

Funktsionaalprogrammeerimise meetod

Kaur Alasoo

2009-01-07

1 Sissejuhatus

1.1 Kompileerimine

Lihtne Hello.hs programm:

```
main
  = print (2 + (-3))
```

Selle kompileerimine:

```
ghc --make -o q p
```

1.2 Moodulite importimine

```
import Data.Ratio
main
  = print (1 % 2 - 1 % 6)
```

2 Avaldised

muutuja - väärtustatakse avaldise arvutamise käigus, algab väikese tähega.

konstruktor - lõppväärtust, mida enam teisendada pole vaja. Konstruktor loetakse täisinformatsiooniks. Sellised on näiteks numbritega kirjutatud arvilised konstandid 0, 1, 1.5 jne, tõeväärtused True ja False ning tühja listi märkiv []. Algab suure tähega.

tails - leiab etteantud listi kõik sabad.

inits - leiab etteantud listi kõik pead.

const väärtus on fst väärtuse curried-kuju.

3 Suvalisi funktsioone

```
splitAt 2 (5 : 6 : 7 : [])
```

väärtus on (5 : 6 : [], 7 : [])

replicate 3 5

väärtus on [5,5,5]

Võtame listist teise elemendi:

(1 : 2 : 3 : []) !! 1

selle väärtus on 2

cycle [1, 2, 1]

väärtus on lõpmatu list elementidega 1, 2, 1, 1, 2, 1, 1, 2, 1,

4 Deklaratsioonid

Mõned näidisdeklaratsioonid

```
(x , y)
  = (5 , 0.2)
p : ps
  = 1 : 3 : 4 : []
p : q : qs
  = 1 : 3 : 4 : []
p : ps
  = [1 , 3 , 4]
p : q : qs
  = [1 , 3 , 4]
c : cs
  = "Hei!" — String on tegelikult ka list
[x, y, z]
  = [1 , 3 , 4]
```

4.1 Ehknäidis (*as-pattern*)

```
z@ (x , y)
  = (5 , 0.2) — z v22rtuseks saab paar (5, 0.2)
z@ (x , -)
  = (5 , 0.2) — - t6ttu 0.2 v22rtus eraldi unustatakse 2ra
```

4.2 Laisk näidis

Vaatleme näiteks deklaratsiooni

```
p : q : qs
  = 1 : []
```

Siin näidis ei sobitu, sest alamnäidis q : qs ei sobitu tühja listiga, struktuur ei klapi. Kui aga teha probleeme tekitanud argument laisaks näidiseks, saame deklaratsiooni

```
p : ~(q : qs)
  = 1 : [] — p v22rtuseks saab 1
```

4.3 Lambdaavaldised

Näiteks avaldis $\backslash x \rightarrow x * x$ väärtuseks on funktsioon, mis igal oma argumendil x annab väärtuseks avaldise $x * x$ väärtuse eeldusel, et x väärtus on x , ehk väärtuse x^2 . Öeldes lühidalt, $\backslash x \rightarrow x * x$ väärtus on ruutfunktsioon. Seega on korrektne näiteks rakendamine $(\backslash x \rightarrow x * x)(1 + 2)$, selle avaldise väärtus on 9.

Keerulisema näitena vaatleme avaldise

```
(\ ~(x : xs) -> const 5 x) []
```

väärtustamist. Kuna näidis $(x : xs)$ on laisk, siis ta sobitub tühja listiga. Näidis $x : xs$ seotakse avaldisega $[]$. Avaldis kirjutatakse ümber avaldiseks `const 5 x`. Nüüd kuna `const 5` on laisk, siis muutuja x väärtust vaja ei ole ja väärtustamise lõpptulemuseks on 5.

Lambdaavaldisi saab kasutada ka deklaratsioonides

```
am2
  = \ a b -> (a + b) / 2
am3
  = \ a b c -> (a + b + c) / 3
```

Nende deklaratsioonide olemasolul on muutujad **am2** ja **am3** oma uues tähenduses kasutatavad nii prefiks- kui infiks kujul. Näiteks avaldise `am2 3 5` ja `3 'am2' 5` väärtus on 4, avaldise `am3 1 6 8` ja `(1 'am3' 6) 8` väärtus aga 5.

5 Hargnemiskonstruktsioonid

5.1 IF-THEN-ELSE

Geomeetrilise keskmise arvutamine:

```
gm2
  = \ x y
    -> if x > 0 && y > 0
        then sqrt (x * y)
        else error "gm2: _argumendid_olgu_positiivsed"
```

5.2 Valikuavaldis

Listi esimene element lõppu:

```
uksLoppu
  = \ xs
    -> case xs of
      z : zs
```

```

-> zs ++ [z]
-
-> []

```

Loenda elementide arvu listis:

```

primLoenda
= \ xs
  -> case length xs of
    0
      -> "Null"
    1
      -> "Yks"
    -
      -> "Mitu"

```

Absoluutväärtuse leidmine *compare* abil:

```

absSuurem
= \ x y
  -> case compare (abs x) (abs y) of
    GT
      -> x
    LT
      -> y
    -
      -> error "absSuurem: _absoluutv22rtused_lv6rdsed" .

```

5.3 Funktsioonide deklareerimine

```

sqr x
= x * x
am2 x y
= (x + y) / 2
am3 x y z
= (x + y + z) / 3
—Saab ka infikselt kujul:
x 'am2' y
= (x + y) / 2
(x 'am3' y) z
= (x + y + z) / 3

```

Hargnemiseks saab kasutada ka mitut deklaratsiooni:

```

uksLoppu (z : zs)
= zs ++ [z]
uksLoppu -
= []

```

5.4 Valvurikonstruktsioon

```

arvuKlass x
| x < 0
  = "Negatiivne"
| x < 1
  = "Nullilahedane"
| otherwise
  = "Suur"

```

Sama saab väljendada ka *compare* abil:

```

arvuKlass x
= case compare (floor x) 0 of
  LT
    -> "Negatiivne"
  EQ
    -> "Nullilahedane"
  -
    -> "Suur"

```

6 Suuremad konstruktsioonid

6.1 *let*-avaldis

```

letNaide x
= let
  y = x + 0.5
  a = x * x + 3 * x * y + 2 * y * y
  b = sin x - 3 * sin x * cos y + cos x
in
b / a * 100

```

6.2 *where*-konstruktsioon

```

veerand x
| a > 0 && b > 0
  = "I" ++ v
| a > 0 && b < 0
  = "II" ++ v
| a < 0 && b < 0
  = "III" ++ v
| a < 0 && b > 0
  = "IV" ++ v
| otherwise
  = "vahepeal"
where
  a = sin x
  b = cos x
  v = "_veerandis" .

```

7 Listikomprehensioon

List kõigi naturaalarvude ruutudest:

```
[x * x | x <- [0 .. 9]]  
—Arv koos oma ruuduga  
[(x , x * x) | x <- [1 .. 9]]
```

Funktsioon, mis võtab argumentiks listide listi ja leiab neist kõigi mittetühjade listide pead:

```
pead xss  
  = [x | x : _ <- xss]
```

```
—Samuti saab ka pikkuseid k2tte  
mittetühjadePikkused xss  
  = [length xs | xs@ (_ : _) <- xss] — (.:_) kontrollib mittetyhjust
```

Mitme generaatoriga listikomprehensioon:

```
[(x , y) | x <- [1 .. 9], y <- [1 .. 5]]  
—Teine generaator võib sõltuda ka esimesest  
[(x , y) | x <- [1 .. 9], y <- [x .. 9]]
```

Lisatingimused:

```
[(x , x * x) | x <- [1 .. 9], x /= 5] — J2tame 5-e v2lja  
[(x , y) | x <- [1 .. 9], y <- [1 .. 9], x <= y]
```

8 Monaadikomprehensioon

do

```
putStrLn "Tere, _Haskell!"  
putChar ' '  
putStr "2_+_(-3)_=_"  
print (2 + (-3))
```

main

```
= do  
  cs <- getLine — loe üks rida  
  c <- getChar — loe üks m2rk  
  putStrLn ("\nKokku_" ++ cs ++ c : ".")
```

—Loeme symboli ja v2ljastame selle koodi

main

```
= do  
  hSetBuffering stdin NoBuffering  
  c <- getChar  
  print (ord c)
```

—*Kahe arvu summa:*

```
kysiSumma
  = do
    s <- getLine
    t <- getLine
    return (read s + read t :: Integer)

main
  = do
    putStrLn "Anna_kaks_taisarvu."
    sum <- kysiSumma
    putStrLn ("Summa_\_\_" ++ show sum ++ "_\_.")
```

9 Rekursioon

suurSumma - funktsioon, mis võtab argumendiks naturaalarvu n ja annab tulemuseks arvu $1^4 + \dots + n^4$ ehk 1-st n -ni kõigi täisarvude 4. astmete summa.

```
suurSumma
  :: (Integral a)
  => a -> a
suurSumma n
  = case compare n 0 of
    GT
      -> suurSumma (n - 1) + n ^ 4
    EQ
      -> 0
    -
      -> error "suurSumma: negatiivne liidetavate arv".
```

Kahanev faktoriaal:

```
kahFact
  :: (Num a, Integral b)
  => a -> b -> a
kahFact x k
  = case compare k 0 of
    GT
      -> x * kahFact (x - 1) (k - 1)
    EQ
      -> 1
    -
      -> error "kahFact: negatiivne teine argument"
```

Fibonacci arvud:

```
fib n
  | n >= 0
  = let
      fib 0
        = (0, 1)
      fib n
```

```

      = let
          (u , v)
            = fib (n - 1)
        in
          (v , u + v)
    in
      fst (fib n)
  | otherwise
    = (if odd n then 1 else -1) * fib (-n). (67)

```

—Nullide arv:

```

nullideArv
  :: (Num a)
  => [a] -> Int
nullideArv (x : xs)
  = let
      ulejaanu
        = nullideArv xs
    in
      if x == 0 then 1 + ulejaanu else ulejaanu
nullideArv -
  = 0

```

—Eemalda listist mingi elemendi esimene esinemine

```

eemaldaEsimene
  :: (Eq a)
  => a -> [a] -> (Bool , [a])
eemaldaEsimene a (x : xs)
  | a == x
    = (True , xs)
  | otherwise
    = let
        (tv , us)
          = eemaldaEsimene a xs
      in
        (tv , x : us)
eemaldaEsimene - -
  = (False , [])

```

—Listi kahe järjestikuse elemendi summad:

```

summad
  :: (Num a)
  => [a] -> [a]
summad (x : xs@ (y : -))
  = x + y : summad xs
summad -
  = []

```

—Loeme listi yle yhe kaheks

```

kaheksLoe
  :: [a] -> ([a] , [a])

```

```

kaheksLoe (x : y : ys)
  = let
      (us , vs)
        = kaheksLoe ys
    in
      (x : us , y : vs)
kaheksLoe xs
  = (xs , [])

```

9.1 Rekursiivselt defineeritud protseduurid

```

kuulaKuni
  :: Char -> IO ()
kuulaKuni a
  = do
      c <- getChar
      if c == a then return () else kuulaKuni a

```

```

loendaRead
  :: (Integral a)
  => IO a
loendaRead
  = do
      rida <- getLine
      if null rida
        then return 0
        else do
            ridadeArv <- loendaRead
            return (1 + ridadeArv)

```

9.2 Rekursiivsed andmestruktuurid

— *Geomeetriline jada:*

```

geom a q
  = a : geom (a * q) q

```

— *Saab ka nii:*

```

geom a q
  = let
      gs
        = a : [x * q | x <- gs]
    in
      gs

```

— *Põimi kaks listi kasvavas järjekorras:*

```

põimi
  :: (Ord a)
  => [a] -> [a] -> [a]
põimi xs@ (a : as) ys@ (b : bs)
  = case compare a b of

```

```

LT
-> a : poimi as ys
GT
-> b : poimi xs bs
-
-> a : poimi as bs
poimi xs []
  = xs
poimi - ys
  = ys

```

9.3 Arvutamine rekursiivse funktsiooni argumentidel

—*Nullide arvu leidmine loenduriga*

```

nullideArv xs
  = let
      arv n (z : zs)
        = arv (if z == 0 then n + 1 else n) zs
      arv n -
        = n
    in
      arv 0 xs

```

—*Sulgude tasakaal*

```

bilanss
  :: String -> Bool
bilanss str
  = let
      bilanss n (c : cs)
        = let
            d = case c of
                '(' -> 1
                ')',
                '-' -> -1
                - -> 0
          in
            n >= 0 && bilanss (n + d) cs
        bilanss n -
          = n == 0
    in
      bilanss 0 str

```

—*Kaheks lugemine j2rjehoidjaga*

```

kaheksLoe xs
  = let
      kaheksLoe b (x : xs)
        = let
            (us , vs)
              = kaheksLoe (not b) xs

```

```

                in
                if b then (us , x : vs) else (x : us , vs)
kaheksLoe - -
    = ([] , [])
in
kaheksLoe False xs

```

{-v6tab argumendiks listi ja annab tulemuseks tema mittetyhjade aligusjuppide arvu, mis annavad elementide summaks nulli. -}

```

nullsummadeArv xs
    = let
        arv s n (z : zs)
            = let
                s'
                    = s + z
                in
                arv s' (if s' == 0 then n + 1 else n) zs
        arv - n -
            = n
    in
        arv 0 0 xs

```

{-v6tab argumendiks listi ja annab tulemuseks tema maksimaalsete elementide arvu-}

```

maxArv
    :: (Ord a)
    => [a] -> Int
maxArv (x : xs)
    = let
        arv m i (z : zs)
            = case compare m z of
                GT
                    -> arv m i zs
                EQ
                    -> arv m (i + 1) zs
                -
                    -> arv z 1 zs
        arv - i -
            = i
    in
        arv x 1 xs
maxArv -
    = 0

```

—suuredSummad akumulaatoriga

```

suuredSummad
    :: (Integral a)
    => [a]
suuredSummad
    = let
        suuredSummad a i
            = a : suuredSummad (a + i ^ 4) (i + 1)

```

```

    in
    suuredSummad 0 1

—Fibonacci arvud akumulaatoriga
fibs
  = let
    fibs a b
      = a : fibs b (a + b)
    in
    fibs 0 1

```

9.4 Jaga ja valitse

```

—Ruutjuure leidmine
taisruut
  :: Integer -> Bool
taisruut n
  | n > 1
    = l2henda n (algpiirid n)
  | otherwise
    = n >= 0(113)

l2henda
  :: Integer -> (Integer , Integer) -> Bool
l2henda n (a , b)
  | b - a <= 1
    = sqr a == n
  | otherwise
    = let
      m = (a + b) 'div' 2
      in
      case compare (sqr m) n of
      LT
        -> l2henda n (m , b)
      GT
        -> l2henda n (a , m)
      -
        -> True.

algpiirid n
  = let
    piirid x
      | sqr y > n
        = (x , y)
      | otherwise
        = piirid y
    where
      y = x + x
    in
    piirid 1

```

—Jaga list esimese elemendi j2rgi kaheks

```

kaheksLahuta
  :: (Ord a)
  => a -> [a] -> ([a] , [a])
kaheksLahuta x (z : zs)
  = let
      (us , vs)
        = kaheksLahuta x zs
      in
      if z < x then (z : us , vs) else (us , z : vs)
kaheksLahuta _ _
  = ([] , [])

```

10 Kõrgemat järku funktsioonid

Muutuja **flip** väärtuseks on funktsioon, mis võtab argumentiks curried-kujulise funktsiooni f ja annab tulemuseks samuti curried-kujulise funktsiooni, mis töötab nagu f , kuid võtab kaks esimest argumenti vastupidises järjekorras.

Muutujate **curry** ja **uncurry** väärtuseks on teisendused funktsioonide curried-kuju ja järjendiargumentidega kuju vahel.

Näiteks `uncurry (+)` väärtus on funktsioon, mis võtab argumentiks arvupaari ja annab tulemuseks selle paari komponentide summa. Niisiis `uncurry (+) (2, 3)` väärtus on 5, sest `(+) 2 3` väärtus on 5. Avaldise `uncurry (flip div)` väärtus on aga funktsioon, mis võtab argumentiks täisarvupaari ja annab tulemuseks teise komponendi jagatise esimesega. Niisiis väärtustades `uncurry (flip div) (3, 17)`, saame tulemuseks 5.

Kompositsioon. . Näiteks avaldise `(+ 2) . (* 5)` väärtuseks on funktsioon, mis igal oma argumentil arvutab väärtuse, korrutades argumenti 5-ga ja liites saadud tulemusele 2. Niisiis avaldise `((+ 2) . (* 5)) 7` väärtus on 37, eelnevas vaadeldud avaldis `uncurry (flip div)` on aga samaväärselt operaatoriga `.` ümber kirjutatav kujul `(uncurry . flip) div`.

Muutuja **iterate** väärtuseks on kõrgemat järku funktsioon, mis võtab argumentiks sellise funktsiooni f , mille argumenti- ja väärtusetüüp on üks ja sama, ja annab tulemuseks funktsiooni, mis võtab sama tüüpi väärtuse x argumentiks ja annab välja lõpmatu listi, mis koosneb väärtustest $x, f x, f (f x)$ jne.

Näiteks avaldise `iterate (* 2) 1` väärtuseks on lõpmatu list kõigi 2 astmetega, st `1 : 2 : 4 : 8 : 16 : . . .`

```

geom a q
  = iterate (* q) a

```

until võtab järjest argumentideks predikaadi p , selle predikaadi argument-tüübist samasse tüüpi töötava funktsiooni f ning sama tüüpi argumenti x ning annab välja esimese väärtuse jadas $x, f x, f (f x) . . .$, mis rahuldab predikaati p , kui seal selline leidub, vastasel korral jääb lõpmatusse tsüklisse.

Näiteks avaldise `until (> 100) (* 2) 1` väärtus on 128, sest 128 on esimene arvu 2 aste, mis on suurem 100-st.

Avaldise `takeWhile (<= 100) (iterate (* 2) 1)` väärtus on `1 : 2 : 4 : 8 : 16 : 32 : 64 : []`, sest need elemendid 2 astmete listi alguses on 100-st väiksemad, järgmine element 128 aga pole. Avaldise `takeWhile (> 100) (iterate (* 2) 1)` väärtus on aga tühi list, sest juba esimene element 1 pole suurem 100-st, st predikaati ei rahulda; see, et kuskil tagapool on ka predikaati rahuldavaid elemente, enam määravaks ei saa.

— *Yks huivtav viis, kuidas predikaati eitada*

```

import Data.Char

```

```
ilmaVaikestetaAlgus a
  = takeWhile (not.isLower) a
```

Muutuja **dropWhile** väärtuseks on funktsioon, mis võtab samad argumentid nagu eelmine, kuid annab neil tulemuseks argumentlisti selle osa, mis takeWhile puhul üle jääb. Niisiis avaldise dropWhile (<= 100) (iterate (* 2) 1) väärtus on lõpmatu list 128 : 256 : 512 : . . ., dropWhile (> 100) (iterate (* 2) 1) väärtus aga kogu 2 astmete list.

Avaldise **filter** (> 100) (iterate (* 2) 1) väärtus on lõpmatu list 128 : 256 : 512 : . . ., sest predikaadi mitterahuldamine esimeste elementide poolt ei sunni otsimist lõpetama. Avaldise filter isUpper "Tere, Haskell!" väärtus on "TH"; võrdluseks takeWhile isUpper "Tere, Haskell!" annab väärtuseks "T" ja dropWhile isUpper "Tere, Haskell!" annab "ere, Haskell!".

span (<= 100) (iterate (* 2) 1) väärtuseks on paar listidega 1 : 2 : 4 : 8 : 16 : 32 : 64 : [] ja 128 : 256 : 512 :

Lisaks võiks mainida muutujaid **all** ja **any**, mis ka võtavad samasugused argumentid nagu takeWhile, dropWhile ja filter, kuid nemad annavad tulemuseks tõeväärtuse: all puhul tuleb True siis, kui listi iga element rahuldab predikaati ja False, kui mõni ei rahulda; any puhul tuleb True siis, kui listi mõni element rahuldab predikaati ja False, kui ükski ei rahulda. Väärtustamisel vaadatakse listi alates algusest senikaua, kuni leitakse element, mis vastavalt kas predikaati ei rahulda (all puhul) või rahuldab (any puhul).

map rakendab listi igale elemendile mingit üht ja sama funktsiooni.

Operaatori map analoog, mis tegutseb korraga kahel listil, on **zipWith**. Tema väärtuseks on funktsioon, mis võtab järjest argumentideks mingi kahe argumentiga curried-kujulise funktsiooni ja kaks listi, tulemuseks aga annab ühe listi, mille elemendid on saadud argumentlistide vastavate elementide kokkuarvutamisel selle funktsiooniga. Kui üks list on teisest pikem, siis ülejääv osa unustatakse lihtsalt ära. Näiteks zipWith (*) [2, 3] [5, 7, 11] väärtus on 10 : 21 : [].

```
summad xs
  = zipWith (+) xs (tail xs)
```

Avaldise **foldl** (+) 0 [1, 2, 3] väärtus on 6, sest 0 + 1 + 2 + 3 = 6. Avaldise foldl (-) 0 [1, 2, 3] väärtus on -6, sest nüüd tuleb arvutada 0 - 1 - 2 - 3. Avaldise foldl (++) [] [[1, 3], [7], []] väärtus on analoogselt 1 : 3 : 7 : [].

Avaldise **foldr** (+) 0 [1, 2, 3] väärtus on 6, sest 1 + 2 + 3 + 0 = 6, st tulemus on sama mis samade argumentidega foldl puhul. Samas avaldise foldr (-) 0 [1, 2, 3] väärtus on mitte -6 nagu foldl puhul, vaid hoopis 2, sest arvutatakse 1 - (2 - (3 - 0)) = 1 - (2 - 3) = 1 - (-1) = 2.

```
suurSumma n
  | n >= 0
    = foldl (\ a i -> a + i ^ 4) 0 [1 .. n]
  | otherwise
    = error "suurSumma: negatiivne liidetavate arv"
```

Kõrgemat järgu funktsioone saab ka ise defineerida, näiteks:

```
korgem f
  = f 1
```

```
mapIgaTeine f (x : y : ys)
  = x : f y : mapIgaTeine f ys
```

```
mapIgaTeine - xs
= xs
```

11 Algebraised tüübid

```
data Kuu
= Jaanuar
| Veebruar
| Marts
| Aprill
| Mai
| Juuni
| Juuli
| August
| September
| Oktoober
| November
| Detsember
deriving (Show, Eq, Ord, Enum)
```

```
suvekuud
  :: [Kuu]
suvekuud
  = [Juuni, Juuli, August]
```

```
suvekuud
  = [Juuni .. August] — sest on olemas deriving(Enum)
```

```
paevi (y, m)
| elem jrk [1, 3, 5, 7, 8, 10, 12]
  = 31
| elem jrk [4, 6, 9, 11]
  = 30
| y `mod` 400 == 0
  = 29
| y `mod` 100 == 0
  = 28
| y `mod` 4 == 0
  = 29
| otherwise
  = 28
where
  jrk
    = fromEnum m + 1
```

```
jargmKuu
  :: (Integer, Kuu) -> (Integer, Kuu)
jargmKuu (y, m)
```

```

= case m of
  Detsember
    -> (succ y , Jaanuar)
  -
    -> (y , succ m)

```

```

data Daatum
  = Daatum Integer Kuu Int
  deriving (Show, Eq, Ord)

```

```

jargmPaev
  :: Daatum -> Daatum
jargmPaev (Daatum y m d)
  | paevi (y , m) > d
  = Daatum y m (d + 1)
  | otherwise
  = let
      (y' , m')
      = jargmKuu (y , m)
    in
    Daatum y' m' 1

```

—*Argumendiga konstruktor*

```

data Kalendripaev
  = Gregooriuse Daatum
  | Juuliuse Daatum
  deriving (Show)

```

—*Defineerime tüübipere*

```

data Eba d
  = Teadmata
  | Piirid d d
  | Valik [d]

```

Viimane kood defineerib ühe parameetriga tüübipere Eba. Konstruktori Piirid tüüp on $d \rightarrow d \rightarrow Eba\ d$, konstruktori Valik tüüp $[d] \rightarrow Eba\ d$ ja konstruktori Teadmata tüüp lihtsalt $Eba\ d$. Nüüd saame kirjutada näiteks avaldise $Piirid\ 0\ 5$, mille tüübiks võib kirjutada $Eba\ Integer$ või üldisemalt $(Num\ d) \Rightarrow Eba\ d$, avaldise $Piirid\ (-3.5)\ 8.9$, mille tüübiks $Eba\ Double$ või üldisemalt $(Floating\ a) \Rightarrow Eba\ d$, avaldise $Valik\ "Aa"$ tüübiga $Eba\ Char$ jne. Tüüp $Eba\ Daatum$ on aga sisuliselt sama mis endine $EbaDaatum$.

Rekursiivsed tüübid

```

data Joru a
  = Üks a
  | Mitu a (Joru a)
  deriving (Show) . (207)

```

Selle definitsiooni puhul on võimalikud avaldised tüübist $Joru\ Int$ näiteks $Üks\ 0$, $Üks\ 1$, $Mitu\ 1\ (Üks\ 2)$, $Mitu\ 3\ (Mitu\ 5\ (Üks\ 15))$ jne, mis kõik esitavad struktuuri, kus vähemalt üks komponent.

```

joruPikkus
  :: Joru a -> Int
joruPikkus (Mitu _ xj)

```

```

    = 1 + joruPikkus xj
joruPikkus -
    = 1

```

```

joruSumma
  :: (Num a)
  => Joru a -> a
joruSumma (Mitu x xj)
  = x + joruSumma xj
joruSumma (Uks x)
  = x

```

—*Kahendpuu (andmed kbigis tippudes)*

```

data Kahend a
  = Tuhi
  | Tipp a (Kahend a) (Kahend a)

```

—*Kahendpuu (andmed ainult lehtedes)*

```

data Kahend' a
  = Leht a
  | Harud (Kahend' a) (Kahend' a)(209)

```

—*Tippude arvu arvutamine*

```

kahendSuurus
  :: Kahend a -> Int
kahendSuurus (Tipp - ut vt)
  = 1 + kahendSuurus ut + kahendSuurus vt
kahendSuurus -
  = 0

```

—*Puu summa leidmine*

```

kahendSumma
  :: (Num a)
  => Kahend a -> a
kahendSumma (Tipp x ut vt)
  = x + kahendSumma ut + kahendSumma vt
kahendSumma -
  = 0

```

—*Kui andmed on ainult lehttippudes*

```

kahend'Summa
  :: (Num a)
  => Kahend' a -> a
kahend'Summa (Harud ut vt)
  = kahend'Summa ut + kahend'Summa vt
kahend'Summa (Leht x)
  = x

```