

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:

$$\text{map } f [x_1, x_2, \dots, x_n] = [f x_1, f x_2, \dots, f x_n]$$

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

Näiteks map saab interpreteerida kui $(a \rightarrow b) \rightarrow ([a] \rightarrow [b])$

Näide

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

```
map (+1) [1,2,3] = map (+1) (1:2:3:[])
  ~> (1+1) : map (+1) (2:3:[])
  ~> 2 : map (+1) (2:3:[])
  ~> 2 : (2+1) : map (+1) (3:[])
  ~> 2 : 3 : map (+1) (3:[])
  ~> 2 : 3 : (3+1) : map (+1) []
  ~> 2 : 3 : 4 : map (+1) []
  ~> 2 : 3 : 4 : []
  = [2, 3, 4]
```

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:

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

Funktsiooni map saab defineerida läbi foldr-i

```

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

```

ehk

$$\text{foldr } g \ [] \ [x_1, x_2, \dots, x_n] = x_1 'g' (x_2 'g' (\dots 'g' (x_n 'g' []))) = f \ x_1 : f \ x_2 : \dots : f \ x_n : []$$

Näide

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

```
foldr (+) 0 (1:2:3:[]) ~> (+) 1 (foldr (+) 0 (2:3:[]))
                       ~> (+) 1 ((+) 2 (foldr (+) 0 (3:[])))
                       ~> (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
                       ~> (+) 1 ((+) 2 ((+) 3 0))
                       ~> (+) 1 ((+) 2 3)
                       ~> (+) 1 5
                       ~> 6
```

Näide

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

```
head xs = foldr (\ a b -> a) undefined xs
```

```
head (3:4:5:[]) ~> foldr (\a b -> a) undefined (3:4:5:[])
                ~> (\a b -> a) 3 (foldr (\a b -> a) undefined (4:5:[]))
                ~> (\b -> 3) (foldr (\a b -> a) undefined (4:5:[]))
                ~> 3
```

Kõrgemat järku funktsioon III

Näiteks

```

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

Funktsiooni `foldl` illustreerib järgmine võrdus:

$$\text{foldl } (+) \mathbf{b} [x_1, x_2, \dots, x_n] = (((\mathbf{b} + x_1) + x_2) + \dots) + x_n$$

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

```

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

ehk

$$\text{foldl } g [] [x_1, x_2, \dots, x_n] = ((([] \text{ 'g' } x_1) \text{ 'g' } x_2) \text{ 'g' } \dots) \text{ 'g' } x_n = x_n : \dots : x_2 : x_1 : []$$

Näide

```

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

```

```

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

```

```

reverse (1:2:3:[]) = foldl g [] (1:2:3:[])
  ~> foldl g (g [] 1) (2:3:[])
  ~> foldl g (g (g [] 1) 2) (3:[])
  ~> foldl g (g (g (g [] 1) 2) 3) []
  ~> g (g (g [] 1) 2) 3
  ~> 3 : (g (g [] 1) 2)
  ~> 3 : 2 : (g [] 1)
  ~> 3 : 2 : 1 : []

```

Näide

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

```
last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ~> foldl g [] (3:4:5:[])
               ~> foldl g (g [] 3) (4:5:[])
               ~> foldl g (g (g [] 3) 4) (5:[])
               ~> foldl g (g (g (g [] 3) 4) 5) []
               ~> g (g (g [] 3) 4) 5
               ~> 5
```


Kõrgemat järku funktsioonid kui disainimustrid

- foldr on listi itereerimise muster.

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

vs.

```
f = foldr (+) 0
```

- Lihtsate funktsioonide puhul on ka rekursiivne lahendus selge.
- Pikema ülesande puhul on hea eraldada listi töötlemine.

Miks eelistada kõrgemat järku funktsioone

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

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

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 kirjudada

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

parameetriline polümorfism: Haskellis ei saa polümorfne 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 -> Bool` või `a -> b -> a`?

Loe lisaks: RWH, peatükk 2, lk 17..27

Mitu erinevat *puhast* väärtust?

- $|\text{Bool}| = 2$: True, False
- $|\text{Int}| \geq 2^{30}$: $-2^{29}, \dots, 2^{29} - 1$
- $|\text{Integer}| = |\mathbb{N}|$: $0, 1, -1, \dots$
- $|\text{Char}| = 2^{16}$: 'a', 'b', 'c', ...

- $|\alpha \rightarrow \beta| = \beta^\alpha$

- Näiteks $|\text{Bool} \rightarrow \text{Bool}| = 2^2 = 4$
 1. {True \rightarrow True, False \rightarrow True}
 2. {True \rightarrow False, False \rightarrow False}
 3. {True \rightarrow True, False \rightarrow False}
 4. {True \rightarrow False, False \rightarrow True}

Mitu erinevat Haskell'i väärtust?

- iga väärtus saab olla undefined, mis lõpetab programmi töö
- ei saa kontrollida, kas mingi väärtus on undefined
- ... sellisel juhul on tagastusväärtus undefined

- Näiteks Bool \rightarrow Bool:

1.	{ True \rightarrow True,	False \rightarrow True,	undefined \rightarrow undefined }
2.	{ True \rightarrow False,	False \rightarrow False,	undefined \rightarrow undefined }
3.	{ True \rightarrow True,	False \rightarrow False,	undefined \rightarrow undefined }
4.	{ True \rightarrow False,	False \rightarrow True,	undefined \rightarrow undefined }
5.	{ True \rightarrow undefined,	False \rightarrow True,	undefined \rightarrow undefined }
6.	{ True \rightarrow undefined,	False \rightarrow False,	undefined \rightarrow undefined }
7.	{ True \rightarrow True,	False \rightarrow undefined,	undefined \rightarrow undefined }
8.	{ True \rightarrow False,	False \rightarrow undefined,	undefined \rightarrow undefined }
9.	{ True \rightarrow undefined,	False \rightarrow undefined,	undefined \rightarrow undefined }
10.	{ True \rightarrow True,	False \rightarrow True,	undefined \rightarrow True }
11.	{ True \rightarrow False,	False \rightarrow False,	undefined \rightarrow False }

- $|\alpha \rightarrow \beta|_H = (|\beta| + \mathbf{1})^{|\alpha|} + |\beta| + \dots$

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 järjesti sünonüüm!

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

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

```
['a', 'X', '8'] :: String
```

```
"Tere" :: [Char]
```

Olemasolevast 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 mustrisobitusiga 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 -> Tõeväärtus
disj x y = eitus (eitus x `conj` eitus y)
```

Algebraalsete 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
```

newtype vs data

```
data MyString1 = MyString1 String
newtype MyString2 = MyString2 String
```

- $|MyString1| = |String| + 1$
- $MyString1\ undefined \neq undefined$
- $|MyString2| = |String|$
- $MyString2\ undefined == undefined$

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 $(:) :: a \rightarrow [a] \rightarrow [a]$.

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..

Näide: Tüüpide summa

Tüübid **Either** a b on defineeritud järgnevalt:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Kasutatakse näiteks siis kui on vaja edastada veateadet:

```
lookup' x [] = Left "Elementi ei leitud!"
lookup' x ((y,z):ys) | x==y = Right z
                      | otherwise = lookup' x ys
```

Näiteks:

- `lookup' 4 [(3, 'x'), (4, 'y')] == Right 'y'`
- `lookup' 2 [(3, 'x'), (4, 'y')] == Left "Elementi ei leitud!"`

Tähelepanekud:

- **Either** a b väärtusi on sama palju kui a väärtusi pluss b väärtusi.