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.

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 tähendus

- `a >>= f` on sama mis

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

- `a >> b` on sama mis

```
do a
   b
```

- Fixity:

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

- Saab kirjutada ühel real:

```
do { x <- a; b; f x }
```

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.)

Näiteks:

- `Maybe a`
- `[a]`
- `IO a`
- ...

Monaadidest üldisemalt (järg.)

Teoorias peaks funktsioonid rahuldama järgnevaid võrdusi:

- Functor

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

- Applicative

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

- Monad

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

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

Parsimise monaad

```
type Parser a = String -> [(a,String)]
return x = \ s -> [(x,s)]
p >>= g => \ s -> concatMap (\ (a,s) -> g a s) (p s)
```

Näide:

```
expr =     do n <- term
              keyw "+"
              m <- expr
              return (n+m)
       <|> term
term =     do n <- atom
              keyw "*"
              m <- term
              return (n*m)
       <|> atom
atom =     do keyw "("
              e <- expr
              keyw ")"
       <|> parse_int
```

Seisundi monaad

```
State s a   -- s on seisundi tüüp; a väärtuse tüüp
```

- definitsioon

```
get :: State s s
put :: s -> State s ()
runState :: State a b -> a -> (a, b)
```

- Tavaliselt tehakse tüübisünonüüm iga konkreetse alamprogrammi jaoks.

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a
```

...

Seisundi monaad II

- Näide:

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a

applyIsik :: String -> (Double -> Double) -> PangaArvutus ()
applyIsik nimi f = do
  s <- get
  put (map g s)
    where g (n,s) | n==nimi   = (n, f s)
                  | otherwise = (n, s)

rahaVälja :: String -> Float -> PangaArvutus Bool
rahaVälja nimi summa = do
  s <- get
  case lookup nimi s of
    Nothing -> return False
    Just r  -> do
      applyIsik nimi (\ x -> x-summa)
      return True
```