Funkcionális programozás

Horváth Zoltán

Eötvös Loránd Tudományegyetem, Budapest
Programozási Nyelvek és Fordítóprogramok Tanszék
E-mail: hz@inf.elte.hu

Ajánlott irodalom

- Plasmeijer et al.: Programming in Clean, http://www.cs.kun.nl/~clean
- Peter Achten: Object IO tutorial, http://www.cs.kun.nl/~clean
- Simon Thompson: Haskell - The Craft of Functional Programming, Addison-Wesley, 1999
- Nyékyné G. J. (szerk): Programozási nyelvek, Bp. 2003. Kiskapu. Horváth Z.: A funkcionális programozás nyelvi elemei, fejezet, 56 oldal

Követelmények

- Hetente (kéthetente) kis beadandó programok
- Beadandó programok feltöltése, futtatása
- Működő beadandó programok forrásszövegének ellenőrzése gyakorlatok elején
- Nagyobb beadandó programok
- Géptermi számonkérés (minimum szint + döntés a megajánlott jegyről)

1. Bevezetés

Funkcionális programozási nyelvek

- A deklaratív nyelvekhez tartoznak: a számítási folyamat leírása deklarációk halmaza
- Típus-, osztály-, függvénydefiníciók, kezdeti kifejezés
- A program végrehajtása a kezdeti kifejezés kiértékelése (átírás)
- A matematikai számítási modellje a λ-kalkulus (Church, 1932-33)

Kiértékelés

- Kiértékelés = átírási lépések sorozata (redukció)
- Függvénydefiníció – példa: `sqr x = x * x`
  függvény azonosítója, formális paraméterek, függvény törzs (kifejezés)
- Kiszámíthatóság, hatékonyság
- Átírási lépés: függvény alkalmazásában a függvény helyettesítése a függvény törzsével (a normál forma eléréséig)
- Kiértékelési stratégia: redexek (reducible expressions) kiválasztási sorrendje, például lusta (először a függvény), mohó (először az argumentumok) vagy párhuzamos
- Egyértelmű normál forma (konfluens átíró rendszerekben), a lusta kiértékelés mindig megtalálja a normál formát, ha az létezik

Példák

Clean

```
inc  x       = x + 1
square x     = x * x
squareinc x = square (inc x)
fact n       = prod [1..n]

fact 10
squareinc 7
```

Haskell

```
inc x = x + 1
square x = x^2
squareinc x = (square (inc x))
fact n = product [1..n]

fact 10
squareinc 7
```

Szigorú (strict) kiértékelés:

```
squareinc 7 -> square (inc 7)
            -> square (7 + 1)
            -> square 8
            -> 8 * 8
            -> 64
```

Lusta (lazy) kiértékelés:

```
squareinc 7 -> square (inc 7)
            -> (inc 7) * (inc 7)
            -> 8 * (inc 7)
            -> 8 * 8
            -> 64
```

Modern funkcionális programozási nyelvek jellemzése

- Nincs előző értéket megsemmisítő értékadás
- Hivatkozási helyfüggetlenség – egyenlőségi érvelés (azonos kifejezés értéke mindig ugyanaz)
- Szigorúan típusos (minden részkifejezésnek fordítási időben meghatározott a típusa), típuslevezetés, polimorfizmus, absztrakt és algebrai adattípusok
- Magasabbrendű függvények (az argumentum vagy érték is függvény)
  `twice f x = f (f x)`
- Curry-féle módszer – minden függvénynek 1 argumentuma van
  `(+) x y` kontra `((+) x) y`
- Rekurzió
- Lusta kiértékelés a mohóság vizsgálatával
  `f x = 0; f (5 + 1); 2 * (5 + 1)`

- Zermelo-Fraenkel halmazkifejezések

  Clean

```
{x * x \\ x <- [1..] | odd(x)}
```

  Haskell

```
[x * x | x <- [1..], odd x]
```

- Argumentumok mintaillesztése

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

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

- Margószabály

```
add4 = twice succ   where
  succ x = x + 2
  add    = ... succ ...
```

```
add4 = twice succ   where
  succ x = x + 2
  add    = ... succ ...
```

- I/O modellek: I/O adatfolyam, monádok, egyszeres hivatkozás

Egyszerű funkcionális programok

Clean

```
module Test
import StdEnv

Start =
  // 5 + 2 * 3
  // sum [1..10]
  // reverse (sort [1, 6, 2, 7])
  // 1 < 2 && 3 < 4
  // 2 < 1 || 3 < 4
  // [1, 2] ++ [3, 4, 5]
  // and [True, 2 < 1, 6 > 5 ]
  // take 3 [1, 2, 3, 4, 5]
  map my_abs2 [7, -4, 3]
```

Haskell

```
module Test  where


main = print $
  -- 5 + 2 * 3
  -- sum [1..10]
  -- reverse (sort [1, 6, 2, 7])
  -- 1 < 2 && 3 < 3
  -- 2 < 1 || 3 < 4
  -- [1, 2] ++ [3, 4, 5]
  -- and [True, 2 < 1, 6 > 5]
  -- take 3 [1, 2, 3, 4, 5]
  map my_abs2 [7, -4, 3]
```

Egyszerű funkcionális programok

Clean

```
my_abs x
  | x < 0  = ~x
  | x >= 0 =  x

my_abs2 x
  | x < 0     = ~x
  | otherwise =  x
```

Haskell

```
my_abs x
  | x < 0  = -x
  | x >= 0 =  x

my_abs2 x
  | x < 0     = -x
  | otherwise =  x
```

Másodfokú egyenlet

Clean

```
module Quadratic
import StdEnv

qeq :: Real Real Real
    -> (String, [Real])

Start = qeq 1.0 (-4.0) 1.0
```

Haskell

```
module Quadratic  where


qeq :: Double -> Double -> Double
    -> (String, [Double])

main = print (qeq 1 (-4) 1)
```

Másodfokú egyenlet

Clean

```
qeq a b c
 | a == 0.0 = ("nem másodfokú" ,[])
 | d <  0.0 = ("komplex gyökök",[])
 | d == 0.0 = ("egy gyök",
              [~b / (2.0 * a)])
 | d >  0.0 = ("két gyök",
              [(~b + r) / (2.0 * a)
              ,(~b - r) / (2.0 * a)])
 where
   d = b * b - 4.0 * a * c
   r = sqrt d
```

Haskell

```
qeq a b c
  | a == 0  = ("nem másodfokú", [])
  | d <  0  = ("komplex gyökök", [])
  | d == 0  = ("egy gyök",
                [-b / (2 * a)])
  | d >  0  = ("két gyök",
                [(-b + r) / (2 * a)
                ,(-b - r) / (2 * a)])
  where
    d = b^2 - 4 * a * c
    r = sqrt d
```

8 királynő

Clean

```
module Queens
import StdEnv

queens 0 = [[]]
queens n = [ [q:b]
           \\ b <- queens (n - 1)
           ,  q <- [0..7]
           |  safe q b ]

safe q b = and [not (checks q b i)
         \\ i <- [0..(length b)-1]]

checks q b i
  =   q  ==  b !! i
  ||  abs (q - b !! i)  ==  i + 1

Start
  = (length (queens 8), queens 8)
```

Haskell

```
module Queens  where


queens 0 = [[]]
queens n = [ q:b
           | b <- queens (n - 1)
           , q <- [0..7]
           , safe q b ]

safe q b = and [not (checks q b i)
         | i <- [0..(length b) - 1]]

checks q b i
  =   q  ==  b !! i
  ||  abs (q - b !! i)  ==  i + 1

main = print
  (length (queens 8), queens 8)
```

Egyszerű I/O

Clean

```
module HelloConsole
import StdEnv

Start :: *World -> *World
Start w
 # (con,w)     = stdio w
 # con = fwrites "Neved? " con
 # (name, con) = freadline con
 # con
    = fwrites ("Szia " +++ name) con
 # (_, con)    = freadline con
 # (ok, nw)    = fclose con w
 | not ok    = abort "hiba"
 | otherwise = nw
```

Haskell

```
module HelloConsole  where


main :: IO ()
main = do

  putStr "Neved? "
  name <- getLine

  putStrLn ("Szia " ++ name)
  _ <- getLine


  return ()
```

Tesztkörnyezet (Clean)

```
module functiontest
import funtest, StdClass, StdEnv

Start :: *World -> *World
Start w =  functionTest funs w

dubl :: Int -> Int
dubl x =  x * 2

plus :: Int Int -> Int
plus x y = x+y

fl :: [[Int]] -> [Int]
fl a = flatten a

funs :: [(([String] -> String),[String],String)]
funs = [ (one_arg dubl, ["2"] , "dubl"),
         (two_arg plus, ["2","10"] , "plus"),
         (no_arg "Hello world", [], "Program"),
         (one_arg fl, ["[[1,2,3,4],[],[4]]"],"flatten")]
```

2. Egyszerűbb elemek

Mintaillesztés

Clean

```
hd [x:xs] = x   // parciális
tl [x:xs] = xs  // parciális

fac 0 = 1
fac n | n > 0   // parciális
  = n * fac (n - 1)

sum []     = 0
sum [x:xs] = x + sum xs

length []     = 0
length [_:xs] = 1 + length xs
```

Haskell

```
head (x:xs) = x   -- parciális
tail (x:xs) = xs  -- parciális

fac 0 = 1
fac n | n > 0     -- parciális
  = n * fac (n - 1)

sum []     = 0
sum (x:xs) = x + sum xs

length []     = 0
length (_:xs) = 1 + length xs
```

Típusellenőrzés

Clean

```
1 + True  // Type error: "argument 2 of +"
          // cannot unify demanded type Int
          // with Bool
length 3  // "argument 1 of length" cannot
          // unify demanded type (a b) |
          // length a with Int
```

Haskell

```
1 + True  -- No instance for (Num Bool)
          --  arising from a use of ‘+’
length 3  -- No instance for (Num [a])
          --  arising from the literal ‘3’
```

Típusdefiníciók

Clean

Alaptípusok: Int, Real, Bool, Char

```
Start :: Int
Start = 3 + 4

x :: [Int]
x =  [1, 2, 3]

y :: [Bool]
y =  [True, True, False]

z :: [[Int]]
z =  [[1, 2, 3], [1, 2]]

sum  :: [Int] -> Int
sqrt :: Real  -> Real
```

Haskell

Alaptípusok: Int, Integer, Float, Double, Bool, Char

```
start :: Int
start = 3 + 4

x :: [Int]
x =  [1, 2, 3]

y :: [Bool]
y =  [True, True, False]

z :: [[Int]]
z =  [[1, 2, 3], [1, 2]]

sum  :: Num a => a -> a
sqrt :: Floating a => a -> a
```

Annotációk (Clean)

A típusdefiníciókban szereplő különböző annotációk (`!`, `*` stb.) például az argumentumok szigorú kiértékelését vagy unique típusú hivatkozását adják meg.

Polimorf típusok

Típusváltozókat tartalmazó típusok. A polimorf típusokkal dolgozó függvényeket polimorf függvényeknek nevezzük.

Clean

```
length :: [a] -> Int
hd     :: [a] -> a
```

Haskell

```
length :: [a] -> Int
head   :: [a] -> a
```

Az a egy típusváltozó, ezek mindig kisbetűvel kezdődnek. A polimorf függvények működése nem függ a tényleges típusuktól.

Túlterhelés, ,,ad hoc" polimorfizmus, osztályok

A + függvények több példánya létezik, a + viselkedése a konkrét típustól függ.
A szignatúra minden esetben megegyezik.

Clean

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

Haskell

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

Típusosztályok segítségével tudunk azonos szignatúrával rendelkező, túlterhelt azonosítókat deklarálni.

```
class (+) infixl 6 a :: !a !a -> a
// szigorú kiértékelésü absztrakt
// (+) függvények

// ha létezik (+) példánya, akkor
// a double példánya is létezik
double :: a a -> a | + a
double n :== n + n
```

```
infixl 6 +

class Num a  where
  ...
  (+) :: a -> a -> a
  ...

double :: Num a => a -> a
double n = n + n
```

A példányok definíciója szabályos helyettesítésekkel:

Clean

```
instance + Bool  where
  (+) :: Bool Bool -> Bool
  (+) True b = True
  (+) a    b = b
```

Haskell

```
instance Num Bool  where
  ...
  True + b = True
  a    + b = b
  ...
```

Szinonímák

- Globális konstansok: csak egyszer értékelődnek ki (futási időben), újrafelhasználhatóak. Optimalizáció: növekszik a memóriaigény, csökkenhető viszont a futási idő.

  Clean

  ```
  smallodds =: [1, 3 .. 10000]
  ```

  Haskell

  ```
  smallodds = [1, 3 .. 10000]
  ```

- Típusszinonímák (fordítási időben cserélődnek)

  ```
  :: Color :== Int
  ```

  ```
  type Color = Int
  ```

- Makrók: kifejezések szinonímái (fordítási időben cserélődnek)

  ```
  Black :== 1
  White :== 0
  ```

  ```
  black = 1
  white = 0
  ```

Magasabbrendű listafüggvények

`filter` – adott tulajdonságot teljesítő elemek leválogatása

Clean

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

even x =  x mod 2 == 0

odd    =  not o even
// odd x = not (even x)

evens = filter even [0..]
```

Haskell

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

even x =  x `mod` 2 == 0

odd = not . even
-- odd x = not (even x)

evens = filter even [0..]
```

`map` – függvény elemenkénti alkalmazása (hossztartó)

Clean

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

odds = map inc evens
```

Haskell

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

odds = map (+ 1) evens
```

`foldr` – elemenkénti fogyasztás

Clean

```
foldr :: (.a -> .(.b -> .b)) .b ![.a] -> .b

foldr f e []     = e
foldr f e [x:xs] = f x (foldr f e xs)

sum = foldr (+) 0
// sum xs = foldr (+) 0 xs

and = foldr (&&) True
```

Haskell

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

foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)

sum = foldr (+) 0
-- sum xs = foldr (+) 0 xs

and = foldr (&&) True
```

`takeWhile` – elemek megtartása amíg p teljesül,
`dropWhile` – elemek eldobása amíg p teljesül

Clean

```
takeWhile p [] = []
takeWhile p [x:xs]
  | p x        = [x : takeWhile p xs]
  | otherwise  = []
```

Haskell

```
takeWhile p [] = []
takeWhile p (x:xs)
  | p x        = x : takeWhile p xs
  | otherwise  = []
```

Iterálás

`f` iterálása amíg p nem teljesül:

Clean

```
until :: (a -> Bool) (a -> a) a -> a
until p f x
  | p x        = x
  | otherwise  = until p f (f x)

powerOfTwo = until ((<) 1000) ((*) 2) 1
```

Haskell

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x
  | p x        = x
  | otherwise  = until p f (f x)

powerOfTwo = until (1000 <) (2 *) 1
```

Példa: négyzetgyök számítása Newton-iterációval

Clean

```
sqrtn :: Real -> Real
sqrtn x = until goodEnough improve 1.0
  where
    improve y    = (y + x / y) / 2.0
    goodEnough y = (y * y) ~=~ x
    (~=~) a b    = abs (a - b) < 0.000001
```

Haskell

```
sqrtn :: Double -> Double
sqrtn x = until goodEnough improve 1
  where
    improve y    = (y + x / y) / 2
    goodEnough y = y^2 ~=~ x
    (~=~) a b    = abs (a - b) < 0.000001
```

3. Listák

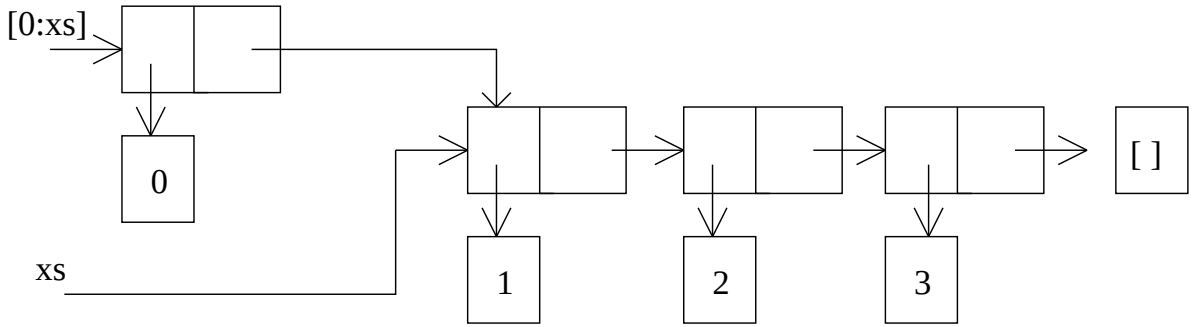
Clean

```
[1, 2, 3 * x, length [1, 2]] :: [Int]

[sin, cos, tan] :: [Real -> Real]
[]                          :: a
[3 < 4, a == 5, p && q] :: [Bool]

[1, 3.. 12], [100, 80..]
```

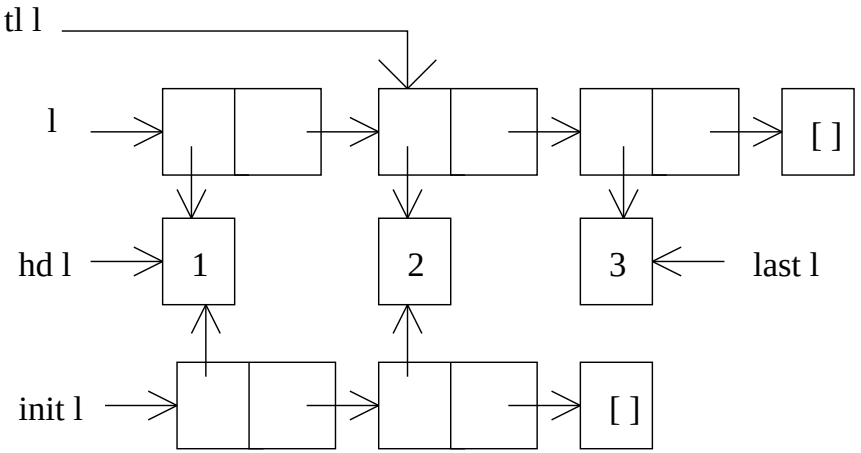
Haskell

```
[1, 2, 3 * x, length [1, 2]] :: [Int]

[sin, cos, tan] :: [Double -> Double]
[]                          :: a
[3 < 4, a == 5, p && q] :: [Bool]

[1, 3.. 12], [100, 80..]
```

Egy lista ábrázolása: `xs = [1, 2, 3]`. Gerinc és elemek. A lista bővítése egy elemmel.



Szabványos listafüggvények

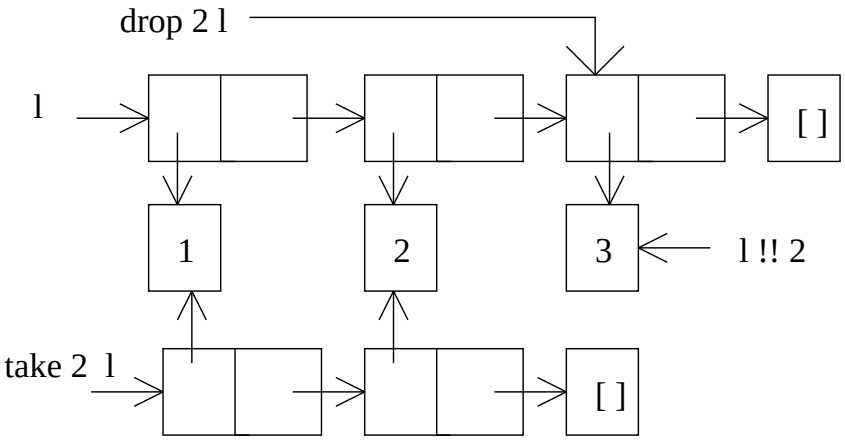


Clean

```
hd [x:xs]   = x
hd []       = abort "hd of []"

tl [x:xs]   = xs
tl []       = abort "tl of []"

last [x]    = x
last [x:xs] = last xs
last []     = abort "last of []"

init []     = []
init [x]    = []
init [x:xs] = [x : init xs]
```

Haskell

```
head (x:xs) = x
head []     = error "head of []"

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

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

init []     = error "empty list"
init [x]    = []
init (x:xs) = x : init xs
```



Clean

```
(!!) infixl 9 :: [a] Int -> a
(!!) [] _   = subscript_error
(!!) list i = index list i
   where  index [x:xs] 0 = x
          index [x:xs] n = index xs (n - 1)
          index [] _     = subscript_error

take 0 _      = []
take n [x:xs] = [x : take (dec n) xs]
take n []     = []
```

Haskell

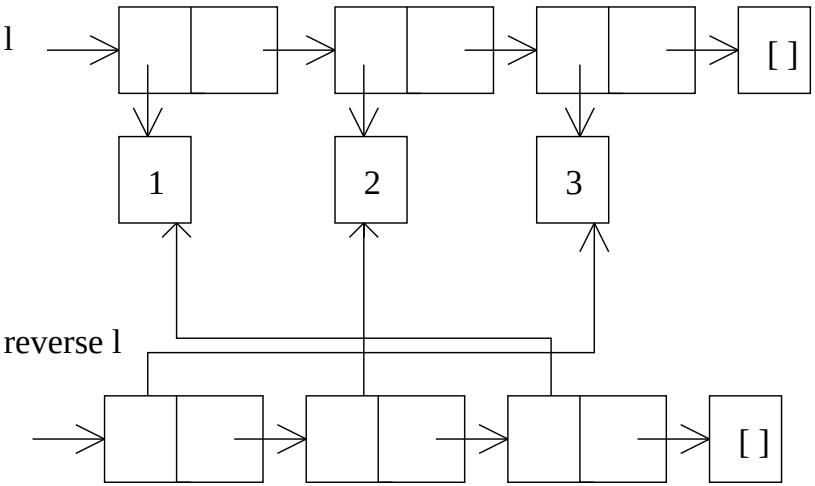
```
infixl 9 !!
(!!) :: [a] -> Int -> a
[]     !! _ = error "subscript"
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n - 1)


take 0 _      = []
take n (x:xs) = x : take (n - 1) xs
take n []     = []
```

Clean

```
drop n cons=:[x:xs]
   | n > 0  = drop (n - 1) xs
            = cons
drop n [] = []

(%) list (frm, to)
  = take (to - frm + 1) (drop frm list)

reverse ls = rev ls []  where
  rev [x:xs] ys = rev xs [x:ys]
  rev []     ys = ys
```

Haskell

```
drop n cons@(x:xs)
  | n > 0     = drop (n - 1) xs
  | otherwise = cons
drop n []     = []

list % (frm, to)
  = take (to - frm + 1) (drop frm list)

reverse ls = rev ls []  where
  rev (x:xs) ys = rev xs (x:ys)
  rev []     ys = ys
```



Clean

```
(++) infixr 5 :: [a] [a] -> [a]
(++) [x:xs] list  = [x:xs ++ list]
(++) []     list  = list

length xs = acclen 0 xs  where
  acclen n [x:xs] = acclen (inc n) xs
  acclen n []     = n

isMember e [x:xs] = x == e || isMember e xs
isMember e []     = False

flatten [x:xs]  = x ++ flatten xs
flatten []      = []
```

Haskell

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
(x:xs) ++ list = x : xs ++ list
[]     ++ list = list

length xs = acclen 0 xs  where
  acclen n (x:xs) = acclen (n + 1) xs
  acclen n []     = n

x `elem` (x:xs) = x == e || x `elem` xs
x `elem` []     = False

concat (x:xs) = x ++ concat xs
concat []     = []
```

Clean

```
instance == [a] | Eq a  where
  (==) [] []    = True
  (==) [] _     = False
  (==) [_:_] [] = False
  (==) [a:as] [b:bs]
      | a == b  = as == bs
                = False

instance < [a] | Ord a  where
  (<) []     []     = False
  (<) []     _      = True
  (<) [_:_]  []     = False
  (<) [a:as] [b:bs]
    | a < b         = True
    | a > b         = False
                    = as < bs
```

Haskell

```
instance Eq a => Eq [a]  where
  []     == []     = True
  (a:as) == (b:bs) = a == b && a == bs
  _      == _      = False

instance Ord a => Ord [a]  where
  _      <  []     = False
  []     <  _      = True

  (a:as) <  (b:bs)
    =   a <  b
    ||  a == b && as < bs
```

Clean

```
repeat x = cons  where
  cons = [x:cons]
  // repeat 3 = [3, 3..]

iterate f x  = [x : iterate f (f x)]
  // iterate inc 3 = [3, 4..]

removeAt 0 [x:xs] = xs
removeAt n [x:xs]
  = [x : removeAt (n - 1) xs]
removeAt n []     = []
```

Haskell

```
repeat x = cons  where
  cons = x : cons
  -- repeat 3 = [3, 3..]

iterate f x = [x : iterate f (f x)]
  -- iterate (+ 1) 3 = [3, 4..]

removeAt 0 (x:xs) = xs
removeAt n (x:xs)
  = x : removeAt (n - 1) xs
removeAt n []     = []
```

Beszúrásos rendezés

Clean

```
insert :: a [a] -> [a] | Ord a
insert e [] = [e]
insert e [x:xs]
   | e <= x    = [e, x:xs]
   | otherwise = [x : insert e xs]

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

Haskell

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

isort :: Ord a => [a] -> [a]
isort []     = []
isort (x:xs) = x `insert` isort xs
```

Összefésüléses rendezés

Clean

```
merge []     ys     = ys
merge xs     []     = xs
merge [x:xs] [y:ys]
  | x <= y     = [x : merge xs [y:ys]]
  | otherwise  = [y : merge [x:xs] ys]

msort :: [a] -> [a] | Ord a
msort xs
  | len <= 1  = xs
  | otherwise = merge (msort ys) (msort zs)
  where
    ys = take half xs
    zs = drop half xs
    half = len / 2
    len = length xs
```

Haskell

```
[]     `merge` ys     = ys
xs     `merge` []     = xs
(x:xs) `merge` (y:ys)
  | x <= y    = x : (xs `merge` (y:ys))
  | otherwise = y : ((x:xs) `merge` ys)

msort :: Ord a => [a] -> [a]
msort xs
  | len <= 1  = xs
  | otherwise = msort ys `merge` msort zs
  where
    ys   = take half xs
    zs   = drop half xs
    half = len `div` 2
    len  = length xs
```

Gyorsrendezés / Listaabsztrakciók

Clean

```
qsort :: [a] -> [a] | Ord a
qsort [] = []
qsort [a:xs]
  =  qsort [x \\ x <- xs | x <= a]
  ++ [a]
  ++ qsort [x \\ x <- xs | x >  a]

sieve [p:xs]
  = [p: sieve [i \\ i <- xs | i mod p <> 0]]
// take 100 (sieve [2..])
```

Haskell

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

sieve (p:xs)
  = p: sieve [i | i <- xs, i `mod` p /= 0]
-- take 100 (sieve [2..])
```

Ortogonális generátorok:

Clean

```
[ (x,y) \\ x <- [1..4], y <- [1..x]
  | isEven x ]
// [(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

Haskell

```
[ (x,y) | x <- [1..4], y <- [1..x]
  , even x ]
-- [(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

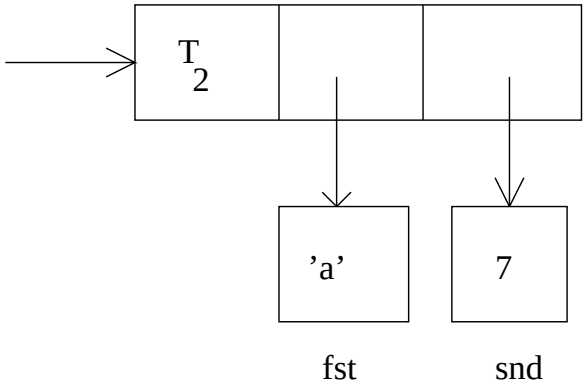
A legbenső változik a leggyorsabban, egy belső generátor változójának értéke nem használható a megelőző generátorokban.

Párhuzamos generátorok:

```
[ x * y \\ x <- [1..2] & y <- [4..6] ]
// [4, 10]
```

```
[ x * y | (x, y) <- zip [1..2] [4..6] ]
-- [4, 10]
```

4. Rendezett n-esek és rekordok



Clean

```
fst (x, _) = x
snd (_, y) = y

splitAt 0     xs    = ([], xs)
splitAt _     []    = ([], [])
splitAt n [x:xs]    = ([x:xs‘], xs‘‘)
  where (xs‘,xs‘‘)  = splitAt (n - 1) xs

zip2 [a:as] [b:bs]  = [(a,b) : zip2 as bs]
zip2 as bs          = []
```

Haskell

```
fst (x, _) = x
snd (_, y) = y

splitAt 0 xs        = ([], xs)
splitAt _ []        = ([], [])
splitAt n (x:xs)    = (x:xs', x'')
  where (xs', xs'') = splitAt (n - 1) xs

zip (a:as) (b:bs)   = (a,b) : zip as bs
zip _      _        = []
```

Clean

```
average ls = s / toReal l
  where
    (s, l) = sumlength list 0.0 0
    sumlength [x:xs] sum l
      = sumlength xs (sum + x) (l + 1)
    sumlength [] sum l = (sum, l)

search [] s   = abort "none"
search [(x,y):ts] s
  | x == s    = y
  | otherwise = search ts s

book = [(1,'a'),(2,'b'),(3,'c')]
// search book 1
```

Haskell

```
average ls = s / fromIntegral l
  where
    (s, l) = sumlength list 0 0
    sumlength (x:xs) sum l
      = sumlength xs (sum + x) (l + 1)
    sumlength [] sum l = (sum, l)

search [] s   = error "none"
search ((x,y):ts) s
  | x == s    = y
  | otherwise = search ts s

book = [(1,'a'),(2,'b'),(3,'c')]
-- search book 1
```

Rekordok

Clean

```
:: Point = {  x       ::  Real
           , y       ::  Real
           , visible ::  Bool
           }
```

```
:: Vector = { dx      ::  Real
           , dy      ::  Real
           }

origo :: Point
origo = { x = 0.0
        , y = 0.0
        , visible = True
        }
```

Haskell

```
data Point
  = Point { x       :: Double
          , y       :: Double
          , visible :: Bool
          }

data VarPoint
  = Point2 { x, y    :: Double }
  | Point3 { x, y, z :: Double }

data Vector
  = Vector { dx      :: Double
           , dy      :: Double
           }

origo :: Point
origo = Point { x = 0
              , y = 0
              , visible = True
              }

-- vagy:
origo = Point 0 0 True
```

Mintaillesztés rekordokra

Clean

```
isVisible :: Point -> Bool
isVisible { visible = True } = True
isVisible _                  = False
```

```
xcoordinate :: Point -> Real
xcoordinate p = p.x

hide :: Point -> Point
hide p = { p & visible = False }

move :: Point Vector -> Point
move p v = { p & x = p.x + v.dx
               , y = p.y + v.dy }
```

Haskell

```
isVisible :: Point -> Bool
isVisible (Point { visible = True }) = True
isVisible _                          = False

-- vagy:
isVisible (Point { visible = v }) = v

-- vagy:
isVisible = visible
```

```
xcoordinate :: Point -> Double
xcoordinate p = x p

hide :: Point -> Point
hide p = p { visible = False }

move :: Point -> Vector -> Point
move p v = p { x = x p + dx v
             , y = y p + dy v }
```

Racionális számok

Clean

```
:: Q = { nom :: Int
       , den :: Int
       }

qZero = { nom = 0, den = 1 }


qOne  = { nom = 1, den = 1 }

simplify { nom = n, den = d }
  | d == 0    = abort "denominator is 0"
  | d < 0     = {nom = ~n/g, den = ~d/g}
  | otherwise = {nom =  n/g, den =  d/g}
 where g = gcd n d

gcd x y = gcdnat (abs x) (abs y)
  where gcdnat x 0 = x
        gcdnat x y = gcdnat y (x mod y)

mkQ n d = simplify { nom = n, den = d }
```

Haskell

```
data Q
  = Q { nom :: Integer
      , den :: Integer
      }

qZero = { nom = 0, den = 1 }
-- vagy: qZero = Q 0 1

qOne  = { nom = 1, den = 1 }

simplify (Q { nom = n, den = d })
  | d == 0    = error "denominator is 0"
  | d <  0    = Q {nom = -n `div` g, den = -d `div` g}
  | otherwise = Q {nom =  n `div` g, den =  d `div` g}
  where g = gcd n d

gcd x y = gcdnat (abs x) (abs y)
  where gcdnat x 0 = x
        gcdnat x y = gcdnat y (x `mod` y)

mkQ n d = simplify (Q { nom = n, den = d })
```

Clean

```
instance * Q  where (*) a b =
  mkQ (a.nom * b.nom) (a.den * b.den)
instance / Q  where (/) a b =
  mkQ (a.nom * b.den) (a.den * b.nom)
instance + Q  where (+) a b =
  mkQ (a.nom * b.den + b.nom * a.den)
      (a.den * b.den)
instance - Q  where (-) a b =
  mkQ (a.nom * b.den - b.nom * a.den)
      (a.den * b.den)
```

```
instance toString Q   where
  toString q =
    toString sq.nom +++ "/" +++
    toString sq.den
      where sq = simplify q
```

Haskell

```
instance Eq Q  where  -- kell a Num elött
  a == b = nom a == nom b && den a == den b
-- vagy: deriving Eq
```

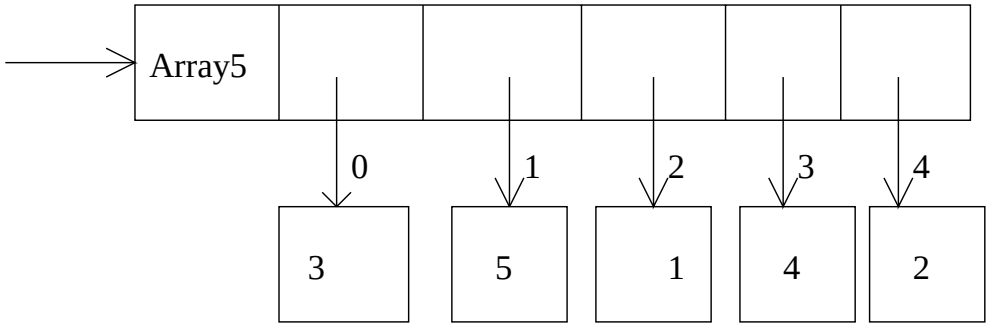
```
instance Num Q  where
  a * b = mkQ (nom a * nom b)
              (den a * den b)
  a + b = mkQ (nom a * den b + nom b * den a)
              (den a * den b)
  a - b = mkQ (nom a * den b - nom b * den a)
              (den a * den b)
  abs = (...)
  signum = (...)
  fromInteger = (...)
```

```
instance Fractional Q  where
  a / b = mkQ (nom a * den b)
              (den a * nom b)
  fromRational = (...)
```

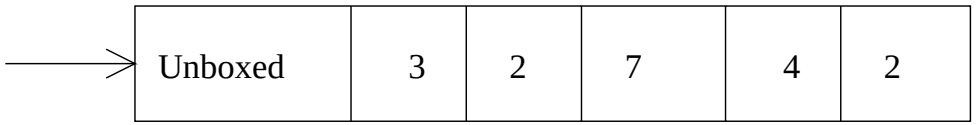
```
instance Show Q  where
  show q = show (nom q) ++ "/" ++ show (den q)
```

Tömbök (Clean)

```
Array5 :: *{Int}
Array5 = { 3, 5, 1, 4, 2 }
```




```
Unboxed :: {#Int}
Unboxed = { 3, 2, 7, 4, 2 }
```



Műveletek tömbökkel (Clean)

Indexelés:

```
Array5.[1] + Unboxed.[0]
```

Tömbabsztrakciók:

```
narray = { e \\ e <- [1, 2, 3] }
nlist  = [ e \\ e <-: Array5 ]
```

Unique tömbök:

```
mArray5 = { Array5 & [3] = 3, [4] = 4 }
mArray  = { Array5 & [i] = k \\ i <- [0..4] & k <-[80, 70..] }
```

5. Algebrai adattípusok

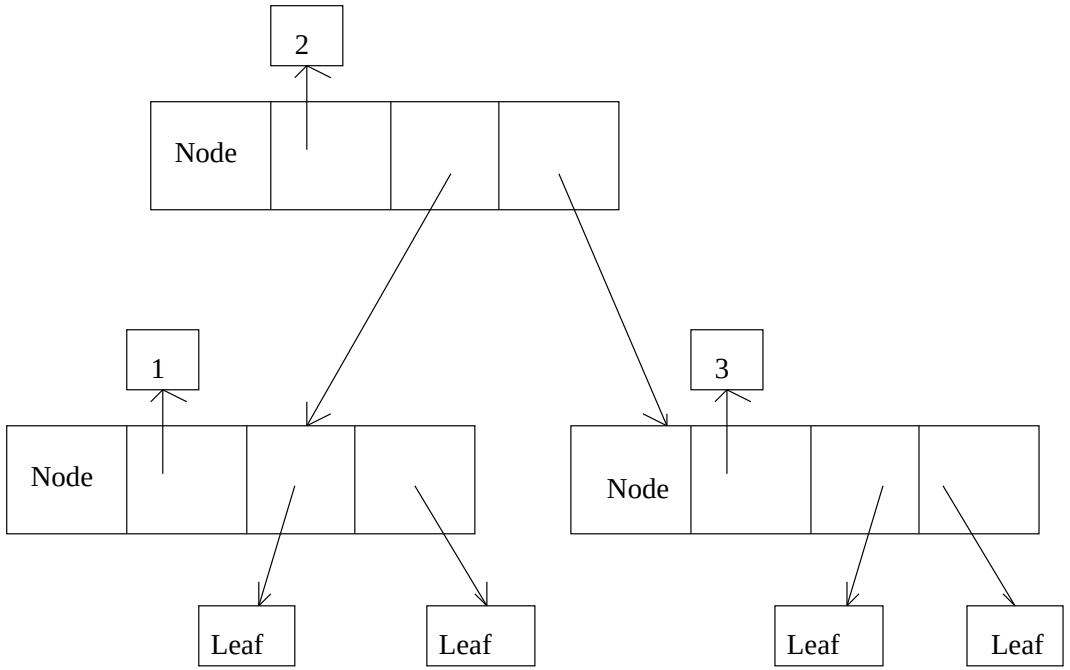
Fák (egyparaméteres fakonstruktor):

Clean

```
:: Tree a = Node a (Tree a) (Tree a)
          | Leaf

aTree = Node 2 (Node 1 Leaf Leaf)
               (Node 3 Leaf Leaf)
```

Haskell

```
data Tree a = Node a (Tree a) (Tree a)
            | Leaf

aTree = Node 2 (Node 1 Leaf Leaf)
               (Node 3 Leaf Leaf)
```



Mintaillesztés adatkonstruktorokra

Clean

```
depth :: (Tree a) -> Int
depth (Node _ l r)
  = (max (depth l) (depth r)) + 1
depth Leaf = 0
```

Haskell

```
depth :: Tree a -> Int
depth (Node _ l r)
  = (depth l `max` depth r) + 1
depth Leaf = 0
```

Maybe: a típus értékkészletének kiterjesztése

```
:: Maybe a = Just a
           | Nothing
```

```
data Maybe a = Just a
             | Nothing
```

Felsorolásos típus
(paraméter nélküli típuskonstruktor, adatkonstruktor):

```
:: Day = Mon | Tue | Wed | Thu
       | Fri | Sat | Sun
```

```
data Day = Mon | Tue | Wed | Thu
         | Fri | Sat | Sun
```

Absztrakt adattípusok

Clean

```
// A verem definíciós modulja
:: Stack a
push :: a (Stack a) -> Stack a
pop  :: (Stack a) -> Stack a
top  :: (Stack a) -> a
empty :: Stack a

// A verem implementációs modulja
:: Stack a :== [a]

push :: a (Stack a) -> Stack a
push e s = [e:s]

pop  :: (Stack a) -> Stack a
pop [e:s] = s

top  :: (Stack a) -> a
top [e:s] = e

empty :: Stack a
empty = []

// Használat:
Start = top (push 1 empty)
```

Haskell

```
module Stack
  ( Stack
  , push
  , pop
  , top
  , empty
  )  where

newtype Stack a = S [a]

push :: a -> Stack a -> Stack a
push e (S s) = S (e:s)

pop :: Stack a -> Stack a
pop (S (e:s)) = S s

top :: Stack a -> a
top (S (e:s)) = e

empty :: Stack a
empty = S []

-- Használat:
main = print $ top (push 1 empty)
```