Redutseerimise järjekord I

Redutseerimiseks valida kõige välimisem avaldis. Kui välist avaldist ei saa redutseerida, võtame ette selle alamavaldised, suunaga vasakult-paremale.

Programm:

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

Reduktsioon:

$$
\begin{aligned}
\texttt{main} &\rightsquigarrow \texttt{double (1+1)} \\
&\rightsquigarrow \texttt{(1+1) + (1+1)} \\
&\rightsquigarrow \texttt{2 + (1+1)} \\
&\rightsquigarrow \texttt{2 + 2} \\
&\rightsquigarrow \texttt{4}
\end{aligned}
$$

Redutseerimise järjekord II

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

- Kui lihtsustamist takistab mustrisobitus, redutseerime sobitatavat avaldist seni, kuni saame otsustada, milline juht valida.

  Reduktsioon:

$$
\begin{aligned}
\texttt{fact 2} &\rightsquigarrow \texttt{2 * fact (2-1)} \\
&\rightsquigarrow \texttt{2 * fact 1} \\
&\rightsquigarrow \texttt{2 * (1 * fact (1-1))} \\
&\rightsquigarrow \texttt{2 * (1 * fact 0)} \\
&\rightsquigarrow \texttt{2 * (1 * 1)} \\
&\rightsquigarrow \texttt{2 * 1} \\
&\rightsquigarrow \texttt{2}
\end{aligned}
$$

- Formaalselt on järjekord defineeritud funktsiooniga fookus ja kaalutlus. (konspektist)

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])`

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:

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

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

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

ehk

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

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:

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

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

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

ehk

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

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
```

$$
\begin{aligned}
\texttt{reverse}\ (1{:}2{:}3{:}[]) &= \texttt{foldl g [] (1:2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g [] 1) (2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g (g [] 1) 2) (3:[])} \\
&\rightsquigarrow \texttt{foldl g (g (g (g [] 1) 2) 3) []} \\
&\rightsquigarrow \texttt{g (g (g [] 1) 2) 3} \\
&\rightsquigarrow \texttt{3 : (g (g [] 1) 2)} \\
&\rightsquigarrow \texttt{3 : 2 : (g [] 1)} \\
&\rightsquigarrow \texttt{3 : 2 : 1 : []}
\end{aligned}
$$

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 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` 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

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 -> Tõeväärtus
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
```