Kirjetüüp

Selle mustri asemel

```
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

```
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

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `[i | (i,c) <- zip [1..] "HaSKell", isUpper c] = [1, 3, 4]`
- ```
  [(x,y) | x <- [1..2], y <- [1..3]] =
      [(1,1), (1,2), (1,3),(2,1), (2,2), (2,3)]
  ```
- ```
  [(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!
- ```
  [(x,y) | x <- [1..4], y <- [x+1..4]] =
      [(1,2), (1,3), (1,4),(2,3), (2,4), (3,4)]
  ```
- `[y*z | x <- [1..4], let z = x+1, y <- [z..4]] = [4, 6, 8, 9, 12, 16]`
  ```

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 polümorfse 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 polümorfsetel 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

```
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)
```

- do-süntaks on lihtne, kuid vajab harjutamist!

do tähendus

- `a >>= f` on sama mis

```
do x <- a
   f x
```

- `a >> b` on sama mis

```
do a
   b
```

-

```
do a                    do do a
   b   on sama mis             b
   c                        c
```

- Fixity:

```
infixl 1 >>
infixl 1 >>=
```

Näide

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

on sama mis

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

Näide

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

on sama mis

```
main =
   putStrLn "Kirjuta midagi!" >>  getLine  >>= ifThenElse
     where ifThenElse xs = if (xs=="") then case1 else case2 xs
           case1         = putStr "Sõnakuulmatu!"
           case2 xs      = 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

Monaadidest üldisemalt

Haskellis in monaad on ühe muutujaga tüübipere, mille jaoks on defineeritud järgnevad funktsioonid:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Intuitsioon: Tüüp `m a` on nagu konteiner, kuhu saab `a` tüüpi väärtust hoida. (Aga igal `m a` ei pruugi sisaldada `a` tüüpi väärtust.)

- Tahame vältida eksplitsiitselt väärtuse konteinerist välja võtmist.

Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Võrrandid:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Intuitsioon: igasugune konteiner `f a`, kus `a` väärtus ei interakteeru konteineriga

Näide: listid, `Maybe`

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Võrrandid:

```
                pure id <*> v == v
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)
          pure f <*> pure x == pure (f x)
               u <*> pure y == pure ($ y) <*> u
```

Intuitsioon: saame väärtusi konteinerisse panna + võime konteineri sees funktsioone rakendada

Näide: Pilveserver (AWS)

Monad

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Võrrandid:

```
return a >>= k  ==  k a
  m >>= return  ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Intuitsioon: järjest rakendatavad masinad

Näide: IO

Maybe monaad

```
data Maybe a = Nothing | Just a deriving (Eq, Ord)
```

- intuitsioon: `Nothing` – viga, nurjumine
- definitsioon

```
return x  =  Just x

(Just a) >>= f  =  f a
Nothing  >>= f  =  Nothing
```

- Näide (pseudokood):

```
getTaxOwed name = do
  number       <- lookup name phonebook
  registration <- lookup number governmentDatabase
  lookup registration taxDatabase
```

Listi monaad

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

- intuitsioon: mitmesus
- definitsioon

```
return x   =  [x]

xs  >>= f  =  concat (map f xs)
```

- Näide (pseudokood):

```
sõpradePalgad :: Person -> [Int]
sõpradePalgad isik = do
  sõber <- getFriends isik
  töö   <- getEmployers sõber
  return (getPay töö sõber)
```

- … see on sama mis listikomprehensioon