n `quotRem` d = qR n d Zero n
```

```
 where qR Zero Zero q r = (Succ q, Zero)
```

```
 qR Zero d q r = (q, r)
```

```
 qR n Zero q r = qR n d (Succ q) n
```

```
 qR (Succ n) (Succ d) q r = qR n d q r
```

```
 toInteger Zero = 0
```

```
 toInteger (Succ n) = 1 + toInteger n
```

A többi metódus vissza van vezetve a quotRem függvényre.



# Többargumentumú típusosztályok – 1

---

## Haskell 98

- többargumentumú függvények
- többargumentumú adatkonstruktorok
- többargumentumú típuskonstruktorok
- egyargumentumú típusosztályok

A típusosztályoknak is lehessen több *típusargumentuma*!

Ez *nem* része a Haskell 98 szabványnak, de több interpreterben (Hugs, GHC) megtalálható kiegészítésként.

## Definiálás, alkalmazás

- Tfh. szeretnénk egy *gyűjtő* osztályt:

```
class Collects e ce where ...
```

- felhasználási lehetőségek:

```
instance Eq e => Collects e [e] where ...
```

```
instance Eq e => Collects e (e -> Bool) where ...
```

```
instance Collects Char BitSet where ...
```

## Többszörös argumentumú típusosztályok – 2

---

### Többértelműségi probléma

- `class Collects e ce where`  
`empty :: ce`  
`insert :: e -> ce -> ce`  
`member :: e -> ce -> bool`

- problémák:

- a típusellenőrzés túl szigorú:

```
empty :: Collects e ce => ce
```

Az `e` típusváltozó nem határozható meg!

- a típus nem eléggé szigorú:

```
f x y = insert x . insert y
```

```
f True 'a' :: (Collects Bool c, Collects Char c) => c -> c
```

Csak futási idejű hibát okoz!

## Többszörös argumentumú típusosztályok – 3

---

### Típuskonstruktorok osztálya

- `class Collects e c where`  
`empty :: c e`  
`insert :: e -> c e -> c e`  
`member :: e -> c e -> bool`

- megoldott problémák:

- `empty :: Collects e c => c e` nem többértelmű

- `f :: (Collects e c) => e -> e -> c e -> c e`  
nem engedi meg az `f True 'a'` jellegű felhasználást

- `instance Collects e [] where ...`

- rossz hír: a másik két felhasználási ötlet nem működik,  
`e -> Bool` és a `BitSet` nem típuskonstruktorok

## Többszörös típusosztályok – 4

### Explicit típusfüggőség

- az osztály egyes típusparaméterei egyértelműen meghatároznak másokat
- *függőségek* megadásával írható le
- általános alak:  $x_1 \ x_2 \ \dots \ x_n \ \rightarrow \ y_1 \ y_2 \ \dots \ y_m, \ n > 0, m \geq 0$
- egy osztályhoz több függőség is megadható
- `class Collects e ce | ce -> e where ...`
- redundáns, nem megengedett függőségek:
  - $a \rightarrow a$
  - $a \rightarrow a \ a$
  - $a \rightarrow$
  - $a \rightarrow b, b \rightarrow c, a \rightarrow c$
- korlátozza a példányosítást
- megoldja a felmerült problémákat

## Többszörös argumentumú típusosztályok – 5

---

### Típusnyelvi programozás

- írhatunk programot a típusellenőrzőre
- „adataink” típusok, nem értékek
- Prologszerű számítási modell
- példa: számábrázolás és műveletek

```
data Zero
data Succ n
type One = Succ Zero; type Two = Succ One

zero = undefined :: Zero; one = undefined :: One

class Add a b c | a b -> c where
 add :: a -> b -> c
instance Add Zero b b
instance Add a b c => Add (Succ a) b (Succ c)

add one one :: Succ (Succ Zero)
```

# A HASKELL MODULNYELVE



# A Haskell modulnyelve – 1

---

- egy Haskell program modulokból épül fel
- kettős cél:
  - névtér felosztása
  - absztrakt adattípusok létrehozása
- a modul törzse deklarációkból áll
- a modulnév alfanumerikus és nagybetűvel kezdődik
- a modulok és az állományok között nincs szigorú kapcsolat (egy modul több fájlban, egy fájlban több modul megengedett)
- általános alak: `module Modulnév (exportlista) where deklarációk`
- az exportlista elhagyható, ilyenkor minden kilátszik
- ```
module Tree ( Tree(Leaf,Node), isLeaf ) where
data Tree a = Leaf | Node a (Tree a) (Tree a)
              deriving (Eq, Ord, Show)
isLeaf      :: Tree a -> Bool
isLeaf Leaf = True
isLeaf _    = False
```

A Haskell modulnyelve – 2

- `Tree (Leaf , Node)` helyett írható `Tree (. .)`
- megengedett az adatkonstruktorok csak egy részét exportálni
- szabad *tovább* exportálni importált neveket
- importálás:
 - `import Modulnév (importlista)`
 - csak a modul legelején állhat
 - az importlista elhagyható, ilyenkor minden exportált nevet importál
- minősített nevek: `Modulnév.név`
- importálás „megnyitás” nélkül: `import qualified Modulnév`
- explicit elrejtés: `import Tree hiding isLeaf`
- átnevezés: `import Tree as T`
- Prelude implicite importált, de explicit importálással felülbíráható:


```
import qualified Prelude as P hiding length
```
- típusosztályok *példányai* automatikusan exportálódnak és importálódnak

„IMPERATÍV” ELEMENK A HASKELLBEN



Monádok – 0

(Avagy: a monádok nem monádok)

- bölcsőjük a kategóriaelmélet és a 1960-as évek
- monád \leftarrow monoid vagy *félcsoport* (zárt, asszociatív, egységelemes, de nincs inverz)
- a funkcionális programozásban alkalmas eszköz *mellékhatások* kezelésére:
 - állapotok
 - kivételkezelés
 - ki- és bevitel
 - nemdeterminizmus
- a Haskellben a monád: típuskonstruktor
- a minimálisan elvárt műveleteket a `Monad` osztály adja
- ismertetések:
 - What the hell are Monads? (Noel Winstanley)
 - Monads for the Working Haskell Programmer (Theodore Norvell)
 - All About Monads (Jeff Newbern)

Meghiúsulás kezelése – 1

Bizonyos számításoknál nem mindig adható értelmes eredmény (v.ö. füzér számmá alakítása). Ilyenkor az eredményt becsomagoljuk egy feltételes típusba (`Maybe a`).

Tfh. van egy adatbáziskezelő könyvtárunk egy lekérdezőfüggvénnyel:

```
doQuery :: Query -> DB -> Maybe Record
```

Több lekérdezésből álló szekvencia:

```
r :: Maybe Record
```

```
r = case doQuery q1 db of
```

```
    Nothing -> Nothing
```

```
    Just r1 -> case doQuery (q2 r1) db of
```

```
        Nothing -> Nothing
```

```
        Just r2 -> case doQuery (q3 r2) db of
```

```
            Nothing -> Nothing
```

```
            Just r3 -> ...
```

Hátrányok:

- sokszor kell leírni ugyanazt
- nem jól olvasható

Meghiúsulás kezelése – 2

Ötlet: vezessünk be egy *kombinátort*, amely elrejtí ezt a mintázatot!

```
thenMB      :: Maybe a -> (a -> Maybe b) -> Maybe b
mB `thenMB` f = case mB of
    Nothing -> Nothing
    Just a   -> f a
```

A lekérdezési szekvencia kombinátorral felírva:

```
r :: Maybe Record
r = doQuery q1      db `thenMB` \r1 ->
    doQuery (q2 r1) db `thenMB` \r2 ->
    doQuery (q3 r2) db `thenMB` ...
```

Előnyök:

- átláthatóbb, olvashatóbb kód
- típusa, viselkedése nem változott

Állapotkezelés – 1

Bizonyos számításoknál egy *állapotot* kell *láncszerűen* végigadogatni függvények egy sorozatának. Az ilyen függvényeket *állapottranszformátoroknak* nevezzük, a típusuk:

```
type StateT s a = s -> (a, s)
```

Tfh. az adatbázisunkat módosítani is akarjuk:

```
addRec :: Record -> DB -> (Bool, DB)
```

```
delRec :: Record -> DB -> (Bool, DB)
```

vagy ugyanez a fenti típuszinonimával leírva:

```
addRec :: Record -> StateT DB Bool
```

```
delRec :: Record -> StateT DB Bool
```

A használatuk:

```
newDB :: StateT DB Bool
```

```
newDB db = let (ok1, db1) = addRec rec1 db
              (ok2, db2) = addRec rec2 db1
              (ok3, db3) = delRec rec3 db2
            in (ok1 && ok2 && ok3, db3)
```

Számos hibalehetőség!

Állapotkezelés – 2

Ötlet: használjunk itt is kombinátort!

```
thenST      :: StateT s a -> (a -> StateT s b) -> StateT s b
st `thenST` f = \s -> let (v,s') = st s
                      in f v s'
```

Ez egy állapottranszformátort kombinál egy állapottranszformátort előállító függvénnyel. Az eredmény visszaadásához szükség van még egy kombinátorra:

```
returnST    :: a -> StateT s a
returnST a = \s -> (a,s)
```

Ez egy értéket *beemel* egy identitás-állapottranszformátorba.

Az előző adatbázismódosítás a kombinátorokkal felírva:

```
newDB :: StateT DB Bool
newDB = addRec rec1 `thenST` \ok1 ->
      addRec rec2 `thenST` \ok2 ->
      delRec rec3 `thenST` \ok3 ->
      returnST (ok1 && ok2 && ok3)
```

Elrejtettük az állapotargumentum továbbadogatását! (v.ö. Prolog DCG)

Állapotkezelés megghiúsulás kezelésével kombinálva – 1

Elképzelhető, hogy egyszerre szeretnénk állapotot továbbadogatni és megghiúsulást kezelni:

```
type MbStateT s a = s -> Maybe (a, s)
```

Ehhez a típushoz új kombinátorokra van szükség:

```
thenMST :: MbStateT s a -> (a -> MbStateT s b) -> MbStateT s b
st `thenMST` f = \s -> case st s of Nothing      -> Nothing
                                     Just (v, s') -> f v s'
```

```
returnMST :: a -> MbStateT s a
```

```
returnMST v = \s -> Just (v, s)
```

Használatuk:

```
addRec :: Record -> MbStateT DB ()
```

```
delRec :: Record -> MbStateT DB ()
```

```
newDB :: StateT DB ()
```

```
newDB = addRec rec1 `thenMST` \_ ->
```

```
    addRec rec2 `thenMST` \_ ->
```

```
    delRec rec3
```

A megghiúsulás-kezelés miatt nincs szükségünk az eredményre, ezért `()` az `MbStateT` második argumentuma.

Állapotkezelés megfiúsulás kezelésével kombinálva – 2

A `_ ->` kiírása eléggé feleslegesnek tűnik. Vezessünk be egy újabb kombinátort:

```
then_MST :: MbStateT s a -> MbStateT s b -> MbStateT s b
st1 `then_MST` st2 = st1 `thenMST` \_ -> st2
```

Ennek használatával `newDB` nagyon egyszerűvé és kifejezővé válik:

```
newDB :: StateT DB ()
newDB = addRec rec1 `then_MST`
        addRec rec2 `then_MST`
        delRec rec3
```


Monádok – 1

Jó lenne ezeket a hasonló hatású kombinátorokat ugyanazzal a szintaktikával írni.

Ötlet: típusosztály bevezetése. Előnyök:

- azonos szintaktika minden monádhoz
- írhatók generikus monadikus függvények
- bevezethetők szintaktikus édesítőszer

A Monad típusosztály

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  -- Minimal complete definition: (>>=), return
  p >> q = p >>= \_ -> q
  fail s = error s
```

Itt $>>=$ (*kötés* vagy *bind*) felel meg a `then...` kombinátornak, $>>$ a `then_...` kombinátornak. $>>=$ felhasználja, $>>$ pedig eldobja a bal oldali monadikus számítás eredményét. A típusosztály `m` paramétere: *típuskonstruktor*. `fail` és $>>$ matematikailag nem kötelező, de hasznos.

Monádok – 2

Törvények

- nem minden szemantikai megkötés adható meg típusokkal
- ezeket ún. *törvényekkel* (laws) adjuk meg
- a törvények betartása a programozó felelőssége
- az Eq osztályban: $x \neq y \equiv \text{not } (x == y)$
- a Functor osztályban: $\text{fmap id} \equiv \text{id}$
 $\text{fmap } (f \cdot g) \equiv \text{fmap } f \cdot \text{fmap } g$

- a Monad osztályban:

$\text{return } a \gg= k$	$\equiv k \ a$	return bal oldali egységelem return jobb oldali egységelem $\gg=$ asszociativitási törvénye
$m \gg= \text{return}$	$\equiv m$	
$m \gg= (\backslash x \rightarrow k \ x \gg= h)$	$\equiv (m \gg= k) \gg= h$	

Párhuzam a félcsoportokkal

- $\gg=$ a félcsoport művelete
- return a félcsoport egységeleme (v.ö. a Monad-típusosztály 1. és 2. törvényével)

A Maybe monád

A megíúsulást kezelő Maybe monád része a Prelude-nek.

Deklaráció

```
instance Monad Maybe where
  Just x  >>= k = k x
  Nothing >>= k = Nothing
  return  = Just
  fail _  = Nothing
```

Használat

```
doQuery :: Query -> DB -> Maybe Record
```

```
r :: Maybe Record
```

```
r = doQuery q1 db      >>= \r1 ->
     doQuery (q2 r1) db >>= \r2 ->
     doQuery (q3 r2) db >>= ...
```

Az ST monád (állapottranszformátorok)

Gond: monadikus típus létrehozásához típuskonstruktorra van szükség; `StateT` típuszinonima, ezért nem használható.

Ötlet: az állapottranszformátort be kell csomagolni egy adatkonstruktorba.

Deklaráció

```
newtype ST s a = ST (StateT s a)
instance Monad (ST s) where
    ST st >>= f = ST (\s -> let (v,s') = st s
                               ST st' = f v
                               in st' s')
    return a    = ST (\s -> (a,s))
```

Használat

```
addRec :: Record -> ST DB Bool
delRec :: Record -> ST DB Bool
newDB  :: ST DB Bool
newDB = addRec rec1 >>= \ok1 ->
        addRec rec2 >>= \ok2 ->
        delRec rec3 >>= \ok3 ->
        return (ok1 && ok2 && ok3)
```

A do jelölés - 1

A `>>=` és `>>` operátorok kényelmesebb használatához van egy szintaktikus édesítőszer, a `do`.
`newDB` egy újabb változata:

```
newDB :: ST DB Bool
newDB = do ok1 <- addRec rec1
          ok2 <- addRec rec2
          ok3 <- delRec rec3
          return (ok1 && ok2 && ok3)
```

Átalakítási szabályok:

- $\text{do } \textit{minta} \leftarrow \textit{kifejezés} \quad \Longrightarrow \quad \textit{kifejezés} \gg= (\textit{minta} \rightarrow \text{do } \textit{parancsok})$
 $\textit{parancsok}$
- $\text{do } \textit{kifejezés} \quad \Longrightarrow \quad \textit{kifejezés} \gg \text{do } \textit{parancsok}$
 $\textit{parancsok}$
- $\text{do } \textit{let } \textit{deklarációk} \quad \Longrightarrow \quad \textit{let } \textit{deklarációk}$
 $\textit{parancsok} \quad \text{in } \text{do } \textit{parancsok}$
- $\text{do } \textit{kifejezés} \quad \Longrightarrow \quad \textit{kifejezés}$

A `do` jelölés - 2

- A `do`-jelöléssel a monadikus számításokat pszeudó-imperatív stílusban, változókat használva írhatjuk fel.
- A `<-` „értékadó” operátorral a monadikus számítás eredményét átadhatjuk egy változónak.
- A `<-` operátor jobb oldalán monadikus típusú kifejezésnek ($m \ a$) kell állnia.
- A `do` művelet a `<-` operátor bal oldalán álló *mintát* a monádon *belüli* a értékre illeszti (ami vagy sikerül, vagy megghiúsul – lásd alább.)

A `fail` függvénynek kitüntetett szerepe van a `do`-jelölésben: a `do` a `fail` függvényt hívja meg, valahányszor a mintaillesztés megghiúsul.

- Példa:

```
f :: Int -> Maybe [Int]
f ix = do let ls = [Just [1,2,3], Nothing, Just [], Just [7..10]]
           x:xs <- ls!!ix -- a pattern match failure calls "fail"
           return xs
```

- Mivel a `Maybe` típusosztályban `fail _ = Nothing`, ezért `f 0 = [2,3]` és `f 3 = [8,9,10]`, de `f 1 = Nothing` és `f 2 = Nothing`.

Alapértelmezés szerint `fail s = error s`, ahol az `s` szöveg célszerűen a hiba helyére utal.

Imperatív stílusú programozás az ST monáddal – 1

Feladat: legnagyobb közös osztó kiszámítása. Egy imperatív pszeudonyelven:

```
while x != y do
  if x < y
  then y := y-x
  else x := x-y
return x
```

Haskellben:

- `type ImpS = (Integer, Integer)`

- lekérdező transzformátorok:

```
getX, getY :: ST ImpS Integer
getX       = ST (\(x,y) -> (x, (x,y)))
getY       = ST (\(x,y) -> (y, (x,y)))
```

- módosító transzformátorok:

```
putX, putY :: Integer -> ST ImpS ()
putX x'     = ST (\(x,y) -> ((), (x',y)))
putY y'     = ST (\(x,y) -> ((), (x,y')))
```

Imperatív stílusú programozás az ST monáddal – 2

A transzformátorok használata:

```
gcdST :: ST ImpS Integer
gcdST = do x <- getX
          y <- getY
          case compare x y of
            EQ -> return x
            LT -> do putY (y-x)
                    gcdST
            GT -> do putX (x-y)
                    gcdST
```

Egy transzformátor alkalmazása egy állapotra:

```
applyST :: ST s a -> StateT s a
applyST (ST st) = st
```

Felhasználás:

```
gcd x y = fst $ applyST gcdST (x,y)
```

```
gcd 8 4 == 4 ; gcd 8 5 == 1 ; gcd 8 6 == 2
```


Monádok – 3

További tulajdonságok

- Absztrakt adatstruktúra definiálásával elérhetjük, hogy csak a `Monad` osztály kombinátorairaival lehessen kezelni egy monád elemeit.
- A monádból a `Monad` típusosztályban definiált kombinátorokkal és függvényekkel nem lehet kilépni: ha egy függvényben, amely csak ilyen kombinátorokat és függvényeket alkalmaz, megjelenik a monád, akkor a függvény eredménye mindenképpen monadikus lesz.
- Azonban nincs akadálya annak, hogy a programozó olyan függvényeket hozzon létre a `Monad` típusosztály valamely példányában, amellyel értékek „hozhatók ki” a monádból.
- Például a `Maybe` monádból a `Just` x mintára való illesztéssel vagy a `fromJust` függvénnyel hozható ki érték.
- Az `IO` monád (lásd később) ún. egyirányú (one-way) monád: nincs mód arra, hogy az `IO` monádból értéket hozzunk ki. Másszóval az `IO` monád függvényeinek csak olyan eredménye lehet, amelynek típusában szerepel az `IO` típuskonstruktor.
- A kombinátorok egyértelműen megadják a kiértékelés *sorrendjét*.

Monádok – 4

További monadikus műveletek

- sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (c:cs) = do x <- c
 xs <- sequence cs
 return (x:xs)

```
fst ((applyST . sequence) [getY,getX,gcdST] (8,6)) == [6,8,2]
```

- sequence_ :: Monad m => [m a] -> m ()
sequence_ [] = return ()
sequence_ (c:cs) = do _ <- c ; _ <- sequence cs ; return ()
- mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f = sequence . map f
- mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f

sequence_-nek és mapM_-nek nincs eredménye: akkor használjuk őket _ nélküli változatuk helyett, ha csak a mellékhatásukra van szükségünk.

Monádok aritmetikája – 1

A félcsoport kibővítése

- a félcsoport kiegészíthető egy nullelemmel (`mzero`) és egy második művelettel (`mplus`)
- törvények:

$$\begin{aligned} mzero >>= k & \equiv mzero \\ p \text{ `mplus` } mzero & \equiv p \\ mzero \text{ `mplus` } p & \equiv p \\ p \text{ `mplus` } (q \text{ `mplus` } r) & \equiv (p \text{ `mplus` } q) \text{ `mplus` } r \end{aligned}$$
- könnyű megjegyezni e törvényeket, ha gondolatban `mzero`-t 0-val, `mplus`-t az aritmetikai összeadással, `>>=`-t pedig az aritmetikai szorzással helyettesítjük
- `mzero` a `>>=` művelet bal és jobb oldali zéruseleme
- `mplus` két független számítás monadikus eredményét kombinálja egyetlen monadikus értéké
- a meghiúsulást kezelő monádok esetében (pl. `Maybe`) az `mzero` elem meghiúsulás jelzésére, az `mplus` kombinátor meghiúsulás kezelésére szolgál (v.ö. `try `mplus` catch`)

Monádok aritmetikája – 2

A MonadPlus osztály

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

```
instance MonadPlus Maybe where
  mzero          = Nothing
  Nothing `mplus` ys = ys
  xs      `mplus` ys = xs
```

- A Maybe monádban definiált mplus két érték közül a másodikat adja vissza, ha az első Nothing, egyébként pedig az elsőt.

A lista mint monád

```

• instance Monad [] where
    (x:xs) >>= f = f x ++ (xs >>= f)
    []         >>= f = []
    return x   = [x]
    fail _     = []

```

Fontos, hogy továbbolvasás előtt megértsük a [] monádban definiált >>= kombinátor működését!

```

• instance MonadPlus [] where
    mzero = []
    mplus = (++)

```

• a listanézet tkp. egy édesítőszer a monadikus kombinátoroknak!

```

[ (x,y) |
  x <- [1,2,3],
  y <- [1,2,3],
  x /= y ]

```

```

do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)

```

Ha a mintaillesztés nem sikerül a True <- ... sorban, akkor a sor egy fail _ = [] hívással lesz ekvivalens, vagyis *üres lista* lesz az eredménye!

A Monad könyvtár

- a MonadPlus osztály és két implementációja (Maybe, listák)

- további hasznos függvények:

- `msum` $::$ MonadPlus `m` \Rightarrow `[m a]` \rightarrow `m a`
`msum` $=$ `foldr mplus mzero`

- `when` $::$ Monad `m` \Rightarrow `Bool` \rightarrow `m ()` \rightarrow `m ()`
`when p s` $=$ `if p then s else return ()`

- `guard` $::$ MonadPlus `m` \Rightarrow `Bool` \rightarrow `m ()`
`guard p` $=$ `if p then return () else mzero`

- `liftM` $::$ Monad `m` \Rightarrow `(a -> b)` \rightarrow `(m a -> m b)`
`liftM f` $=$ `\a -> do { a' <- a; return (f a') }`

- `liftM2` $::$ Monad `m` \Rightarrow `(a -> b -> c)` \rightarrow `(m a -> m b -> m c)`
`liftM2 f` $=$ `\a b -> do { a' <- a; b' <- b; return (f a' b') }`

- `ap` $::$ Monad `m` \Rightarrow `m (a -> b)` \rightarrow `m a -> m b`
`ap` $=$ `liftM2 ($)`

Ki- és bevitel – 1

Alapok

- tisztán funkcionális világ \Rightarrow ugyanaz a kifejezés mindig ugyanazt az értéket adja
- ha I/O-t szeretnénk, kell egy argumentum, amely a világ állapotát képviseli: `World`
- az I/O függvények a `World` állapot transzformátorai
- monád használatával el lehet rejteni az állapot továbbadogatását

```
data IO a = IO (StateT World a)
```

- az `IO a` típus *absztrakt*, nem lehet kibontani \Rightarrow csak monadikusan lehet kezelni
- az `IO a` típusú értékeket *akciónak* nevezzük

Akciók kezelése

- ha az interpreternek akciót kell kiértékelnie, *végrehajtja*, azaz átadja neki a világ állapotát
- önálló program írásához definiálni kell a `Main.main :: IO a` (általában `IO ()` típusú) függvényt
- az `IO` könyvtár tartalmazza a fájlkezeléshez szükséges függvényeket

Ki- és bevitel – 2

Egyszerű I/O függvények

- beolvasás:

- `getChar` `:: IO Char`
- `getContents` `:: IO String`
- `getLine` `:: IO String`
- `getLine` `= do c <- getChar`
 `if c=='\n' then return ""`
 `else do cs <- getLine; return (c:cs)`

- kiírás:

- `putChar` `:: Char -> IO ()`
- `putStr` `:: String -> IO ()`
- `putStrLn` `:: String -> IO ()`
 `putStrLn s = putStr s >> putChar '\n'`

- kommunikáció:

```
interact :: (String -> String) -> IO ()
interact f = getContents >>= (putStr . f)
```


Ki- és bevétel – 3

Egy teljes példa: a wc Unix program

```
import System (getArgs)
main :: IO ()
main = do
  args <- getArgs
  case args of
    [fname] -> do fstr <- readFile fname
                  let nWords = length . words $ fstr
                      nLines = length . lines $ fstr
                      nChars = length fstr
                  putStrLn . unwords $ [ show nLines
                                         , show nWords
                                         , show nChars
                                         , fname ]
    _        -> putStrLn "usage: wc fname"
```

Ki- és bevétel – 4

Hibakezelés

- az IO monádba hibakezelés is be van építve (ld. `MBStateT`)
- a hibák `IOError` típusúak
- hiba jelzése: `ioError :: IOError -> IO a`
- felhasználói hiba: `userError :: String -> IOError`
- a kettő együtt: `fail = ioError . userError`
- hibakezelés: `catch :: IO a -> (IOError -> IO a) -> IO a`

```
getChar' :: IO Char
getChar' = getChar `catch` (\e -> return '\n')
```

szemben ugyanez:

```
getChar' :: IO Char
getChar' = getChar `catch` eofHandler
  where eofHandler e = if isEOFError e
                        then return '\n'
                        else ioError e
```

„A HASKELL PROGRAMOZÓ EVOLÚCIÓJA”

Egyszerű megoldások

Az elsőéves:

```
fac n = if n == 0 then 1
        else n * fac (n-1)
```

A kezdő:

```
fac 0 = 1
fac n = n * fac (n-1)
```

A haladó (jobb, ill. baloldali érzületű, és aki mást mutat, mint ami):

```
fac n = foldr (*) 1 [1..n]
fac n = foldl (*) 1 [1..n]
fac n = foldr (\x g n -> g (x*n)) id [1..n] 1
```

A „memoizer”:

```
facs = scanl (*) 1 [1..]
```

```
fac n = facs !! n
```

Valamivel komplikáltabb megoldások

Az akkumuláló:

```
facAcc a 0 = a
```

```
facAcc a n = facAcc (n*a) (n-1)
```

```
fac = facAcc 1
```

A fixpontos:

```
y f = f (y f)
```

```
fac = y (\f n -> if (n==0) then 1 else n * f (n-1))
```

A kombinátoros:

```
s f g x      = f x (g x)
```

```
k x y        = x
```

```
b f g x      = f (g x)
```

```
c f g x      = f x g
```

```
y f          = f (y f)
```

```
cond p f g x = if p x then f x else g x
```

```
fac = y (b (cond ((==) 0) (k 1)) (b (s (*)) (c b pred)))
```

A „statikus”

```
data Zero
```

```
data Succ n
```

```
class Add a b c | a b -> c where
```

```
  add :: a -> b -> c
```

```
instance Add Zero b b
```

```
instance Add a b c => Add (Succ a) b (Succ c)
```

```
class Mul a b c | a b -> c where
```

```
  mul :: a -> b -> c
```

```
instance Mul Zero b Zero
```

```
instance (Mul a b c, Add b c d) => Mul (Succ a) b d
```

```
class Fac a b | a -> b where
```

```
  fac :: a -> b
```

```
instance Fac Zero One
```

```
instance (Fac n k, Mul (Succ n) k m) => Fac (Succ n) m
```

A Ph.D. fokozatot szerzett – 1

```
-- explicit type recursion based on functors
newtype Mu f = Mu (f (Mu f)) deriving Show
in      x    = Mu x
out (Mu x) = x

-- cata- and ana-morphisms for *arbitrary* (regular) base functors
cata phi = phi . fmap (cata phi) . out
ana  psi = in  . fmap (ana  psi) . psi

-- base functor and data type for natural numbers,
-- using a curried elimination operator
data N b = Zero | Succ b deriving Show
instance Functor N where
    fmap f = nelim Zero (Succ . f)
nelim z s Zero      = z
nelim z s (Succ n) = s n

type Nat = Mu N
```

A Ph.D. fokozatot szerzett – 2

```
-- conversion to internal numbers, conveniences and applications
int = cata (nelim 0 (1+))
instance Show Nat where
  show = show . int

zero    = in    Zero
suck    = in . Succ  -- pardon my "French" (Prelude conflict)
plus n  = cata (nelim n      suck    )
mult n  = cata (nelim zero (plus n))

-- base functor and data type for lists
data L a b = Nil | Cons a b deriving Show
instance Functor (L a) where
  fmap f = lelim Nil (\a b -> Cons a (f b))
lelim n c Nil          = n
lelim n c (Cons a b) = c a b

type List a = Mu (L a)
```


A Ph.D. fokozatot szerzett – 3

```
-- conversion to internal lists, conveniences and applications
list = cata (lelim [] (:))

instance Show a => Show (List a) where
  show = show . list

prod = cata (lelim (suck zero) mult)

upto = ana (nelim Nil (diag (Cons . suck)) . out)

diag f x = f x x

fac = prod . upto
```

A professzor

```
fac n = product [1..n]
```