Funkcionális programozás – függvények

- **Hello World:** main = putStrLn "hello world"
- **N alatt az M:**

over n m = div (product [n,n-1..n-m+1])(product [1..m])

- **Két szám távolsága:** distance n m = abs (n-m)
- **Számok 1től N-ig, majd vissza 1-ig:**

hegycsucs n = [1..n] ++ [n-1,n-2..1]

- **Négyzetszámok 1től N-ig:**

negyzetszamok n = [x^2|x<-[1..n]]

- **Kettő hatványai 0től N-ig:**

kettohatvany n = [2^x|x<-[0..n]]

- **Listában kisebb listák**

parts n = [[1..m]|m<-[1..n]]

Pl.: parts 3 = [[1],[1,2],[1,2,3]]

- **Szorzótábla**

szorzo = [[m+n*10|m<-[0..9]]|n<-[0..9]]

- **Két számjegy összeadó tábla**

addTable = [[m+n|m<-[0..9]]|n<-[0..9]]

- **Piramisok**

pyramid n = [[1..n] ++ [n-1,n-2..1]|n<-[1..n]]

- **Három maximuma**

max3 a b c = max (max a b) c

- **Szökőév**

```
isLeapYear n = mod n 4 == 0 && (mod n 100 /= 0 || mod n 400 == 0)
```

- **Száz osztói**

```
szazoszto = [n|n<-[1..100],mod 100 n == 0]
```

- **Osztója-e**

```
divides n m = m `mod` n == 0
```

- **Osztók divides-el**

```
divisors n = [m |m<-[1..n],divides m n]
```

- **Valódi osztók**

```
properDivisors n = [y|y<-divisors n,y /= 1,y /= n]
```

- **Prímszám-e**

```
isPrime n = [length[m|m<-[1..n],mod n m == 0,m/=1,m/=n]] == [0]
```

- **Pitagoraszi számhármasok**

```
ps = [(a,b,c)|a<-[1..9],b<-[1..9],c<-[1..9],a^2+b^2==c^2]
```

- **Pont tükrözése az Y tengelyre**

```
mirrorY (x, y) = (-x,y)
```

- **Vektornyújtás**

```
scale l (x,y) = (l*x,l*y)
```

- **Lista első eleme**

```
head' (x: xs) = x
```

- **Lista utolsó eleme**

```
tail' (x: xs) = xs
```

- **Első n elem**

```
elsőKétElem (x: (y: ys)) = (x,y)

elsőHáromElem (x: (y: (z: zs))) = (x,y,z)
```

- **Sum**

```
sum' [] = 0

sum' (x: xs) = x + sum' xs
```

- **Product**

```
product' [] = 1

product' (x: xs) = x * product' xs
```

- **Maximum**

```
maximum' (x:[]) = x

maximum' (x: xs) = max x (maximum' xs)
```

- **Minimum**

```
minimum' (x:[]) = x

minimum' (x: xs) = min x (minimum' xs)
```

- **Első pontra tükrözzük a másodikat**

```
mirrorPoint (a,b) (x,y) = ((2*a)-x,(2*b)-y)
```

- **Horner**

```
horner (x: []) n = x

horner (x: xs) n = x + n * ((horner xs) n)
```

- **Faktoriális**

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

- **Páratlan számok szorzata**

```
semifact :: Integer{- pozitív páratlan szám-} -> Integer
semifact 1 = 1
semifact n = n*semifact(n-2)
```

- **Fibonacci adott N. elemei**

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib(n-1)+fib(n-2)
```

- **Fibonacci sorozat N-ig**

```
fiboo n = [fib n|n<-[1..n]]
```

- **++ újradefiniálása**

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

- **Zip**

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

- **Két szám szorzata**

```
mul :: Integer -> Integer -> Integer
mul n 0 = 0
mul 0 m = 0
mul n m = n + mul (m-1) n
```

- **Hatványozás**

```
pow :: Integer -> Integer -> Integer
pow m 0 = 1
pow m n = m * pow m (n-1)
```

- **Listában szereplő karakterek UniCode összegzése**

```
sumCodes :: [Char] -> Int
sumCodes [] = 0
sumCodes (x:xs) = fromEnum (x :: Char) + sumCodes xs
```

- **Függvény, mely elhagyja a lista utolsó két elemét**

```
init2 :: [a] -> [a]
init2 [x,y] = []
init2 (x:xs) = x:init2 xs
```

- **Megállapítja, hogy egy elem nincs benne a listában**

```
nottElem :: Eq a => a -> [a] -> Bool
nottElem _ [] = True
nottElem a (x:xs) = a /= x && nottElem a xs
```

- **Egyszer az egyikből, egyszer a másikból beszúrás**

```
merge' [] [] = []
merge' [] xs = xs
merge' xs [] = xs
merge' (x:xs) (y:ys) = x:y:merge' xs ys
```

- **Cikk-cakkos listaösszefűzés**

```
merge2 :: [a]->[a]->[a]
merge2 _ [] = []
merge2 (x:xs)(y:ys) = x:y: merge2 ys xs
```

- **3 listás zip**

```
zip3' :: [a]->[b]->[c]->[(a,b,c)]
zip3' (a:as) (b:bs) (c:cs) = (a,b,c) : zip3' as bs cs
zip3' _ _ _ = []
```

- **Minden elem páros-e**

```
allEven :: [Int] -> Bool
allEven [] = True
allEven (x:xs) = mod (x) (2) == 0 && allEven xs
```

- **Páros elemek kiszűrése**

```
evens :: [Int] -> [Int]
evens xs = [a|a<-xs,mod (a)(2) == 0]
```

- **Darabszámlálás**

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

- **Minden második elem**

```
everySecond :: [a] -> [a]
everySecond [] = []
everySecond [x] = []
everySecond (x:y:ys) = y : everySecond ys
```

- **Megszámolja a space karaktereket**

```
numOfSpaces :: [Char]->Int
numOfSpaces [] = 0
numOfSpaces (x:xs)
  |x==' ' = (numOfSpaces xs)+1
  |otherwise = (numOfSpaces xs)
```

- **Adott számú elem vétele egy listából**

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

- **Adott számú elem eldobása egy listából**

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

- **Minden n-edik elem kiírása**

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

everyNth n [] = []

everyNth n xs = (take 1 (drop (n-1) xs)) ++ (everyNth n (drop (n) xs))
```

- **Egy szám a valahányadikon**

```
sqr x = x * x

x ^^^ 0 = 1

x ^^^ n | odd n = x * (x ^^^ (n-1))

x ^^^ n = sqr(x ^^^ div n 2)
```

- **Rendezés**

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

sortMerge [] xs = xs

sortMerge xs [] = xs

sortMerge (x:xs) (y:ys)

  | x <  y = x : sortMerge xs (y:ys)

  | x >= y = y : sortMerge (x:xs) ys
```

- **Minden elemre egy adott művelet végrehajtása**

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

- **Adott tulajdonságú elemek kiírása**

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

- **Egy elem beszúrása rendezett listába**

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

- **Lista elvágása egy adott elemnél**

```
splitAt' :: Int -> [a] -> ([a],[a])
splitAt' _ [] = ([],[])
splitAt' n xs | n <=0 = ([],xs)
splitAt' n (x:xs) = (x:as,bs)
  where
    (as,bs) = splitAt' (n-1) xs
```

- **Páros és páratlan indexű elemek szétválogatása**

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

- **Addig írja ki az elemeket amíg egy feltétel igaz**

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