Laisk väärtustamine

programm: rida deklaratsioone. Näiteks:

```
double x = x + x
main = double (1+1)
```

redex: redutseeritav avaldis – programmis olev avaldis, mida saab lihtsustada. Näiteks: `1+1`

kontekst: kõik definitsioonid, mis kehtivad redex-i asukohas.

Näited ülevaloleva programmi kohta:

- `1+1` puhul on kontekstis kogu programm, kuid neid definitsioone ei lähe vaja.
- järgmisel sammul, kui redexiks `x + x` on selle näite puhul kontekstis ka `x = 2`
- `double (1+1)` puhul on kontekstis kogu programm, kuid vaja läheb vaid `double` definitsiooni.

Laisk väärtustamine

Lihtsustada saab

1. sisseehitatud operaatoreid, aga ainult siis, kui argumendid on juba normaalkujul.
   - `1 + x` ei saa lihtsustada
   - `1 + 1` lihtsustub avaldiseks `2`
2. nimesid, mis on kontekstis defineeritud
   - `x` lihtsustub arvuks `5`, kui kontekstis on `x = 5`
3. argumendi rakendamist lambda-avaldisele
   - `(\x -> x+x) 5` lihtsustub avaldiseks `x+x where x = 5`
4. funktsioonirakendust

   `f` e_1 `...` e_n (kus `f` x_1 `...` x_n `= b`)

   ⇓

   `b where {` x_n `=` e_n`; ...` x_n `=` e_n `}`

   Näiteks: `double (1+1)` lihtsustub avaldiseks `x+x where x = (1+1)`

- Avaldis on *normaalkujul*, kui teda ei saa enam lihtsustada!
- Kui kontekstis on juba muutuja defineeritud, tuleb funktsiooni/lambda argument ümbernimetada!

Redutseerimise järjekord I

Redutseerimiseks valida kõige välimisem avaldis. Kui välist avaldist ei saa redutseerida, võtame ette selle alamavaldised, suunaga vasakult-paremale. (Viivitame `where`-i asendamisega)

Programm:

```
double x = x + x
main = double (1+1)
```

Reduktsioon (nn. laisk väärtustamine, Haskell):

```
main
==> double (1+1)
==> x + x where x = 1+1
==> x + x where x = 2
==> 2 + 2
==> 4
```

Redutseerimise järjekord II

Kui mitte kasutada *where*-konstruktsiooni, tekivad sisse kordused!

Programm:

```
double x = x + x
main = double (1+1)
```

Reduktsioon (nn. normaaljärjekord):

```
main
==> double (1+1)
==> (1+1) + (1+1)
==> 2 + (1+1)
==> 2 + 2
==> 4
```

Redutseerimise järjekord III

Normaalkujul või ühekordselt esinevad argumendid võime kohe asendame funktsiooni! (S.t. ei pea lisama *where*-konstruktsiooni.)

Programm:

```
fact 0 = 1
fact x = x * fact (x-1)
main = fact 2
```

NB! Kui lihtsustamist takistab mustrisobitus, tuleb redutseerida sobitatavat avaldist (nii vähe kui võimalik)!
Reduktsioon:

```
main
==> fact 2
==> 2 * fact (2-1)
==> 2 * fact 1
==> 2 * (1 * fact 0)
==> 2 * (1 * 1)
==> 2 * 1
==> 2
```

Loe lisaks: RWH, lk 32..36

Kõrgemat järku funktsioon

... on funktsioon, mis võtab argumendiks (või tagastab) funktsiooni.

Näiteks

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

Funktsiooni `map` illustreerib järgmine võrdus:

$$\texttt{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

FP keskne mõte: arvutada saab ka arvutustega (e. funktsioonidega).

Näiteks `map` saab interpreteerida kui `(a -> b) -> ([a] -> [b])`

Kõrgemat järku funktsioon II

Näiteks

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

Funktsiooni `foldr` illustreerib järgmine võrdus:

$$\texttt{foldr}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = x_1 + (x_2 + (\ldots + (x_n + b)))$$

Funktsiooni `map` saab defineerida läbi `foldr`-i

```
map f xs = foldr g [] xs
   where g x y = (f x) : y
```

ehk

$$\texttt{foldr}\ g\ []\ [x_1, x_2, \ldots, x_n] = x_1 `g` (x_2 `g` (\ldots `g` (x_n `g` [])))) = f\ x_1 : f\ x_2 : \ldots : f\ x_n : []$$

Kõrgemat järku funktsioon III

Näiteks

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

Funktsiooni `foldl` illustreerib järgmine võrdus:

$$\texttt{foldl}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = (((b + x_1) + x_2) + \ldots) + x_n$$

Funktsiooni `reverse` saab defineerida läbi `foldl`-i

```
reverse xs = foldl g [] xs
  where g x y = y : x
```

ehk

$$\texttt{foldl}\ g\ []\ [x_1, x_2, \ldots, x_n] = (((([]\ `g`\ x_1)\ `g`\ x_2)\ `g`\ \ldots)\ `g`\ x_n = x_n : \ldots : x_2 : x_1 : []$$

Miks eelistada kõrgemat järku funktsioone

- Programmeerijatena ei jõua me kaugele mõeldes *bittide*-tasemel.
- Peame jõudma kõrgemale tasemele — abstraktsemalt

```
f :: [Double] -> [Double]
f (0:_)  = []
f (x:xs) = x + f xs
f []     = []
```

vs.

```
f = sum . takeWhile (/=0)
```

Loe lisaks: RWH, peatükk 4, lk 84..99

Curry stiil ja fun. osaline rakendamine

- Selle asemel, et kirjutada

```
uusFunktsioon x y z = olemasolevFunktsioon (x+1) y z
```

tuleks Haskellis kirjuldada

```
uusFunktsioon x = olemasolevFunktsioon (x+1)
```

  - Pane Tähele: funktsioon `olemasolevFunktsioon` on osaliselt rakendatud. Mõlemad funktsioonid võtavad (vähemalt!) kolm argumenti.

- Realistlikum näide:

```
f xs = sum (takeWhile (/=0) xs)
```

või siis

```
f = sum . takeWhile (/=0)
```

- Soovitus: Kirjuta funktsioone nii, et neid oleks võimalik ka osaliselt rakendada.
  - Funktsiooni `(a,b) -> c` karritatud (*curry*tud) kuju on `a -> b -> c`.

- Saab viia ka liiga kaugele:

```
max3 :: Int -> Int -> Int -> Int
max3  =  (. max) . ((.) . max)
```

Haskelli tüübisüsteem

Haskelli on *staatiliselt tüübitud*, *tugevalt tüübitud*, *tüübituletusega* programmeerimiskeel.

- staatiliselt tüübitud — tüübid teada kompileerimise ajal
- tugevalt tüübitud
  - garantii, et väärtused on just seda tüüpi, millega nad end esitavad
  - tüüpe ei teisendata automaatselt. (nagu C-s int -> float)

S.t. rohkem eeltööd, et tüübivead eemaldada. Programm on aga siis töökindlam.

- tüübituletus — enamasti kirjutatakse tüüpe ainult selleks, et kontrollida, kas kompilaator saab programmist samamoodi aru nagu programmeerija.

Tüübid

- Baastüübid
  - Int, Integer, Char, Double, Float, Bool
- Funktsiooni tüüp
  - Int -> Char, Float -> Float, Int -> (Char -> Int)
- Tüübimuutujad – algavad väiketähega ja tähistavad ükskõik millist konkreetset tüüpi. (polümorfism, i.k. *polymorfism*)
  - a -> a, [a] -> Int, [a] -> [a]

Parameetriline polymorfism: Haskellis ei saa polymorfne funktsioon teha kindlaks, mis konkreetne tüüp tüübimuutujal parajasti on. S.t. funktsioon on defineeritud olenemata konkreetset tüübist.

- Mis funktsioonid võivad olla tüüpidega `a -> a`, `[a] -> Int` või `[a] -> [a]`?

Loe lisaks: RWH, peatükk 2, lk 17..27

Uute tüüpide loomine

Uusi tüüpe saab luua kolmel viisil:

- **data** e. uue algebralise andmetüübi loomine,
- **type** e. tüübi sünonüümi loomine ja
- **newtype** e. olemasolevale tüübile uue ja eristatava nime tegemine.

Näiteks sõned on Haskellis tähtede järejendi sünonüüm!

```
type String = [Char]
```

S.t. Tähtede listid on sõned ja sõned on tähtede listid!

```
['a','X','8'] :: String
"Tere" :: [Char]
```

Olemasolvast tüübist saab teha uue!

```
newtype Sõne = TeeSõne String
```

Nüüd saame kasutada konstruktorit `TeeSõne :: String -> Sõne`, s.t.

```
TeeSõne "Tere" :: Sõne
```

Algebralised andmetüübid

Tõeväärtuste tüüp `Bool` on defineeritud järgnevalt:

```
data Bool = True | False
```

Sellest koodireast loeme välja järgnevat:

- defineeritakse uus andmetüüp `Bool`;
- defineeritakse konstruktor `True :: Bool`;
- defineeritakse konstruktor `False :: Bool`.

Hiljem saab mustrisobitusega vaadata, millise konstruktoriga väärtus loodi:

```
f True  = ...
f False = ...
```

Näide

```
data Tõeväärtus = Tõene | Väär   deriving Show

test1 :: Tõeväärtus
test1 = Tõene

test2 :: Tõeväärtus
test2 = Väär

eitus :: Tõeväärtus -> Tõeväärtus
eitus Tõene = Väär
eitus Väär  = Tõene

conj :: Tõeväärtus -> Tõeväärtus -> Tõeväärtus
conj Tõene Tõene = Tõene
conj _     _     = Väär

disj :: Tõeväärtus -> Tõeväärtus -> Tõ  eväärtus
disj x y = eitus (eitus x `conj` eitus y)
```

Algebraliste andmetüüpide defineerimine

```
data UusTüüp a b = Konstr1 Int | Konstr2 a Char b | Konstr3
```

Ehk siis:

- defineeritakse uued andmetüübid `UusTüüp a b`;
  - näiteks `UusTüüp Int Int`,
  - `UusTüüp Char (UusTüüp Char Int)`;
- defineeritakse konstruktor `Konstr1 :: Int -> UusTüüp a b`;
- defineeritakse konstruktor
  `Konstr2 :: a -> Char -> b -> UusTüüp a b`;
- defineeritakse konstruktor `Konstr3 :: UusTüüp a b`;

Mustrisobitus:

```
f Konstr3  = ...
f (Konstr1 x) = ...
f (Konstr2 x y z) = ...
```

Näide

```
type Radius =  Float
type Width  =  Float
type Height =  Float

data Shape  =  Circle Radius  |  Rect Width Height      deriving Show

area :: Shape -> Float
area (Circle r)  =  pi * r^2
area (Rect w h)  =  w * h

s1 :: Shape
s1 = Circle 1.5

s2 :: Shape
s2 = Rect 0.75 3.0

test :: Float
test = area s1 + area s2
```

Näide

```
data Tree a = Empty  |  Branch a (Tree a) (Tree a)      deriving Show

flatten :: Tree a -> [a]
flatten Empty            = []
flatten (Branch x t1 t2) = flatten t1 ++ [x] ++ flatten t2

puu1 :: Tree Char
puu1 = Branch 'b' (Branch 'a' Empty Empty) (Branch 'c' Empty Empty)

puu2 = Branch 'd' puu1 Empty

puu3 = Branch 'x' puu3 puu3   -- lõpmatu puu

test = flatten puu1
```

Näide: Listid

Listide tüübiperet võime ette kujutada järgnevalt:

```
data [a] = [] | a : [a]
```

Ehk siis:

- Iga tüübi `a` jaoks eksisteerib list `[a]`;
- tühja listi konstruktor `[] :: [a]`;
- mittetühja listi konstruktor `(:) :: a -> [a] -> [a]`.

Näide:

```
list_inf :: [Int]
list_inf = 1 : list_inf
```

Näide: nurjumisvõimalusega funktsioonid

Tüübipere `Maybe` on defineeritud järgnevalt:

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

Listist otsimise funktsioon:

```
lookup x []                      = Nothing
lookup x ((y,z):ys)| x==y        = Just z
                   | otherwise = lookup x ys
```

Näiteks:

- `lookup 4 [(3,'x'),(4,'y')] == Just 'y'`
- `lookup 2 [(3,'x'),(4,'y')] == Nothing`

Tähelepanekud:

- `Maybe a` väärtusi on ühe võrra rohkem kui `a` väärtusi

Loe lisaks: RWH, peatükk 2, lk 41..55..

Kirjetüüp

Selle mustri asemel

```haskell
data Raamat = Raamat
                String   -- pealkiri
                [String] -- autorid
                Int      -- aasta
pealkiri (Raamat p _ _)   = p
autorid  (Raamat  _ as _) = as
aasta    (Raamat _ _ a)   = a

lisaAutor :: Raamat -> String -> Raamat
lisaAutor (Raamat p as a) x = Raamat p (x:as) a
```

saab Haskellis kirjutada ka nii

```haskell
data Raamat = Raamat {
    pealkiri :: String
    autorid :: [String]
    aasta :: Int
  }
lisaAutor raamat x = raamat { autorid = x : autorid r }
```

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`
- `aSeq5 = [10,7..(-3)]`
  - `[10, 7, 4, 1, -2]`
- `aSeq6 = [1..]`
  - `[1, 2, 3, 4, 5, 6, 7, ...` lõpmatu list!
- `aSeq7 = ['A'..'Z']`
  - `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`

Listikomprehensioon

- `lcEx1 = [x*x | x <- [1..5]]`
  - `[1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `lcEx2 = [x*x | x <- [1..10], even x]`
  - `[4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `lcEx3 = [i | (i,c) <- zip [1..] "HaSKell", isUpper c]`
  - `[1, 3, 4]`
- `lcEx4 = [(x,y) | x <- [1..2], y <- [1..3]]`
  - `[(1,1), (1,2), (1,3),(2,1), (2,2), (2,3)]`
- `lcEx5 = [(x,y) | y <- [1..3], x <- [1..2]]`
  - `[(1,1), (2,1), (1,2),(2,2), (1,3), (2,3)]`
  - NB! Parempoolne generaator muutub vasakpoolsest kiiremini!
- `lcEx6 = [(x,y) | x <- [1..4], y <- [x+1..4]]`
  - `[(1,2), (1,3), (1,4),(2,3), (2,4), (3,4)]`

Tüübiklassid I

Haskellis tuleb tihti kirjutada funktsioone, mis erinevad ainult natuke tüübi poolest:

```
equalChar   ::   Char ->   Char -> Bool
equalInt    ::    Int ->    Int -> Bool
equalString :: String -> String -> Bool
```

Sellist koodi *ei saa* kirjutada parameetrilise polymorfse funktsiooniga, kuna funktsioonide implementatsioon on erinev:

```
equal :: a -> a -> Bool
equal = ???  -- pole def. mis töötaks iga tüübi korral
```

Selleks ongi loodud tüübiklassid

```
class Equal a where
  equal :: a -> a -> Bool
instance Equal Char where
  equal = ... -- :: Char -> Char -> Bool
instance Equal Int where
  equal = ... -- :: Int -> Int -> Bool
...
```

Näide

```
class Equal a where
  equal :: a -> a -> Bool

data ValgusFoor = Punane | Kollane | Roheline

instance Equal ValgusFoor where
  equal Punane   Punane   = True
  equal Kollane  Kollane  = True
  equal Roheline Roheline = True
  equal _        _        = False

test :: Bool
test = Kollane `equal` Roheline   -- False
```

Tüübiklassid II

Haskelli standardteegis on defineeritud tüübiklass `Eq`:

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

```
  -- Minimaalne definitioon peab sisaldama:
  -- (==) või (/=)
  x /= y = not (x == y) -- vaikedefinitsioon
  x == y = not (x /= y) -- vaikedefinitsioon
```

Võrldusoperaatori tüüp on:

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

s.t me saame operaatorit kasutada, kui tema argumentitüübil on defineeritud `Eq` instants!

Kõikidel polymorfsetel funktsioonidel, mis kasutavad võrdust peab tüübi kontekst olema `Eq`:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Näide

```
data ValgusFoor = Punane | Kollane | Roheline

instance Eq ValgusFoor where
  Punane   == Punane   = True
  Kollane  == Kollane  = True
  Roheline == Roheline = True
  _        == _        = False

test :: Bool
test = Kollane == Roheline   -- False
```

Tüübiklasside automaatne defineerimine

Osade tüübiklasside definitsioonid on keerukamad, kui tundub, et nad peaks olema

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  GHC.Read.readPrec :: Text.ParserCombinators.ReadPrec.ReadPrec a
  GHC.Read.readListPrec ::  Text.ParserCombinators.ReadPrec.ReadPrec [a]

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
```

... kuigi, neid tüübiklasse kasutatakse suuresti ainult kahe funktsiooni jaoks:

```
read :: Read a => String -> a  -- parsib väärtuse sõnest
show :: Show a => a -> String  -- muudab väärtuse sõneks
```

Standardteegi tüübiklasse nagu `Eq`, `Show`, `Read`, `Ord`, `Enum` saab lasta defineerida automaatselt. Näiteks:

```
data Loom = Kass | Koer | Muu String    deriving (Show, Eq)
```

Loe lisaks: RWH, peatükk 6; LYaH, peatükk 8

Näide

```haskell
data ValgusFoor = Punane | Kollane | Roheline  deriving (Eq)

test :: Bool
test = Kollane == Roheline   -- False
```

Standardsed tüübiklassid

- `Eq`
  - `(==), (/=) :: Eq a => a -> a -> Bool`
- `Ord`
  - `(<=) :: Ord a => a -> a -> Bool`
  - `min, max :: Ord a => a -> a -> a ...`
- `Show`
  - `show :: Show a => a -> String ...`
- `Read`
  - `read :: Read a => String -> a ...`
- `Ix`
  - `range :: (a, a) -> [a]`
  - `index :: (a, a) -> a -> Int ...`

Enum tüübiklass

Enumeratsioone kirjeldab järgnev tüübiklass:

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]                    -- [x ..]      = enumFrom x
  enumFromThen :: a -> a -> [a]           -- [x, y ..]   = enumFromThen x y
  enumFromTo :: a -> a -> [a]             -- [x .. y]    = enumFromTo x y
  enumFromThenTo :: a -> a -> a -> [a]    -- [x, y .. z] = enumFromThenTo x y z

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

- Tüübiklass `Enum` on tuletatav, kui konstruktoritel pole argumente!

  ```
  data Värvid = Punane | Sinine | Kollane   deriving (Enum)
  ```

- `Enum`-id on tihti ka `Bounded`

Abstraktsed andmestruktuurid

- Mitmed senimaani vaadatud andmestruktuurid on implementeeritud Haskellis.
  - `Bool`, `()`, `Maybe a`, `[a]`, `Either a b` jne.
  - Konstruktorid otse kasutatavad.
- Osad on implementeeritd madalamal tasemel, näiteks:
  - `Int`, `Integer`, `Char`, `Float`, `Double`
  - Konstruktorid peidetud — tüübid abstraktsed!
- Vahetevahel on mõistlik luua ise abstraktseid andmestruktuure.
  - Näiteks `Data.Map.Lazy`, `Data.Set`
    - Sellisel puhul ei ekspordita moodulist konstruktorite nimesid
    - … kuid eksporditakse muid funktsioone:

```
empty :: Set a
null  :: Set a -> Bool
singleton :: a -> Set a
insert :: Ord a => a -> Set a -> Set a
delete :: Ord a => a -> Set a -> Set a
foldr  :: (a -> b -> b) -> b -> Set a -> b
...
```

Sisend-väljund Haskellis

- Haskelli puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random :: () -> Int`
- Lahendus: `IO` *monaad*
  - `IO a` tüüpi väärtus — "masin mis arvutab `a` tüüpi väärtuse"
  - `return :: a -> IO a` — masina tagastab esimese argumendi väärtuse
  - `(>>=) :: IO a -> (a -> IO b) -> IO b` — masin käivitab esimese argumendi ja rakendab tulemuse teisele
  - … lisaks baasfunktsioonid nagu `putStrLn :: String -> IO ()` ja `getLine :: IO String`.
- Nii saab kombineerida olemasolevaid `IO` "masinaid". Näiteks:

```
main :: IO ()
main = randomRIO (1, 10) >>= classify >>= putStrLn
  where classify :: Int -> IO String
        classify x | odd x     = return "paaritu"
                   | otherwise = return "paaris"
```

do-süntaks I

Eelneval slaidil olnud koodi on keeruline lugeda ja kirjutada:

```
main :: IO ()
main = randomRIO (1, 10) >>= classify >>= putStrLn
  where classify :: Int -> IO String
        classify x | odd x     = return "paaritu"
                   | otherwise = return "paaris"
```

Sama saab saavutada järgnevalt

```
main :: IO ()
main = do
  r <- randomRIO (1, 10)
  c <- classify r
  putStrLn c
    where classify :: Int -> IO String
          classify x | odd x     = return "paaritu"
                     | otherwise = return "paaris"
```

või

```
main = do
  r <- randomRIO (1, 10)
  if odd r
    then putStrLn "paaris"
    else putStrLn "paaritu"
```

do-süntaks II

Näide

```
proc = do
  s <- getLine
  let n = read s
      n2 = 2*n
  putStrLn ("Kaks korda " ++ s ++ " on " ++ show n2)
```

Do-süntaks algab **do**-võtmesõnaga, millele järgnevad *järjest töödeldavad* laused.

- Laused mustriga x `<-` p, kus p :: `IO` a siis x :: a,
- **let** laused ning
- avaldised e, mille tüüp on `IO a`.

Mitme **do** kasutamine

- **do** seob kokku IO-avaldised, kuid ei saa vaadata konstruktsioonide sisse
- S.t. ühe avaldise jaoks pole do-d vaja
  - `main = putStrLn "Hello World"!`
- Hargenmise puhul võib olla vaja kasutada mitut do-d:

```
main = do
  putStrLn "Kirjuta midagi!"
  xs <- getLine
  if (xs=="")
    then putStr "Sõnakuulmatu!"
    else do
      putStrLn "Tänan!"
      putStrLn ("Kirjutasid: " ++ xs)
```

Loe lisaks: RWH, peatükk 7 (algus)

Reduktsioon IO monaadis

- Üldised reeglid kehtivad aga saab natuke lihtsustada.
- Kui redexiks on avaldis *e* tüübist `IO a`, tuleb kõrvalefekt enda peas teha ja asendada tulemus tagasi avaldisse.
  - Näiteks `putStrLn "Tere!"` trükib välja "Tere!", peale mille tuleb avaldis asendada väärtusega `()`.
- Kuna IO sunnib peale kindla järjestuse võime endi tööd natuke lihtsustada: redutseeritavaid IO avaldisi ei pea iga sammu järel programmi tagasi paigutama. Näiteks

```
printFirst 0 _      = return ()
printFirst n (x:xs) = do  print x
                          printFirst (n-1) xs

main = do  printFirst 3 [1..]
           putStrLn "Kõik!"
```

- Kõigepealt arvutame `printFirst 3 [1..]` ja alles siis pöördume tagasi `main`-i juurde