Start

Ez a jegyzet a tudásom és a http://pnyf.inf.elte.hu/fp/Index.xml szerint készült, előfordulhatnak benne hibák, de igyekszem megértetni.

Elsőnek érdemes elolvasni, kis leírás a Haskell programnyelvről:
http://pnyf.inf.elte.hu/fp/Intro.xml
http://pnyf.inf.elte.hu/fp/Middle4.xml

Továbbiakban használom még ezeket:
Haskell honlapja, ahonnan a GHCi is letölthető: http://www.haskell.org/
Haskell leírások (data, types stb..): http://www.haskell.org/ghc/docs/latest/html/libraries/
Haskell leírások (functions, operators stb..): http://zvon.org/other/haskell/Outputglobal/index.html

Ez pedig még a zh-n is használható: http://www.haskell.org/hoogle/

A megértéshez fontos használni a honlapokat is, mert anélkül hiányosságok lehetnek (például a zárójelezésről kezdetben még nem írok).

Az elejétől fogva érdemes minden feladatot kipróbálni, kielemezni és értelmezni, hiszen későbbiekben ezekre épül a többi. A feladatok megoldásai is szerepelnek itt (persze nem minden), de csak útmutatónak, nem megoldásnak..

http://tryhaskell.org/
http://nyelvek.inf.elte.hu/leirasok/Haskell/
http://hackage.haskell.org/package/base-4.7.0.0/docs/src/

Számok

http://pnyf.inf.elte.hu/fp/Syntax.xml

Osztás

Kétféle osztás van. Az egyik Fractional a-val (*/*) tér vissza, a másik pedig Integral a-val (*div*).

(/)

Fractional a jelentése: a kicserélhető a Rational, Double, … típusokra.

Példa

1 / 3 0.3333333333333333 :: Double

(div)

Integral a jelentése: a kicserélhető az Int, Integer, … típusokra.

Példa

53 `div` 5 akár így is felírható: „div 53 5”

10 :: Integer vagyis *53-ban az 5 10-szer van meg.*

Maradék

53 `mod` 5 akár így is felírható: „mod 53 5”

3 :: Integer vagyis 53 osztva 5-tel 3 maradékot ad.

Feladatok

Körülbelül hány másodperc van egy évben?

365 * 24 * 60 * 60 31536000 :: Integer

1.01 sugarú gömb térfogata?

4 * 1.01^3 * pi / 3 4.315714736781623 :: Double

23 osztja-e a 532253373-at?

mod 532253373 23 == 0 True :: Bool

Konverziók

http://pnyf.inf.elte.hu/fp/Conversion.xml

Leírás

Általános esetben két különböző típussal nem lehet dolgozni. Egy Double-t még összehasonlítani sem lehet egy Float-tal. Erre vannak az ún. konverziók, amelyekkel az effajta típuskülönbségek kiküszöbölhetőek. Ezek csak azon típusokra használhatóak, amelyek az értelmezési tartományaikban szerepelnek, de az értékkészletben lévő típusúként használhatóak.

Unikód

Részletek a Listánál (String-nél)

fromIntegral

- Értelmezési tartomány: Int vagy Integer
- Értékkészlet: Int, Integer, Rational, Float, Double

realToFrac

- Értelmezési tartomány: Int, Integer, Rational, Float vagy Double
- Értékkészlet: Rational, Float, Double

Használat

(5 :: Int) == (5 :: Integer) => Nem lehet összehasonlítani, mert a típusuk különbözik (egyik Int, a másik Integer)
De fromIntegral (5 :: Int) == fromIntegral (5 :: Integer) == True :: Bool
Így már lehet műveleteket végezni két különböző számmal. Ez szintén igaz a *realToFrac*-ra is, amely több típusra használható.

Kerekítések

truncate

```
truncate :: (Integral b, RealFrac a) => a -> b
```

Nulla felé eső egész számra kerekít.

```
truncate 1.001 == truncate 1.999 == 1 :: Integer
truncate (-1.999) == truncate (-1.001) == -1 :: Integer
```

round

```
round :: (Integral b, RealFrac a) => a -> b
```

Legközelebbi egész számhoz kerekít.

```
round 0.001 == round 0.500 == 0 :: Integer
round 0.501 == round 1.499 == 1 :: Integer
round (-0.501) == round (-1.499) == -1 :: Integer
```

Megj.: 0.5 –> 0; *1.5 –> 2;* 2.5 –> 2; *3.5 –> 4;* 4.5 –> 4; *5.5 –> 6;* 6.5 –> 6; *7.5 –> 8;* 8.5 –> 8; *9.5 –> 10*

ceiling

```
ceiling :: (Integral b, RealFrac a) => a -> b
```

Felfelé kerekít.

```
floor 1.001 == floor 1.999 == 2 :: Integer
floor (-1.001) == floor (-1.999) == (-1) :: Integer
```

floor

```
floor :: (Integral b, RealFrac a) => a -> b
```

Lefelé kerekít.

```
floor 1 == floor 1.999 == 1 :: Integer
floor (-1.001) == floor (-1.999) == (-2) :: Integer
```

Magyarázat

Mint látható, a *round* x.5 esetén mindig a páros számhoz kerekít. Ennek az összegzésnél van szerepe:
Vegyük a [0.5, 1.5 .. 100.5] listát, amelynek az összege: sum [0.5, 1.5 .. 100.5] == 5100.5 :: Double
Ha felfelé kerekítenénk (*ceiling),* akkor ez 5151 lenne, lefelé kerekítés (*floor)* esetén pedig 5050.
A *round*-nál ez az összeg 5100.

Feladatok

10^9 gyökéhez legközelebb eső egész szám?

```
round (sqrt 10 ^ 9)                    31623 :: Integer
```

Mi az unikód kódja az 'x' karakternek?

```
fromEnum 'x'                           120 :: Int
```

Melyik az 50 unikód kódú karakter?

```
(toEnum 50 :: Char)                    '2' :: Char
```

Bool

http://pnyf.inf.elte.hu/fp/Bools.xml

Kisebb (<), nagyobb (>), kisebb egyenlő (<=), nagyobb egyenlő (>=), egyenlő (==), nem egyenlő (/=) és a negálás (*not).* Szintaktikában az *a == b* helyes, de az *a == b == c* már nem, csak zárójelezve, *(a==b)==c,* ahol c *True* vagy *False.*

Műveletek

(&&)

```
True && True == True        True && False == False
False && True== False       False && False == False
```

(||)

```
True || True == True        True || False == True
False || True== True        False || False == False
```

Feladatok

Kifejezés, amely pontosan akkor True ha a 23 nem osztja a 532253373-at!

```
div 532253373 23 /= 0    vagy    not (div 532253373 23 == 0)
```

Írjuk ki a rejtett zárójeleket!

```
6 < 4 || 4 >= 5 && 12 /= 4 * 4    =>    (((6 < 4) || (4 >= 5)) && (12 /= (4 * 4)))
```

Távolítsunk el minél több zárójelpárt!

```
((1 < 2) && (50 > (100 - 2) `mod` 50))    =>    1 < 2 && 50 > (100 - 2) `mod` 50
```

Zárójelezzük a következő kifejezést!

```
2 < div 18 4 || mod 15 5 > (-3)    =>    ((2 < (div 18 4)) || ((mod 15 5) > (-3)))
```

Listák

http://pnyf.inf.elte.hu/fp/Lists.xml

Felépítésük

A listák elemei csak ugyanolyan típusúak lehetnek, mint például [1,2,3], ahol a típus az Integer, vagy [True, True], ahol Bool, de az [1,2,True], már helytelen. Üres lista: []

A sorrend és a hossz is lényeges:

[1,2,3] /= [3,1,2] [1,1] /= [1]

Ugyanígy az egy elemű listák sem összehasonlíthatóak az elemmel:

[13] és 13 nem ugyanaz, de nem is összehasonlíthatóak!!

([13] :: [Integer], 13 :: Integer, vagyis más a típusuk)

PontPont kidejezések

PontPont kifejezéseket egyszerűség miatt szoktunk használni. Például az [1..5] megegyezik az [1,2,3,4,5] listával. Két elemet is megadhatunk kezdetnek, ekkor ez a sorozat növekedését vagy csökkenését adja meg:
[1,3..10] azt jelenti, hogy 2-vel növekszik a sorozat, így [1,3,5,7,9] lesz.
[1,(-2).. (-10)] pedig ezzel lesz egyenlő [1,-2,-5,-8]

Végtelen listákra is ezt alkalmazzuk:
[1..] 1-től meg a végtelenbe egyesével, [1,3..] pedig kettesével..

Szöveg (String)

A szöveg karakterekből épül fel, ezért a String típus nem más, mint [Char].

„abc” == [‘a’,’b’,’c’]
[‘a’..’z’] a kisbetűk listája, vagyis „abcdefghijklmnopqrstuvwxyz” :: [Char]
[‘A’..’Z’] a nagybetűk listája, vagyis ”ABCDEFGHIJKLMNOPQRSTUVWXYZ” :: [Char]
[‘0’..’9’] a számok listája, vagyis ”0123456789” :: [Char]

Karakter unikód kóddá alakítás

fromEnum ‘4’ == ord ‘4’ == 52 :: Int
digitToInt ‘4’ == 4 :: Int (ez csak 0-9 és A-F esetén)

Egy szám szerinti karakter

toEnum 52 :: Char == chr 52 == ‘4’ :: Char
intToDigit 4 == ‘4’ :: Char (ez csak 0-15 esetén)

A toEnum és a *fromEnum* Prelude-ben van benne, az *ord, chr, intToDigit* és a *digitToInt* pedig a Data.Char-ban.

Művelet listákon

Több fontosabb művelet (*filter, map, concat stb…*) később fognak szerepelni.

length

Visszaadja a lista hosszát véges lista esetén.
length [4,7,8] == 3 :: Int length „alma” == 4 :: Int length [1..1000] == 1000 :: Int

(!!)

A megadott helyen lévő elemet adja vissza. A listák 0-tól indexelődnek.
[‘a’,’b’,’c’] !! 1 == ‘b’ :: Char „abc” !! 0 == ‘a’ :: Char „a” !! 1 == Error [1..10] !! 9 == 10 :: Integer

(++)

Két listát konkatenál.
„he” ++ „llo” == „hello” [1..5] ++ [6..10] == [1..10] [1] ++ [] == [1] [] ++ [1] == [1]

sum és product

sum a lista összegét adja vissza, a *product* pedig a szorzatot. Ez csak Num a típusúakra használható.

```
sum [1..10] == 55 :: Integer                    product [1..10] == 3 628 800 :: Integer
sum [1.0, 1.1, 1.2] == 3.3 :: Double            product [1.0,1.1,1.2] == 1.32 :: Double
```

Feladatok

Soroljuk fel 10-től visszafelé -10-ig a számokat!

```
[10,9.. -10]        [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10] :: [Integer]
```

Adjuk meg a 113. elemét annak a számtani sorozatnak, amelynek az első két eleme 11 és 32!

```
[11,32..] !! 112                    2363 :: Integer
```

Hányféleképpen lehet sorba rendezni 10 különböző elemet?

```
product [1..10]                     3628800 :: Integer
```

Hányféleképpen választhatunk ki 70 különböző elemből 30 elemet?

```
div (product [41..70]) (product [1..30])        55347740058143507128 :: Integer
```

Halmazkifejezések

http://pnyf.inf.elte.hu/fp/Comprehensions.xml

Felépítésük

A példa: { n^2 | n ∈ N, n páros } Megoldás: [n^2 | n <- [1..], n `mod` 2 == 0]

Így kell definiálni egy listagenerátort: [visszatérési érték | változó_1 <- lista_1, .., változó_n <- lista_n, feltétel]

– *Visszatérési érték* lehet olyan, ahol a változónk szerepel vagy nem. Akkor adódik hozzá, ha a feltétel igaz. Itt n^2.

– a *változó <- lista* annyit jelent, hogy a változó a lista elemeinek értékét veszi fel sorrendben. Ha több változónk van, és valamelyik a másikra hivatkozik, akkor a másikat előrébb kell definiálni.
Itt n <- [1..] azt jelenti, hogy elsőnek n az 1 lesz, aztán 2, 3, 4. stb..

– *Feltételnek* olyat szabad csak megadni, ami igazzá vagy hamissá értékelődik ki. Különben hiba.
Itt n `mod` 2 == 0 azt jelenti, hogy csak a páros számokkal foglalkozik.

Rendezett párok

Rendezett párok elemei, a listával ellentétben, lehetnek különböző típusúak. Például (‘1’,True) helyes felírás, ekkor a típusa (Char,Bool) lesz a típusa.

Ezekre is lehet írni listagenerátort, például
[(a,b) | a <- "abc", b <- [1,2]] [(‘a’, 1), (‘a’, 2), (‘b’, 1), (‘b’, 2), (‘c’, 1), (‘c’, 2)] :: [(Char, Integer)]

A rendezett párok első elemét az *fst* függvénnyel kapjuk meg: fst (‘a’,True) == ‘a’ :: Char
második elemét pedig az *snd*-vel: snd (‘a’,True) == True :: Bool

zip – unzip

A *zip* „összezipel” két listát, azonos helyen lévőket mindaddig, amíg valamelyik lista nem fogy el.
zip "abcd" [1,2] [(‘a’, 1), (‘b’, 2)] :: [(Char, Integer)]
Az *unzip* pedig „kicsomagol” egy már „becsomagolt” listát egy rendezett párba
unzip [(‘a’, 1), (‘b’, 2)] ("ab",[1,2]) :: ([Char], [Integer])

take – drop

A *take* adott hosszúságú listát hagy meg a megadott lista elejétől.
take 5 [1,2]== [1,2] :: [Integer] take 2 „abcde” == „ab” :: [Char]

A *drop* pedig adott hosszúságú listát hagy el a megadott lista elejéről.
drop 5 [1,2] == [] :: [Integer] drop 2 „abcde” == „cde” :: [Char]

concat, words – unwords

A *concat* csak „listák a listában” típusúakra alkalmazható. Példák
concat [„Van”,”Egy”,”Alma”] == „VanEgyAlma” [[Char]] -> [Char]
concat [[1..5],[6..10]] == [1..10] [[Integer]] -> [Integer]

A *words* egy szöveget szavakra szed szét. A szöveg szóközöket tartalmaz. Példa
words „Van Egy Alma.” == [„Van”,”Egy”,”Alma.”] [Char] -> [[Char]]

Az *unwords* pedig olyasmi, mint a *concat*, de minden közé tesz egy szóközt.
unwords ["Van","egy","alma."] == "Van egy alma." [[Char]] -> [Char]

A *word* és az *unwords* csak Char típusúakra használható.

Feladatok

2 hatványai növekvő sorrendben 1-től 2^{10}-ig!

```
[2^n | n<-[0..10]]
```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024] :: [Integer]

Első 10 négyzetszám kétszerese!

```
[2*n^2 | n<-[0..9]]
```
[0, 2, 8, 18, 32, 50, 72, 98, 128, 162] :: [Integer]

Állítsunk elő olyan 10 hosszúságú listát, amely váltakozva tartalmazza a False és True értékeket!

```
[even n | n<-[1..10]]
```
[False, True, False, True, False, True, False, True, False, True] :: [Bool]

Melyik legkisebb 2 hatvány nagyobb, mint 10^{20}?

```
head [2^n | n<-[0..], 2^n >10^20]
```
14757395258967641928 :: Integer

Melyik legkisebb n természetes számra igaz: $1024^n > 2 * 1000^n$?

```
head [n | n<-[0..], 1024^n > 2*1000^n]
```
30 :: Integer

Soroljuk fel a 60 osztóit!

```
[n | n<-[1..60], mod 60 n == 0]
```
[1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60] :: [Integer]

Hány osztója van a 60-nak?

```
length [n | n<-[1..60], mod 60 n == 0]
```
12 :: Int

Prímszám-e az 123457?

```
length [n | n<-[2..(div 123457 2)],mod 123457 n == 0] == 0
```
True :: Bool

Állítsuk elő azt a listát, amely sorrendben tartalmazza az összes (óra, perc) párt!

```
[(h,m) | h<-[0..23], m<-[0..59]]
```

Állítsuk elő azt a listát, amely párként tartalmazza az összes dominót: [(0,0),(0,1),(1,1)...] !

A (1,0) ne szerepeljen, mert az ugyanazt a dominót reprezentálja, mint a (0,1). Megjegyzés: A dominók legkevesebb nulla, legtöbb kilenc pontot tartalmazhatnak.

```
[(x,y) | x<-[0..9], y<-[x..9]]
```

Keressünk olyan a, b, c logikai értékeket, melyekre teljesül a következő logikai feltétel:

```
(a || (b && c)) /= ((a || b) && c)
```
(True|| (True && False)) /= ((True|| True) && False)

Állítsuk elő azt a listát, amely sorrendben tartalmazza az összes (hónap, nap) párt egy 365 napos évben!

```
[(m,d) | m<-[1..12], d<-[1..31], (m `elem` [4, 6, 9, 11]) <= (d <= 30), (m == 2) <= (d <= 28)]
```

Állítsuk elõ az [(1,'a'),(2,'b'),...(...,'z')] listát!

```
zip [1..] ['a'..'z']
```

Állítsuk elõ a következõ listát: [1,2,2,3,3,3,4,4,4,4, ...] ! Az i szám i-szer szerepel a listában.

```
[ x | x <- [1..], y <- [1..x]]
```

Állítsuk elő az 1,2,1,2,3,2,1,2,3,4,3,2,1,2,3,4,5,4,3,2,1,2,3,4,5,6,5,4,3,2,1,.. sorozatot!

```
concat [[1..n] ++ [n-1,n-2..2] | n <- [2..]]
```

Állítsuk elő a következő végtelen szöveget: "* ** *** **** ***** ..." !

```
unwords [['*'| m <- [1..n]]|n <- [1..]]
```

Függvények definiálása

.hs fájlok (vagy .lhs)

Egy Haskell modul szerkezete:

- Fejléc (legfelső szintű modulnál nem kell): module modulnév where
- import deklarációk (például a Data.Char)
- egyéb deklarációk

GHCi-ben egy modul betöltése: „:l <fájl neve>.hs”, újratöltés pedig a „:r” paranccsal történik. Állítsuk be alapméretezett programnak a programot, ezután dupla kattintásra más megnyitja és be is tölti a fájlt. Ha duplán kattintunk a fájlunkra, akkor rögtön megnyitja a GHCi-t és be is tölti a fájlt.

A Prelude automatikusan betöltődik, ha pedig még import-álunk a fájlunkban, akkor azok is mellé.

Ha nem töltünk be semmilyen fájlt, a GHCi ugyanúgy használható, de ekkor külön import-álni kell azokat, amelyekre szükségünk van (Data.List, etc…). Az import deklarációk csak a modul elején lehetnek!

Egy definíciót, amelyet felül szeretnénk írni (például az even), azt így rejthetjük el:
import Prelude hiding (even) , de többet is el lehet akár (import Prelude hiding (even,odd))

Egysoros megjegyzés: `-- megjegyzés a sor végéig`
Többsoros megjegyzés, amelyek egymásba ágyazhatóak: `{- megjegyzés -}`

Deklarációk

A deklarációk sorrendje nem számít!

- típusdeklarációk
  Azt mutatják meg, hogy milyen típusú paraméterekre van szüksége a függvénynek, és ezekből milyen típusú lesz eredmény. (Például: f :: Int -> Int)
- függvénydefiníciók (Például: f x = x)
- konstansdefiníciók (Például: pi = 2 * acos 0)
- operátor definíciók (Például: a <= b = not (a > b))
- típusdefiníciók (Például: type String = [Char])
- típusosztály definíciók (Például: class Num a where)
- típusosztály példányosítás (Például: instance Num Int where)

Függvénydeklarálás

Egyparaméteres függvény típus nélkül

```
f x = x + 1
```

Használata: `f 2` Eredmény: `3`

Kétparaméteres függvény típussal

```
g :: Integer -> Integer -> Integer
g a b = a * b + 1
```

Használata: `g 2 3` Eredmény: `7`

Mint látható (típusból is), 2 Integer-re van szüksége, amiből egy Integer lesz.

Feladatok

Tesztesetek a honlapon vannak, ide csak a feladatot, típust és a deklarációt írom le.

Generáljuk a következő listát: [1, 2, …, n-1, n, n-1, …, 2, 1]!

```
mountain :: Integer -> [Integer]
mountain n = [1..n-1] ++ [n,n-1..1]
```

Háromszög szerkeszthetősége

```
areTriangleSides :: Real a => a -> a -> a -> Bool
areTriangleSides a b c = (a+b > c) && (a+c > b) && (b+c > a)
```

Prelude.even

```
even :: Integer -> Bool
even n = mod n 2 == 0
```

Megj.: *even n = not (odd n)* , ha *odd* definiálva van

Prelude.odd

```
odd :: Integer -> Bool
odd n = mod n 2 /= 0
```

Megj.: *odd n = not (even n)* , ha *even* definiálva van

Oszthatóság

```
divides :: Integer -> Integer -> Bool
divides a b = mod b a == 0
```

Szőkőév-e

```
isLeapYear :: Integer -> Bool
isLeapYear n = (mod n 400 == 0)|| (mod n 4 == 0 && mod n 100 /= 0)
```

Négyzetösszeg

```
sumSquaresTo :: Integer -> Integer
sumSquaresTo n = sum [n^2 | n <- [0..n]]
```

Osztók

```
divisors :: Integer -> [Integer]
divisors n = [x | x <- [1..n], mod n x == 0]
```

Valódi osztók

```
properDivisors :: Integer -> [Integer]
properDivisors n = [x | x <- [2..n-1],mod n x == 0]
```

Mintaillesztés

http://pnyf.inf.elte.hu/fp/Patterns.xml

Minta lehet

- változó: x, xs, y, a, …
- joker: _
- típus specifikus minták: True, 0, (a, b), …
- Üres lista minta: []
- Egyelemű lista minta: [a]
- Kételemű lista minta: [a, b]
- legalább 1 elemű lista: (x:xs)
- legalább 2 elemű lista: (x:y:xs)

Példa a jokerre

```
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

Egyszerűbben

```
True && True = True
_    &&   _  = False
```

Annyit jelent, hogy ha `True && True` -t hívunk meg, akkor `True`, egyéb esetben, bármi is az első vagy a második, `False` lesz.

(:) jelentése és példa

Szemantika:

- [] az üres listákra illeszkedik
- A h:t minta akkor illeszkedik, ha a lista nem üres: a h minta illeszkedik a lista fejére, a t minta pedig illeszkedik a lista törzsére.

A kettőspont típusa: (:) :: a -> [a] -> [a]
Tehát ha h:t típusa [x], akkor h típusa x, t típusa pedig [x]. A kettőspont jobbra köt!

Példa

[1,2,3] == 1:2:3:[] => 1:2:[3] => 1:[2,3] => [1,2,3]
(1:(2:(3:[]))) == [1,2,3]
'H' : „ello" == „Hello" :: [Char]
0 : [1,2,3] == [0,1,2,3] :: [Integer]

Hivatkozás listára

Például *"tails (x:xs) = (x:xs) : tails xs"* esetén célszerűbb *"tails l@(x:xs) = l : tails xs"*–t írni, tehát listára így hivatkozunk: tails l@(x:xs) = l : tails xs

head – last

Nincs mit magyarázni, a *head* a lista első elemét adja vissza
a *last* pedig az utolsót

head [1..5] == 1 :: Integer
last [1..5] == 5 :: Integer

Mindkettő nemüres listákon működik csak.

Feladatok

Elemcsere

```
swap :: (a, b) -> (b, a)
swap (a,b) = (b,a)
```

Tükrözés az x tengelyre

```
mirrorX :: Num a => (a, a) -> (a, a)
mirrorX (a,b) = (a,-b)
```

Origó középpontú nagyítás

```
scale' :: Num a => a -> (a, a) -> (a, a)
scale' n (a,b) = (n*a,n*b)
```

Pontra tükrözés

```
mirrorP :: Num a => (a, a) -> (a, a) -> (a, a)
mirrorP (a,b) (x,y) = (2*a-x, 2*b-y)
```

Két pont távolsága

```
distance :: Floating t => (t, t) -> (t, t) -> t
distance (a,b) (x,y) = sqrt((a-x)^2+(b-y)^2)
```

Modulo 3 szorzás

```
mul3 :: Int -> Int -> Int
n `mul3` m = n*m `mod` 3
```

Sortörés-szóköz csere

```
replaceNewline :: Char -> Char
replaceNewline '\n' = ' '
replaceNewline x = x
```

Sortörés-szóköz cserék

```
replaceNewlines :: String -> String
replaceNewlines [] = []
replaceNewlines (x:xs) = replaceNewline x : replaceNewlines xs
```

Vagy `replaceNewlines l = [ replaceNewline x| x <- l]`

Megj.: csak akkor működik, ha a *replaceNewLine* függvény már definiálva van.

„a" – „az" csere

```
swap_a_az :: String -> String
swap_a_az "a" = "az"
swap_a_az "az" = "a"
swap_a_az str = str
```

„a" – „az" cserék

```
swapAll_a_az :: String -> String
swapAll_a_az str = unwords [swap_a_az x| x <- words str]
```

Megj.: csak akkor működik, ha a *swap_a_az* függvény már definiálva van.

1 elemű-e a lista?

```
isSingleton :: [a] -> Bool
isSingleton [x] = True
isSingleton _ = False
```

Kezdőbetű nagybetűvé

```
toUpperFirst :: String -> String
toUpperFirst (x:xs) = toUpper x:xs
```

Megj.: a *toUpper* függvény a Data.Char-ban szerepel.

Összes kezdőbetű nagybetűvé

```
toUpperFirsts :: String -> String
toUpperFirsts l = unwords [toUpperFirst s| s <- words l]
```

Megj.: csak akkor működik, ha a *toUpperFirst* függvény már definiálva van.

Példa:

1) *toUpperFirsts "az az alma" == unwords [toUpperFirst s| s <- words "az az alma"]*
2) *words "az az alma" => ["az","az","alma"]*

3) *concat [toUpperFirst s| s <- [”az”,”az”,”alma”]] =>*
   concat [toUpperFirst ”az”, toUpperFirst ”az”, toUpperFirst ”alma”]
4) *concat [toUpperFirst ”az”, toUpperFirst ”az”, toUpperFirst ”alma”] =*
   concat [”Az”,”Az”,”Alma”]
5) *unwords [”Az”,”Az”,”Alma”] => ”Az Az Alma”*

Azonos szavak egy szövegben

```
countOfAs :: String -> Int
countOfAs l = length [x| x <- words l, x == "a"]
```

Szűrés elemek közti távolság alapján

```
distantPairs :: [(Integer,Integer)] -> Int
distantPairs l = length [x | x <- l, (snd x - fst x) >= 2]
```

A lista minden 5. eleme

```
everyFifth :: [a] -> [a]
everyFifth [] = []
everyFifth l = head l : everyFifth (drop 5 l)
```

Rekurzió

http://pnyf.inf.elte.hu/fp/Recursion.xml

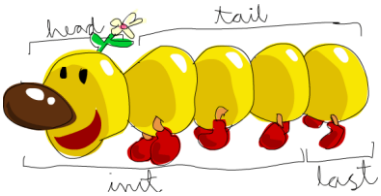
Leírás

Nézzük az egyik egyszerű függvényt, az összegzést (*sum*). Egy rekurzív definíciója:

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

sum [1,2,3,4,5] => 1+sum [2,3,4,5] => 1+2+sum [3,4,5] => 1+2+3+sum [4,5]
=> ... => 1+2+3+4+5+sum [] => 1+2+3+4+5+0 => 15 :: Integer

Vagy példának jó az *init* is, ami az utolsó elem kivételével mindent visszaad:




```
init :: [a]{-nemüres-} -> [a]
init [x] = []
init (x:xs) = x : init xs
```

init [1,2,3] => 1 : init [2,3] => 1 : 2 : init [3] => 1 : 2 : [] => [1,2] :: [Integer]
(Eddigi ismeretek alapján ugye 1:2:[] => 1:[2] => [1,2])

A *minimum* pedig így működik:

```
minimum :: Ord a => [a]{-véges, nemüres-} -> a
minimum [x] = x
minimum (x:xs) = min x (minimum xs)
```

minimum [3,2,1,4] => min 3 (minimum [2,1,4]) => min 3 (min 2 (minimum [1,4])) =>
min 3 (min 2 (min 1 (minimum [4]))) => min 3 (min 2 (min 1 (4))) =>
min 3 (min 2 (1)) => min 3 (1) => 1 :: Integer

Feladatok

Prelude.last

```
last :: [a]{-nemüres-} -> a
last [x] = x
last (x:xs) = last xs
```

Prelude.concat

```
concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ (concat xs)
```

Prelude.++

```
(++) :: [a] -> [a] -> [a]
[] ++ l = l
(x:xs) ++ l = x : (xs ++ l)
```

Összefésülés

```
merge :: [a] -> [a] -> [a]
merge [] l = l                    merge [] l = l
merge l [] = l                    merge l [] = l
merge (x:xs) l = x : merge l xs   merge (x:xs) (y:ys) = x:y:merge xs ys
```

Prelude.zip

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []                zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ [] = []                zip _ _ = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Data.List.isPrefixOf

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf [] _ = True
isPrefixOf _ [] = False
isPrefixOf (x:xs) (y:ys) = x==y && isPrefixOf xs ys
```

Prelude.elem

```
elem :: Eq a => a -> [a]{-véges-} -> Bool
elem _ [] = False
elem n (x:xs) = n==x|| elem n xs
```

Data.List.nub

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x: nub [e| e <- xs, x/=e]
```

Polinom kiértékelése

```
polinom :: Num a => [a] -> a -> a
polinom [] _ = 0
polinom (x:xs) n = x + n * polinom xs n
```

Megj.: rekurzió nélkül: *polinom l n = sum [(fst x)*n^(snd x) | x <- (zip l [0..])]*
Vagy *polinom l n = sum (map (\ (x,y) -> x*n^y) (zip l [0..]))*
(a *map*-ről és a névtelen függvényekről később lesz szó)

Lista feldarabolása

```
runs :: Int -> [a] -> [[a]]
runs _ [] = []
runs n l = [take n l] ++ runs n (drop n l)
```

Feldarabolás másképp

```
slice :: [Int] -> [a] -> [[a]]
slice _ [] = [[]]    slice [] _ = []
slice [] _ = []      slice (n:ns) l = take n l : slice ns (drop n l)
slice (x:xs) l = runs x (take x l) ++ slice xs (drop x l)
```

Minden n-edik elem

```
every :: Int -> [a] -> [a]
every _ [] = []
every n l = head l : every n (drop n l)
```

Gyorsrendezés

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

Data.List.tails

```
tails :: [a] -> [[a]]
tails [] = [[]]
tails l = l : tails (tail l)
```

Megj.: az utolsó sor így is lehetne: *tails l@(x:xs) = l : tails xs*
rekurzió nélkül az egész: *tails l = [drop n l | n <- [0..length l]]*

Data.List.inits

```
inits :: [a] -> [[a]]
inits l = [take n l | n <- [0..length l]]
```

Esetszétválasztás

http://pnyf.inf.elte.hu/fp/Guards.xml

Felépítésük

függvénynév változók
| feltétel_1 = kifejezés_1
| feltétel_2 = kifejezés_2
...
| feltétel_n = kifejezés_n

függvénynév változók =
if feltétel_1 then kifejezés 1
else if feltétel_2 then kifejezés_2
...
else if feltétel_n then kifejezés_n

Vagy

függvénynév változók | feltétel_1 = kifejezés_1 | feltétel_2 = kifejezés_2 . | feltétel_n = kifejezés_n
függvénynév változók = if feltétel_1 then kifejezés_1 else if feltétel_2 then kifejezés_2 . else if feltétel_n then kifejezés_n

Tehát a típus, *név* és a változók után egy |-vel kezdődik, utána a feltétel, ezt követi az = és a kifejezés. Ezeket lehet egy sorba is írni, de általában külön sorba írjuk őket és bentebb kezdjük, mint a *függvénynév.*

A | felváltható az if, else if és az else szavakkal, az = pedig then szóval, a változók után pedig =-t kell tenni, ekkor if/else if feltétel then kifejezés és else kifejezés.

Megj.: Átláthatóság miatt célszerűbb a vonalas |, több sorban rendezett változatot használni.

A feltételek Bool típusúak és fentről lefelé vizsgáljuk. Amelyik elsőnek értékelődik ki True-ra, annak a kifejezése hajtódik végre.

Az otherwise szó pedig egy konstans a Prelude-ban, aminek az értéke True :: Bool. Ezt legtöbbször olyan esetekben használjuk, hogy ha biztosan lesz eredmény. Például:

```
ertek :: Int -> String
ertek n
  | n > 0 = "A szam pozitiv"
  | n == 0 = "A szam a 0" else
  | otherwise = "A szam negativ"
```

```
ertek n =
  if n >0 then "A szam pozitiv"
  if n == 0 then "A szam a 0"
  else "A szam negativ"
```

Ha ertek (-1)-et hívok meg, akkor az n > 0 és az n == 0 sem igaz, ezért lesz az utolsó (kizárásos alapon egy szám nagyobb, mint 0, egyenlő 0-val vagy kisebb, mint 0). Vagy egy szám páros vagy páratlan.

Feladatok

Nagybetű – kisbetű

```
upperLower :: Char -> Char
upperLower x
    | elem x ['a'..'z'] = toUpper x
    | elem x ['A'..'Z'] = toLower x
    | otherwise = x
```

Megj.: a *toLower* a Data.Char-ban szerepelnek.

Data.Char.ditigToInt

```
digitToInt :: Char -> Int
digitToInt x
    | elem x ['0'..'9'] = ord x - ord '0'
    | elem x ['A'..'F'] ||  elem x ['a'..'f'] = ord (toUpper x) - ord 'A' + 10
    | otherwise = error "not a digit"
```

Megj.: az *ord* a Data.Char-ban szerepelnek.

Prelude.^

```
(^) :: Num a => a -> Integer -> a
x^n
    | n == 0 = 1
    | mod n 2 == 1 = x * x^(n-1)
    | otherwise = sqr(x^(div n 2))
```

Megj.: akkor működik, ha az *sqr* függvény definiálva van.

Kettes számrendszerbeli számjegyek (fordítottan)

```
toBin :: Integer -> [Int]
toBin n
    | n == 0 = []
    | n == 1 = [1]
    | n `mod` 2 == 0 = [0] ++ toBin (div n 2)
    | n `mod` 2 == 1 = [1] ++ toBin (div (n-1) 2)
```

Megj.: ezt lehet egyszerűbben is, de az nem esetszétválasztás, hanem rekurzió:

```
toBin 0 = []
toBin n = [fromIntegral (mod n 2)]++ toBin (div n 2)
```

Megj 2.: ez fordított, ezért ha rendes átváltást szeretnénk, akkor: *toBin (div n 2) ++ [fromIntegral (mod n 2)]*

Prelude.drop

```
drop :: Int -> [a] -> [a]
drop _ [] = []
drop n l@(x:xs)
    | n <= 0 = l
    | length l <= n = []
    | otherwise = drop (n-1) xs
```

Prelude.take

```
take :: Int -> [a] -> [a]
take _ [] = []
take n l@(x:xs)
    | n <= 0 = []
    | length l <= n = l
    | otherwise = x : take (n-1) xs
```

Data.List.insert

```
insert :: Ord a => a -> [a] -> [a]
insert n [] = [n]
insert n (x:xs)
    | n > x = x : insert n xs
    | n <= x = n : x : xs
```

Rendezett összefűzés

```
sortMerge :: Ord a => [a] -> [a] -> [a]
sortMerge [] l = l
sortMerge l [] = l
sortMerge l@(x:xs) k@(y:ys)
    | x <= y = x : sortMerge xs k
    | otherwise = y : sortMerge l ys
```

where

http://pnyf.inf.elte.hu/fp/Where.xml

A where kulcsszó, amely korlátozza az utána szereplő függvények vagy konstansok láthatóságát. Úgy is mondhatni, hogy lokális definíciók lesznek, amelyek csak az adott függvénydefinícióban szerepelnek. A definiálás sorrendje viszont lényegtelen, ha pedig csak 1 ilyen lokális definíció van, azt lehet a where-rel egy sorban is írni (példa a feladatok között).

```
test l = (a,b)
   where
      a = take 5 l
      b = take 3 a
```

```
test l = (a,b)
   where
      b = take 3 a
      a = take 5 l
```

Mindkettőnek ez az eredménye: test [1..10] == ([1,2,3,4,5],[1,2,3]) :: ([Integer],[Integer])

Példának vegyük a *split* függvényt, amely egy listát kettévág. Definíciója:

```
split :: [a] -> ([a], [a])
split [] = ([],[])
split [x] = ([x],[])
split (x:y:xs) = (x:l1,y:l2)
  where (l1,l2) = split xs
```

split [1..7] => (1 : l1, 2 : l2) where (l1, l2) = split [3..7]

(l1, l2) = split [3..7] => (3 : l1, 4 : l2) where (l1, l2) = split [5,6,7]

(l1,l2) = split [5,6,7] => (5 : l1, 6 : l2) where (l1, l2) = split [7]

(l1, l2) = split [7] => ([7], [])

(1 : [3,5,7], 2: [4,6]) => *([1,3,5,7],[2,4,6])*

(3 : [5,7],4 : [6]) => ([3,5,7],[4,6]), tehát l1=[3,5,7], l2=[4,6]

(5 : [7], 6 : []) => ([5,7],[6]), tehát l1=[5,7], l2=[6]

tehát l1=[7], l2=[]

Feladatok

Prelude.unzip

```
unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([],[])
unzip ((a,b):as) = (a:l1,b:l2)
  where
     (l1,l2) = unzip as
```

Megj.: *unzip l = (map fst l, map snd l)*, de kétszer hivatkozik a listára, ezért időigényes lehet

Prelude.splitAt

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt _ [] = ([],[])
splitAt n l@(x:xs)
 | n <= 0 = ([],l)
 | otherwise = (x:l1,l2)
  where
   (l1,l2) = splitAt (n-1) xs
```

Megj.: *splitAt n l = (take n l, drop n l)*, de kétszer hivatkozik a listára, ezért időigényes lehet

Magasabb rendű függvények bevezetés

http://pnyf.inf.elte.hu/fp/Higherorder.xml

Zárójelezés és a dollárjel

A zárójelezés elsősorban a precedenciától függ, de előfordul, hogy másképp kell használnunk a zárójeleket. Például nem mindegy, hogy (5+2)*3 vagy 5+(2*3). Ha nem írunk zárójelet, akkor 5+2*3 a precedencia miatt 5+(2*3)-ként fog kiértékelődni.

Az elsőbbségi és a kötési táblázat itt található:
http://pnyf.inf.elte.hu/fp/Bools.xml#kötési-erősség-összefoglalás

A dollár operátor ($) zárójelezés szempontjából gyorsabb, de jól kell használni.

```
infixr 0 $
($) :: (a -> b) -> a -> b
($) a b = (a b)
```

Zárójelpárokat lehet vele „rövidíteni”, például

(fst (head [(‘a’,True),(‘a’,False)]))

elejéről és a végéről elhagyható a zárójel, de az *fst* és a *head* közé pedig már kell:

fst (head [(‘a’,True),(‘a’,False)])

Itt viszont felváltható $ jellel:

fst $ head [('a',True),('a',False)]

Ez esetén	div (product [41..70]) (product [1..30])
Csak a 2. helyére lehet tenni	div (product [41..70]) $ product [1..30]
Ha az első helyére tennénk	div $ product [41..70] (product [1..30])
Az így értékelődne ki, ami már hibás	div (product [41..70] (product [1..30]))
Egymásba is ágyazhatóak, például	length (snd (head [('a',"asd"),('a',"dsa")]))
Megegyezik ezzel	length $ snd $ head [('a',"asd"),('a',"dsa")]
De	length (snd (head [('a',"asd"),('a',"dsa")])) + 1
Már nem írható át erre	length $ snd $ head [('a',"asd"),('a',"dsa")] + 1
Hiszen ez ezt jelentené, és ez hibás	length (snd (head [('a',"asd"),('a',"dsa")] + 1))
Ezért így kell átírni	(length $ snd $ head [('a',"asd"),('a',"dsa")]) + 1

Megj.: A ($) felváltása függvénykompozícióval (.) később jön elő.

Névtelen függvények

A névtelen függvények a hozzárendelési szabályoknak felelnek meg. Például az 5+1 felírható (+1) 5 alakban, ez pedig (\ x -> x + 1) 5 formulában.

Felépítésük:

(\ változók -> kifejezés), ahol a változók lehetőleg kicsi, egybetűs nevek legyenek (ez egy függvénynél a paraméternek felel meg). Előző példánál egy olyan x kell, amelyre a (+1) működik.
(\ (a,b) -> a+b) esetén egy rendezett párra van szükség, ahol az összeadás definiálva van.
A joker (_) jel is használható, és érdemesebb, ha valamelyik bemenő adatra nincs szükségünk.
Például: (\ (a,_) -> a) esetén a rendezett pár második eleme számunkra nem lényeges.
Bármilyen típushoz lehet írni névtelen függvényt, és az eredmény típusa is bármi lehet.

Feladatok

Prelude.map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
```

```
map f (x:xs) = f x : map f xs
```

Vagy `map f l = [f x | x <- l]`

Prelude.filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs)
    | f x = x : filter f xs
    | otherwise = filter f xs
```

Vagy `filter f l = [x | x <- l, f x]`

Megjegyzés

filter felt (map fg lista) == [fg x | x <- lista, felt (fg x)]

Számlálás

```
count :: (a -> Bool) -> [a] -> Int
count f l = length $ filter f l
```

Prelude.takeWhile

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile f [] = []
takeWhile f (x:xs)
    | f x = x : takeWhile f xs
    | otherwise = []
```

Prelude.dropWhile

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile f [] = []
dropWhile f (x:xs)
    | f x = dropWhile f xs
    | otherwise = (x:xs)
```

Prelude.span

```
span :: (a -> Bool) -> [a]{-véges-} -> ([a],[a])
span f [] = ([],[])
span f (x:xs)
    | f x = (x:l1,l2)
    | otherwise = ([],(x:xs))
       where (l1,l2) = span f xs
```

Prelude.iterate

```
iterate :: (a -> a) -> a -> [a]
iterate f n = n : iterate f (f n)
```

Predulle.all

```
all :: (a -> Bool) -> [a]{-véges-} -> Bool
all f [] = True
all f (x:xs) = f x && all f xs
```

Prelude.any

```
any :: (a -> Bool) -> [a]{-véges-} -> Bool
any f [] = False
any f (x:xs) = f x || any f xs
```

Prelude.elem

```
elem :: Eq a => a -> [a]{-véges-} -> Bool
elem n l = any (==n) l
```

Több elem szűrése

```
filters :: Eq a => [a] -> [a] -> [a]
filters _ []= []
filters l (x:xs)
    | elem x l = filters l xs
    | otherwise = x : filters l xs
```

Vagy

```
filters l1 l = [x | x <- l, not $ elem x l1]
```

Prelude.zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ _ [] = []
zipWith _ [] _ = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Megj.: *zip l1 l2 == zipWith (,) l1 l2*

Különbségsorozat

```
differences :: Num a => [a] -> [a]
differences [x] = []
differences (x:y:xs) = y - x : differences (y:xs)
```

Vagy

```
differences l = zipWith (-) (tail l) l
```

Fibonacci párok

```
fibPairs :: [(Integer, Integer)]
fibPairs = iterate (\ (a,b) -> (b,a+b)) (0,1)
```

Data.List.group

```
group :: Eq a => [a] -> [[a]]
group [] = []
group l@(x:xs) = a : group b
  where (a,b) = span (==x) l
```

Ismétlődő elemeket tartalmazó lista tömörítése

```
compress :: Eq a => [a] -> [(Int,a)]
compress l = zip (map length $ group l) (map head $ group l)
```

Pascal-háromszög

```
pascalTriangle :: [[Integer]]
pascalTriangle= iterate (\l -> zipWith (+) ([0]++l) (l++[0])) [1]
```

Prelude.uncurry

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry fg (a,b) = fg a b
```

Kitömörítés

```
decompress :: Eq a => [(Int,a)] -> [a]
decompress l = concat [replicate (fst x) (snd x) | x <- l]
```

Súlyozott szöveg

```
weightedSum :: Num a => [(a,a)] -> a
weightedSum l = sum [(fst x) * (snd x) | x <- l]
```

Függvénykompozíció

http://pnyf.inf.elte.hu/fp/Composition.xml

Definíciója

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . p) x = f (p x)                                    vagy (.) f p x = f (p x)
```

Használat

Például a *map*-nél vagy akár a *filter*-nél:

```
map ((*2) . (+1)) [1..5]                    filter ((==3) . length) ["a","aaa","asd"]
map (*2) $ map (+1) [1..5]                  [ x | x <- ["a","aaa","asd"], ((==3).length) x ]
(map (*2) . map (+1)) [1..5]                [ x | x <- ["a","aaa","asd"], (length x) == 3]
[4,6,8,10,12] :: Integer                    ["aaa","asd"] :: [[Char]]

((*2).(+1)) 4      (*2) ((+1) 4)      (*2) (4+1)      (4+1)*2      10 :: Integer
```

De függvények definíciójában is előfordul

```
uniq = map head . group . sort              firstLetters = unwords . map (take 1) . words
uniq l = (map head . group . sort) l        firstLetters l = (unwords . map (take 1) . words) l
uniq l = (map head (group (sort l)))        firstLetters l = (unwords (map (take 1) (words l)))
uniq l = map head $ group $ sort l          firstLetters l = unwords $ map (take 1) $ words l
```

Feladatok

1, 11, 111, 1111, .

```
numbersMadeOfOnes :: [Integer]
numbersMadeOfOnes = iterate ((+1).(*10)) 1
```

3, 33, 333, 3333, .

```
numbersMadeOfThrees :: [Integer]
numbersMadeOfThrees = iterate ((+3).(*10)) 3
```

1, 31, 331, 3331, .

```
numbersMadeOfThreesAndOne :: [Integer]
numbersMadeOfThreesAndOne = iterate ((+21).(*10)) 1
```

Szóközök eldobása elölrol

```
dropSpaces :: String -> String
dropSpaces = dropWhile (==' ')
```

Szóközök eldobása elölrol és hátulról

```
trim :: String{-véges-} -> String
trim = reverse . dropSpaces . reverse . dropSpaces
```

Megj.: akkor működik, ha a *dropSpaces* függvény definiálva van.

Minimumok maximuma

```
maximumOfMinimums :: Ord a => [[a]] -> a
maximumOfMinimums = maximum . map minimum
```

Dupla map

```
mapMap :: (a -> b) -> [[a]] -> [[b]]
mapMap p = map (map p)
```

Monogram

```
monogram :: String -> String
monogram = unwords . map ((++".").(take 1)) . words
```

Egymás utáni ismétlodo elemek kihagyása

```
reduce :: Eq a => [a] -> [a]
reduce = map head . group
```

Egyedi elemek

```
uniq :: Ord a => [a]{-véges-} -> [a]
uniq = map head . group . sort
```

Megj.: a *group* és a *sort* a Data.List-ben vannak

Ismétlődő elemek

```
repeated :: Ord a => [a]{-véges-} -> [a]
repeated = map head . filter ((>1).length) . group . sort
```

Részsorozatok

```
sublists :: [a] -> [[a]]
sublists = concat . map (init . tails) . tail . inits
```

Adott hosszúságú részlisták

```
subListWithLength :: Int -> [a] -> [[a]]
subListWithLength n = filter ((==n).length) . sublists
```

Adott számnál nem hosszabb részlisták

```
subListWithMaxLength :: Int -> [a] -> [[a]]
```

subListWithMaxLength n = filter ((<=n) . length) . concat . map (tail . inits) . init . tails

Prelude.until

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until f p = head . filter (f) . iterate p
```

Hajtogatások

http://pnyf.inf.elte.hu/fp/Folds.xml

Hajtogatások (fold)

Észrevehetjük, hogy a bizonyos rekurzív, listafeldolgozó függvényeink jellemzo hasonlóságot mutatnak egymáshoz. Ezt a közös, sémaként kiemelheto részt ún. hajtogatások (angolul *folding)* segítségével írhatjuk le. A hajtogatás lényege, hogy egy listából egy binér függvény segítségével valamilyen eredményt „hajtogatunk". Ekkor lépésenként összekombináljuk a lista elemeit a függvény által kiszámított eredménnyel.

Jobbról

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Levezetés

foldr (*) 2 [3,5,7] => *(*) 3 (foldr (*) 2 [5,7])* => *(*) 3 ((*) 5 (foldr (*) 2 [7]))*
(*) 3 ((*) 5 ((*) 7 (foldr (*) 2 []))) => *(*) 3 ((*) 5 ((*) 7 2))* => *3 * (5 * (7 * 2))*
*3 * (5 * 14)* => *3 * 70* => *210 :: Integer*

Balról

Fontos különbség, hogy a foldr képes végtelen listákkal is dolgozni, míg a foldl nem.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ e [] = e
foldl f e l = f (foldl f e (init l)) (last l)
```

Egy kiértékelési példa

foldr (\x y -> concat ["(",x,"+",y,")"]) "0" (map show [1..5]) *"(1+(2+(3+(4+(5+0)))))" :: [Char]*
foldl (\x y -> concat ["(",x,"+",y,")"]) "0" (map show [1..5]) *"(((((0+1)+2)+3)+4)+5)" :: [Char]*

Kezdoértékkel

A hajtogatásoknál nem kötelező megadni a kezdőértéket. Létezik olyan változat, ahol a kezdőérték automatikusan a lista első eleme lesz. Ekkor viszont (értelemszerűen) a lista nem lehet üres!

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldr1 :: (a -> a -> a) -> [a] -> a
```

Hajtogatások részeredménnyel (scan)

A hajtogatások másik lehetséges fajtája az, amikor a keletkező részeredményeket összegyűjtjük egy listába és ezt adjuk vissza. Ez pásztázásnak (vagy angolul *scanning)* nevezzük.

Jobbról

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ e [ ] = [e]
scanr f e l = [foldr f e l] ++ scanr f e (tail l)
```

A hajtogatás és pásztázás kapcsolatát az alábbi állítás írja le:

```
head (scanr f z xs) == foldr f z xs
```

Példa

scanr (+) 0 [1..5] => [foldr (+) 0 [1..5] ++ scanr (+) 0 [2..5] =>
[15] ++ [foldr (+) 0 [2..5]] ++ scanr (+) 0 [3,4,5] => [15]++[14] ++ [foldr (+) 0 [3,4,5]] ++ scanr (+) 0 [4,5] =>
[15]++[14]++[12] ++ [foldr (+) 0 [4,5]] ++ scanr (+) 0 [5] => [15]++[14]++[12]++[9] ++ [foldr (+) 0 [5]] ++ scanr (+) 0 []
=> [15]++[14]++[12]++[9]++[5] ++ [0] => [15,14,12,9,5,0] :: Integer

Balról

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

Ebben az esetben a következo összefüggés érvényes:

```
last (scanl f z xs) == foldl f z xs
```

Kezdoérték nélküli változatok

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
scanr1 :: (a -> a -> a) -> [a] -> [a]
```

Feladatok

Prelude.sum

```
sum :: Num a => [a]{-véges-} -> a
sum = foldr (+) 0
```

Prelude.product

```
product :: Num a => [a]{-véges-} -> a
product = foldr (*) 1
```

Prelude.and

```
and :: [Bool]{-véges-} -> Bool
and = foldr (&&) True
```

Prelude.or

```
or :: [Bool]{-véges-} -> Bool
or = foldr (||) False
```

Prelude.concat

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

Prelude.length

```
length :: [a]{-véges-} -> Int
length = foldr (\x y -> y + 1) 0
```

Prelude.reverse

```
reverse :: [a]{-véges-} -> [a]
reverse = foldl (\x y -> y : x) []
```

Prelude.maximum

```
maximum :: Ord a => [a]{-véges, nemüres-} -> a
maximum l = foldr (\ x y -> max x y) (head l) l
```

Bináris szám értéke

```
fromBin :: [Int] -> Integer
fromBin = foldr (\ (a,b) y -> y + (2^a * (fromIntegral b))) 0 . zip [0..]
```

Prelude.minimum

```
minimum :: Ord a => [a]{-véges, nemüres-} -> a
minimum = foldr1 min
```

Megj.:A maximum pedig: *maximum = foldr1 max*

Összegsorozat

```
sums :: [Integer] -> [Integer]
sums = scanl1 (+)
```

Fibonacci sorozat

```
fibs :: [Integer]
fibs = 1 : scanl (+) 0 fibs
```

Növekvő maximumok sorozata

```
increasingMaximums :: Ord a => [a] -> [a]
increasingMaximums = nub . scanl1 max
```

Magasabbrendű függvények kombinálása

http://pnyf.inf.elte.hu/fp/ByFunctions.xml

Compare és Ordering

Az Ordering egy háromértékű adattípus

```
compare :: Ord a => a -> a -> Ordering
```

1 `compare` 2	LT :: Ordering	Less Than (<)
2 `compare` 2	EQ :: Ordering	Equal (==)
3 `compare` 2	GT :: Ordering	Greater than (>)

Az on függvény

Megj.: Ez a Data.Function-ban található

```
on :: (b -> b -> c) -> (a -> b) -> (a -> a -> c)
on f g x y = f (g x) (g y)
```

Vagy `(f `on` g) x y = f (g x) (g y)`

Példa

```
( (==) `on` isDigit ) '1' '2'
(==) ( isDigit '1' ) ( isDigit '2' )
( isDigit '1' ) == ( isDigit '2' )
True :: Bool
```

```
( (+) `on` (*2) ) 1 9
(+) ( (*2) 1 ) ( (*2) 9 )
( 1 * 2 ) + ( 9 * 2 )
20 :: Integer
```

```
( compare `on` length ) "asd" "dsa"
compare ( length "asd" ) ( length "dsa" )
EQ :: Ordering
```

Egyéb függvények

Megj.: Ezek a Data.List-ben találhatóak

groupBy

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

```
groupBy ((==) `on` isDigit) "34 24x +48"          ["34", " ", "24", "x +", "48"] :: [[Char]]
groupBy (\ x y -> (mod (x*y) 3) == 0) [1..9]      [[1], [2, 3], [4], [5, 6], [7], [8, 9]] :: [[Integer]]
```

maximumBy

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

```
maximumBy compare ["alm","fa","ablak"]            "fa" :: [Char]
maximumBy (compare `on` length) ["alm","fa","ablak"]   "ablak" :: [Char]
```

minimumBy

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

```
maximumBy (compare `on` last) ["alm","fa","ablak"]   "alm" :: [Char]
```

sortBy

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

```
sortBy (compare `on` last) ["alm","fa","ablak"]   ["fa", "ablak", "alm"] :: [[Char]]
```

Feladatok

Szövegbeli számok kiszedése

```
numbersInString :: String -> [String]
numbersInString = filter (isDigit . head) . groupBy ((==) `on` isDigit)
```

Szöveg leghosszabb szava

```
longestWord :: String -> String
longestWord = maximumBy (compare `on` length) . words
```

Legtöbbször előforduló karakter

```
mostFrequentChar :: String -> Char
mostFrequentChar = head . maximumBy (compare `on` length) . group . sort
```

Origóhoz legközelebbi pont

```
closestToOrigo :: Real a => [(a, a)] -> (a, a)
closestToOrigo = minimumBy (compare `on` (\(a,b) -> a*a+b*b))
```

Hányadik elem volt a legnagyobb?

```
maxIndex :: Ord a => [a] -> Int
maxIndex = fst . maximumBy (compare `on` snd) . zip [1..]
```

Nagyság szerinti sorrend indexekkel

```
maxIndices :: Ord a => [a] -> [Int]
maxIndices = map fst . sortBy (compare `on` snd) . zip [1..]
```

Prelude.flip

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f n m = f m n
```