Prelude függvények:

```haskell
not :: Bool -> Bool
not True  = False
not False = True

&& :: Bool -> Bool -> Bool
True  && x     = x
_     &&  _     = False

||  :: Bool -> Bool -> Bool
False  || x     = x
_     ||  _     = True

fst:: (a,b) -> a
fst (a,b) = a

snd:: (a,b) -> a
snd (a,b) = b

even :: Integer -> Bool
even x = mod x 2 == 0

odd :: Integer -> Bool
odd x = mod x 2 == 1

null :: [a] -> Bool
null [] = True
null _  = False

head :: [a] -> a
head (x:xs) = x
head [] = error "head of []"

tail :: [a] -> [a]
tail (x:xs) = xs
tail [ ]= error "tail of []"

last :: [a] -> a
last [a] = a
last (x:xs) = last xs
last [] = error "last of []"



init :: [a] -> [a]
init [] = error "empty"
init [a] = []
init (x:xs) = x : init xs

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

drop :: Int -> [a] -> [a]
drop _ [] = []
drop n xs | n<=0 = (xs)
```

```haskell
drop n (x:xs) = drop (n-1) xs


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

length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

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

min :: a -> a -> a
min x y
    | x <= y    =  x
    | otherwise =  y

max :: a -> a -> a
max x y
    | x >= y    =  x
    | otherwise =  y

minimum :: [a] -> a
minimum [a] = a
minimum (x:xs) = min x (minimum xs)

maximum :: [a] -> a
maximum [a] = a
maximum (x:xs) = max x (maximum xs)

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

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

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

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

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

Egyszerű függvények:

```haskell
xor :: Bool -> Bool -> Bool
a `xor` b = a != b

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

mountain :: Integer -> [Integer]
mountain n = [1..n] ++ [n-1,n-2..1]

areTriangleSides :: Double -> Double -> Double -> Bool
areTriangleSides a b c = a + b > c && a + c > b && b + c > a

divides :: Integer -> Integer -> Bool
divides n m = mod m n==0

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

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

properDivisors :: Integer -> [Integer]
properDivizors n = [y|y <- divizors n, y/=1, y/=n]

mirrorX :: (Double, Double) -> (Double, Double)
mirrorX  (a,b) = (a,-b)

mirrorY :: (Double, Double) -> (Double, Double)
mirrorY  (a,b) = (-a,b)

mirrorO :: (Double, Double) -> (Double, Double)
mirrorO  (a,b)  = (-a,-b)

mirrorP :: (Double, Double) -> (Double, Double) -> (Double, Double)
mirrorP  (a,b) (c,d) = (2*c-a,2*d-b)

magnify :: Double -> (Double, Double) -> (Double, Double)
magnify x (a,b) = (x*a,x*b)

distance :: (Double, Double) -> (Double, Double) -> Double
distance (a,b) (c,d) =  (a-c)^2 + (b-d)^2

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

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

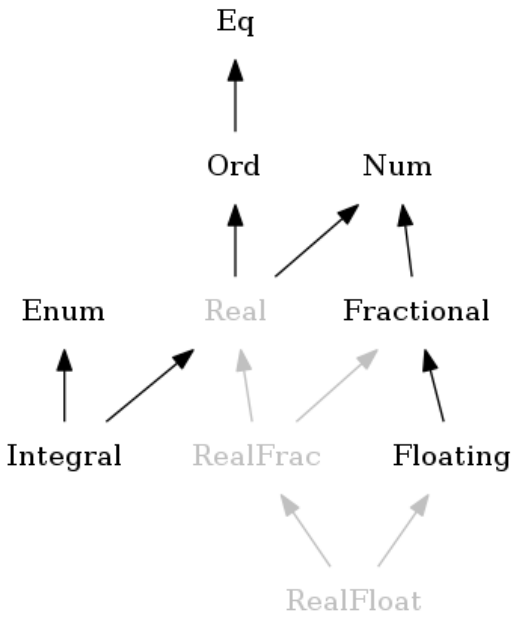
Számok

Integer	egész szám
Int	korlátos egész szám (-2147483648..2147483647)

Rational	racionális szám
Double	dupla pontosságú lebegőpontos szám
Ratio Integer	(= Rational) Két Integer hányadosa
Ratio Int	Két Int hányadosa
Fixed E6	(= Micro) Tizedestört 6 tizedesjeggyel
Fixed E12	(= Pico) Tizedestört 12 tizedesjeggyel
Complex Float	Komplex szám (Float pár)
Complex Double	Komplex szám (Double pár)

,,ad hoc” polimorfizmus

```
(+) :: Num a => a -> a -> a

(+) :: Int      -> Int      -> Int
(+) :: Integer  -> Integer  -> Integer
(+) :: Rational -> Rational -> Rational
(+) :: Double   -> Double   -> Double


(-) :: Num a => a -> a -> a

(*) :: Num a => a -> a -> a

(/) :: Fractional a => a -> a -> a

div :: Integral a => a -> a -> a

mod :: Integral a => a -> a -> a

^) :: (Integral b, Num a) => a -> b -> a

(^^) :: (Fractional a, Integral b) => a -> b -> a

(**) :: Floating a => a -> a -> a
```



Zárójelezés

^, ^^, ** irány: (.(.))

*, / irány: ((.).)

+, - irány: ((.).)

A függvényalkalmazás minden operátornál erősebb:
sin pi + 1 zárójelezése (sin pi) + 1

A függvényneveket `` jelek közé téve infix módon használhatjuk.
Az infix módon használt div és mod kötési erőssége és zárójelezése ugyanaz, mint a * és / operátoroké.
20 `div` 4

Az operátorneveket zárójelek közé téve prefix módon használhatjuk.
A prefix módon használt operátorok kötési erőssége a függvényalkalmazások erősségével egyezik meg (azaz a legerősebb).
(+) 1 2

Konverziók

Test> (1 :: Int) * 5

```
5 :: Int

Test> 1 * (5 :: Int)
5 :: Int

Test> 1 * 5 :: Int
5 :: Int


Test> (2 :: Int) :: Integer
<interactive>:1:2: error:
    • Couldn't match expected type ‘Integer’ with actual type ‘Int’
    • In the expression: (2 :: Int) :: Integer


Test> 3 ^ 4 :: Double
81.0 :: Double
```

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

realToFrac :: (Real a, Fractional b) => a -> b
```

Integral:	Int, Integer
Num:	Int, Integer, Rational, Float, Double
Real:	Int, Integer, Rational, Float, Double
Fractional:	Rational, Float, Double

Kerekítés

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

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

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

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

toEnum :: Enum a => Int -> a

fromEnum :: Enum a => a -> Int

Bool:

True :: Bool

False :: Bool

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

(<) :: Ord a => a -> a -> Bool

Eq a és Ord a jelentése: a kicserélhető majdnem minden típusra, de nem cserélhető ki például függvénytípusra.
Ezek azonos erősségű operátorok és gyengébbek az eddigi operátoroknál.
Ezek az operátorok se jobbra, se balra nem kötő operátorok.
Azonos kötési erősségű, nem kötő operátorok egymás mellett szerepeltetése hibás. Például a == b == c hibás, és a < b > c is hibás.

Test> 'a' < 'x'
True :: Bool

Test> "alma" < "almafa"
True :: Bool

Test> [312,3] < [312,1,3]
False :: Bool

Test> [[]] < [[[]]]
True :: Bool

Test> True && False
False :: Bool

Test> True || False
True :: Bool

Test> not True

False :: Bool

(&&) :: Bool -> Bool -> Bool

(||) :: Bool -> Bool -> Bool

(&&) erősebben köt a (||)-nál!

függvények (sqrt, div, mod, stb.)	((.).)
^, ^^, **	(.(.))
*, /, `div`, `mod`	((.).)
+, - (kivonás és negálás)	((.).)
==, /=, <, <=, >, >=	-
&&	(.(.))
||	(.(.))

Listák

[True, False, False] :: [Bool]

[1, 2, 3] :: Num a => [a]

[] :: [a]

A listák homogének, azaz csak azonos típusú elemeket tartalmazhatnak!

Test>
[1, True]

<interactive>:1:2: error:
- No instance for (Num Bool) arising from the literal ‘1’
- In the expression: 1
  In the expression: [1, True]

'a', 'ű', '\369', '\\', '\n' :: Char

"alma" :: [Char]

"alma", "\n\n", "" :: String

['a', 'b', 'c'] :: [Char]

A String a [Char] szinonimája.

Test> [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] :: [Integer]

Test> [1,3..10]
[1, 3, 5, 7, 9] :: [Integer]

Test> [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103,
 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
 130, 131, 132, 133, 134, 135, …, ……] :: [Integer]

Test> [1,3..]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35,
 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69,
 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101,
 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127,
 129, 131, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153,
 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179,
 181, 183, 185, 187, 189, 191, 193, 195, 197, 199, 201, 203, 205,
 207, 209, 211, 213, 215, 217, 219, 221, 223, 225, 227, 229, 231,
 233, 235, 237, 239, 241, 243, 245, 247, 249, 251, …, ……] :: [Integer]

Test> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz" :: [Char]

length :: [a] -> Int

(!!) :: [a] -> Int -> a

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

Figyelem: (!!) nullától számozza a listaelemeket.

```
Test> :t [[]]
[[]] :: [[a]]

Test> :t [True, False, [False], True]
<interactive>:1:15: error:

Test> :t "alm" ++ ['a']
"alm" ++ ['a'] :: [Char]

Test> :t length [] ++ [1]
<interactive>:1:1: error:

Test> :t [[1,2],[3,4]] ++ [] ++ [[]] ++ [[[]]]
[[1,2],[3,4]] ++ [] ++ [[]] ++ [[[]]] :: Num [a] => [[[a]]]
```

Halmazkifejezések

Matematikai példakép: { n2 | n ∈ N, n páros }

```
Test> [ n^2 | n <- [1..], even n ]
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784,
 900, 1024, 1156, 1296, 1444, 1600, 1764, 1936, 2116, 2304, 2500,
 2704, 2916, 3136, 3364, 3600, 3844, 4096, 4356, 4624, 4900, 5184,
 5476, 5776, 6084, 6400, 6724, 7056, 7396, 7744, 8100, 8464, 8836,
 9216, 9604, 10000, 10404, 10816, 11236, 11664, 12100, 12544, 12996,
 13456, 13924, 14400, 14884, 15376, 15876, 16384, 16900, 17424,
 17956, 18496, 19044, 19600, 20164, 20736, 21316, 21904, 22500,
 23104, 23716, 24336, 24964, 25600, 26244, 26896, 27556, 28224,
 28900, 29584, 30276, 30976, 31684, 32400, 33124, 33856, 34596,
 35344, 36100, 36864, …, ……] :: [Integer]
```

Tetszőlegesen sok generátor és feltétel.
A halmazokkal ellentétben az elemekből több is lehet és számít a sorrend.
A generátorokkal bevezetett új változók csak a generátortól jobbra láthatóak (és a | előtt).

Tehát ez hibás:

```
Test> [ n^2 | even n, n <- [1..] ]
<interactive>:1:14: error: Variable not in scope: n :: Integer
```

A rendezett párok elemei lehetnek különböző típusúak:
(True, 'a') :: (Bool, Char)

```
Test> [(a,b) | a<-"abc", b<-[1,2]]
[('a', 1), ('a', 2), ('b', 1), ('b', 2), ('c', 1), ('c', 2)] :: [(Char, Integer)]
```

```
zip :: [a] -> [b] -> [(a, b)]
```

```
Test> zip "abc" [1,2]
[('a', 1), ('b', 2)] :: [(Char, Integer)]
```

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

```
Test> take 6 [1,3..]
[1, 3, 5, 7, 9, 11] :: [Integer]
```

```
Test> ['a','b','c'] !! 0
'a' :: Char
```

```
Test> take 1 ['a','b','c']
"a" :: [Char]
```

```
Test> take 10 [[1..]]
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
  19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
  36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
  53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
  70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
  87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102,
  103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
  116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
  129, 130, 131, 132, 133, 134, …, ……]] :: [[Integer]]
```

Függvények definiálása

Egy Haskell program deklarációk halmaza.

Deklarációk:

típusdeklarációk: f :: Int -> Int
függvénydefiníciók: f x = x
konstansdefiníciók: pi = 2 * acos 0
operátor definíciók: a <= b = not (a > b)

További deklarációk:

típusdefiníciók: type String = [Char]
típusosztály definíciók: class Num a where …
típusosztály példányosítás: instance Num Int where …

A deklarációk sorrendje nem számít!

Modulok

A definíciókat modulokba rendezzük, a modul a fordítási egység.
Egy Haskell modul szerkezete:

Fejléc (legfelső szintű modulnál nem kell): module modulnév where
import deklarációk
egyéb deklarációk

Minden Haskell modul egy szövegfájl .hs vagy .lhs kiterjesztéssel.

Az egysoros megjegyzések a -- jelekkel kezdődnek, és a sor végéig tartanak.

A többsoros megjegyzéseket a {- és -} jelek határolják. A többsoros megjegyzések egymásba ágyazhatók.

A függvény- és változónevek betűkből és számokból állhatnak és kisbetűvel kell kezdődniük. Tartalmazhatják az aláhúzás és az egyszeres idézőjel karaktereket is: f_1, f', f''.

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

(.*.) :: Double -> Double -> Double
a .*. b = a * b + 1
```

Az operátornevek nem ASCII szimbólumokból és a következő ASCII szimbólumokból állnak: !?.#$%@&*+-~^/|\<=>: Kivételt képeznek az alábbi kulcsszavak, amelyek nem operátornevek: =, .., |, <-, ->, =>, ::, \, @, ~

Mintailleszkedés

Egy függvényt több alternatívával (vagy szabállyal) is definiálhatunk. A függvényalternatíváknak egymás mellett kell elhelyezkedni a modulban.
A függvényparaméterek helyén minták vannak. A minta fogalmával a következő diákon ismerkedünk meg.
A függvényalternatívákat a kiértékelés során fentről lefelé próbáljuk végig.

Minta lehet:

- változó: x, xs, y, a, …
- joker: _
- típus specifikus minták: True, 0, (a, b), …

A joker és a változó minden kifejezésre illeszkedik.

Üres lista minta: []
Egyelemű lista minta: [a]
Kételemű lista minta: [a, b]
…
Nemüres lista minta: a: b

Esetszétválasztás

Példa:

```
min x y
   | x <= y    =  x
   | otherwise =  y
```

Szintaxis:

- Esetek száma: 1, 2, 3, …
- Minden egyes eset: | őrfeltétel = kifejezés
- Az őrfeltétel egy logikai típusú kifejezés.

Szemantika: Az őrfeltételeket fentről lefelé vizsgáljuk, és az első teljesülő feltételnek megfelelő kifejezést választjuk.

A két definíció hasonlít, az eltérések a következők:

- A Haskellben a feltételek és az eredmények oszlopa fel van cserélve (bizonyos szempontból ez a logikusabb).

A nagy kapcsos zárójel helyett függőleges vonalak vannak (technikai okokból).
A kapcsos zárójel előtti egyenlőségjel beljebb került, a feltételeket választja el az eredményektől.

A szintaxis tehát a következő:

Az esetek száma lehet egy, kettő, vagy több.
Minden egyes eset a következő alakú:
| őrfeltétel = kifejezés
Az őrfeltétel egy logikai típusú kifejezés.

Az esetszétválasztás szemantikája: Az őrfeltételeket fentről lefelé vizsgáljuk. A végredmény az első igaz őrfeltétel melletti kifejezés lesz.

Where

Példa:

```
res = f * (f - y)  where

   y = 1 / 4

   f = 6 * y
```

Szintaxis:

A where kulcsszó.
Nagyobb behúzás.
Szemantika: Korlátozott látókör a változók (y, f) számára.

A lokális definíciók korlátozott látókörű konstans és függvénydefiníciók.

A lokális definíciók szintaktikája:

A lokális definíciókat a where kulcsszó vezeti be.
A lokális definíciók behúzása nagyobb, mint a globális definícióké.
Az egy csoportba tartozó lokális definíciók behúzása egyenlő, és nem választhatja szét ezeket kisebb behúzású definíció.
A res globális definíció, azaz a modulban mindenütt látható. Az f és y lokális definíciók, ezek csak egymás számára láthatóak, továbbá láthatóak a where előtti definícióban.

A where kulcsszót érdemes a sor végére tenni. A where írható a következő sorba is, de mindenképp beljebb kell húzni, mint az azt megelőző definíciót.

Lokális definíciót akkor érdemes használni, ha egyidejűleg teljesül a következő két kitétel:

A definíciót csak egyetlen definícióban használjuk (de itt lehet többször is).
A definíciót nem érdemes globálissá tenni, mert túl speciális, vagy lokális definíciókra vagy paraméterekre hivatkozik.

Példa
A függvénykompozíció definiálása lokális definícióval:

(.) :: (b -> c) -> (a -> b) -> (a -> c)

f . g = h where

  h x = f (g x)

A függvénykompozíció definiálása lokális definíció nélkül:

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)

Magasabb rendű függvények

A magasabbrendű függvényeket függvényekkel lehet paraméterezni. A szeletek jól használhatók magasabbrendű függvényekkel.

elemenkénti feldolgozás:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x: map f xs
```

Megjegyzés. A map függvényt így is lehetne definiálni: map f l = [f e | e <- l].

Test> map (+1)   [1..5]
[2, 3, 4, 5, 6] :: [Integer]

Test> map (^2)   [1..5]
[1, 4, 9, 16, 25] :: [Integer]

Test> map (2^)   [1..5]
[2, 4, 8, 16, 32] :: [Integer]

Test> map (`mod` 2) [1..5]
[1, 0, 1, 0, 1] :: [Integer]

Test> map even [1..5]

[False, True, False, True, False] :: [Bool]

szűrés:

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

Test> filter even [1..10]
[2, 4, 6, 8, 10] :: [Integer]

Test> filter (4<) [1..10]
[5, 6, 7, 8, 9, 10] :: [Integer]

Test> filter (<4) [1..10]
[1, 2, 3] :: [Integer]

Test> filter (/= 3) [1,2,3,4,5,4,3,2,1]
[1, 2, 4, 5, 4, 2, 1] :: [Integer]

Megjegyzés. A filter függvényt így is lehetne definiálni: filter p l = [e | e <- l, p e].

adott tulajdonságú elem előfordulásainak száma egy listában:

```
count :: (a -> Bool) -> [a] -> Int
count x [] = 0
count x (y:ys)
 |x==y = (count x ys)+1
 |otherwise= count x ys
```

Test> count (==3) [1,2,3,4,5,4,3,2,1]
2 :: Int

Test> count ("alma" `isPrefixOf`) (words "almafa alma nem alma")
3 :: Int

takeWhile:

```
takeWhile :: (a-> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (a:as)
  | p a = a : takeWhile p as
  | otherwise = []
```

Test> takeWhile (<5) [1,2,3,4,5,6,5,4,3,2]
[1, 2, 3, 4] :: [Integer]

Test> takeWhile (<500) (map (^2) [1..])
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
 256, 289, 324, 361, 400, 441, 484] :: [Integer]

dropWhile:

```
dropWhile :: (a-> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (a:as)
  | p a = as
  | otherwise = dropWhile p as
```

Test> dropWhile (<5) [1,2,3,4,5,6,5,4,3,2]
[5, 6, 5, 4, 3, 2] :: [Integer]

span:

```
span :: (a -> Bool) -> [a]{-véges-} -> ([a],[a])
span p xs = (takeWhile p xs, dropWhile p xs)
```

Test> span (< 3) [1,2,3,4,1,2,3,4]
([1, 2], [3, 4, 1, 2, 3, 4]) :: ([Integer], [Integer])

Test> span (< 9) [1,2,3]
([1, 2, 3], []) :: ([Integer], [Integer])

Test> span (< 0) [1,2,3]
([], [1, 2, 3]) :: ([Integer], [Integer])

iterate:

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

Test> take 10 (iterate (*2) 1)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512] :: [Integer]

Megjegyzés. A függvény alkalmazásának eredménye egy végtelen lista.

A párok egyetlen konstruktora:

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

Noha írhatjuk azt, hogy (,) 2 'c', ehelyett inkább (2,'c')-t szokás írni. Néha kifejezetten hasznos a (,) jelölés. Például a zip egy lehetséges definíciója:

zipWith:

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

Test> zipWith (+) [1,2,3,4,5] [5,6,7,8]
[6, 8, 10, 12] :: [Integer]

Test> zipWith (,) [1,2,3] "hello"
[(1, 'h'), (2, 'e'), (3, 'l')] :: [(Integer, Char)]

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

```
foldl :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldl f e [] = e
foldl f e (x:xs) = f  (foldl f e xs) x
```

Névtelen függvények

Matematikai minta: x ↦ x + 1 (hozzárendelési szabály)

Test> \x -> x + 1 -- ugyanaz, mint a (+ 1)

Test> \x -> 2 * x + 1

Test> \x -> x `mod` 2 == 0 -- ugyanaz, mint az even

A változó tetszőleges kisbetűs név lehet.

Az új változó elfedhet külső változókat vagy függvényeket:

```
Test> \pi -> pi + 1                -- ugyanaz, mint a (+ 1)
```

Jó tanács: Névtelen függvényekben mindig egybetűs változóneveket használjunk.

Minden szelet egyszerűen átírható névtelen függvénnyé; a fordító is ezt teszi.

```
( +1)          ≡      \x -> x + 1
( `mod`5)      ≡      \x -> x `mod` 5
(2^)           ≡      \x -> 2 ^ x
```

A szeletek használata mindenütt javasolt, ahol lehetséges.

Típus

Tetszőleges típust névvel láthatunk el a type kulcsszó segítségével.

Példák:

```
type Name = String
type Complex = (Double, Double)
```

A nevesített típusnak lehet paramétere is:

```
type Two a = (a, a)
```

Ezután igaz:

```
Test> (1,2) :: Two Int
```

```
Test> ('x','y') :: Two Char
```

A jobb oldal szabad változóinak szerepelnie kell a paraméterek között. Viszont lehet olyan paraméter, amelyet nem használunk fel:

```
type T a = Int
```

A newtype kulcsszóval egy új típust hozhatunk létre korábbi típusokból.

```
newtype Name = N String
```

Megadunk:

egy új típusnevet: Name,
egy konstruktornevet: N

és egy létező típust: String.

Az N konstruktor tesz különbséget a Name és a String típus között.

Példa N használatára kifejezésben:

```
x :: Name
x = N "asztal"
```

Példa N használatára mintában:

```
properName :: Name -> Bool
properName (N s) = isUpper (head s)
```

Az új típus annyira új, hogy még egyenlőségvizsgálat sincs rá.

A deriving kulcsszóval különböző műveletek automatikus definícióját kérhetjük:

```
-- newtype Name = N String
--     deriving (Eq, Ord, Show)
```

Eq: egyenlőségvizsgálat
Ord: összehasonlítás
Show: kiírás (szöveggé alakítás)

type és newtype

Példa:

```
type A = String
newtype B = X String
```

Különbségek A és B között:

Az A típus azonos a String típussal, a B nem.

A B típusnak van egy X konstruktora, amely a konvertálást lehetővé teszi B és String között.

type és newtype a felhasználás terén

A type használata a kód olvashatóságát növeli, bevezetésekor csak a típusokat kell módosítani.

A newtype használata a kód olvashatóságát és megbízhatóságát is növeli, bevezetésekor a típusokat is módosítani kell és a definíciókat konvertálásokkal kell kiegészíteni.

Data:

Példa:

```
data IPoint = P Int Int
        deriving (Eq, Ord, Show, Data, Typeable)
```

data kulcsszó
IPoint típuskonstruktor
P konstruktor
paraméterek

```
data Maybe a
   = Nothing
   | Just a
     deriving (Eq, Ord, Show, Typeable, Data)
```

```
data Tuple2 a b   =  Tuple2 a b
Tuple2 'a' True :: Tuple2 Char Bool

data Tuple3 a b c =  Tuple3 a b c
Tuple3 'a' True "xx" :: Tuple3 Char Bool [Char]

data Day = Mon | Tue | Wed | Thu | Fri | San | Sun
 deriving (Eq,Show)
```

```
data Bool
    = False
    | True
       deriving (Eq, Ord, Show, Read, Enum)
```