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
```

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: 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:

```
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..

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

- `[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`