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 = ...
```

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

- pindala:

```
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
```

- puud:

```
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
```

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: 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
...
```

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
```

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