Liidesed

Idrisis tuleb kirjutada funktsioone, mille tüübid erinevad ainult natuke:

```
equalChar   :   Char →   Char → Bool
equalInt    :    Int →    Int → Bool
equalString : String → String → Bool
```

Sellist koodi *ei saa* kirjutada ühe puhta parameetrilise polümorfse funktsiooniga, kuna funktsioonide implementatsioon on erinev:

```
equal : a → a → Bool
equal = ?eq_v  -- pole def. mis töötaks iga tüübi korral
```

Selleks saame teha liidese:

```
infix 6 ==?
interface Equal a where
    (==?) : a → a → Bool

Equal Integer where
    a ==? b = ?eq_Integer -- Integer'ide võrdsus
Equal Bool where
    a ==? b = ?eq_Bool    -- Bool'ide võrdsus
```

Tüübiklassid standardteegis

Idrise standardteegis on defineeritud tüübiklass `Eq`:

```
interface Eq a where
   (==), (≠) : a → a → Bool

    -- Minimaalne definitioon peab sisaldama:
    -- (==) või (≠)
   x ≠ y = not (x == y) -- vaikedefinitsioon
   x == y = not (x ≠ y) -- vaikedefinitsioon
```

Võrdlusoperaatori 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 *kontekstis* olema `Eq`:

```
lookup : Eq a ⇒ a → List (a, b) → Maybe b
```

Range liides

Enumeratsioone kirjeldab järgnev tüübiklass:

```
interface Range a where
   rangeFromTo : a → a → List a           -- [x..y]
   rangeFromThenTo : a → a → a → List a   -- [x,y..z]
   ...
```

REPL-is saab "`:doc Range`"-ga vaadata, mis instantsid implementeeritud on.

Standardteegi liidesed ja instantsid

- Aritmeetikaga seotud liidesed:
  - `Num`: `+`, `*`, `fromInteger`
  - `Integral`: `div`, `mod`
  - `Fractional`: `/`, `recip`
  - `Neg`: `-`, `negate`
  - `Abs`: `abs`
- Võrdlemisega seotud liidesed:
  - `Eq`: `==`, `/=`
  - `Ord`: `<`, `>`, `<=`, `>=`, `compare`, `max`, `min`
- Andmestruktuuridega seotud liidesed:
  - `Functor`: `map`
  - `Applicative`, `Monad`, `Foldable`, `Alternative`, `Traversable`
- Tekstiks tegemine `Show` liidesega: `show`

Liideste vahele ja instantside vahele tekivad seosed

- Kõik `Integral`-id on `Num`-id.
- Paaride võrdus on defineeritud üldiselt. `(Eq a, Eq b) => Eq (a,b)`

Ülesanne

Leia funktsiooni maxElem tüüp.

```
data T a = L a | B (T a) (T a)

maxElem :
maxElem (L x) = x
maxElem (B x y) = max (maxElem x) (maxElem y)

-- Main> :t max
-- Prelude.max : Ord ty ⇒ ty → ty → ty
```

Ülesanne

Olgu defineeritud:

```
data NPuu a = Br a (List (NPuu a))

puu : NPuu Int
puu = Br 3 [Br 2 [], Br 5 [Br 6 []], Br 10 [Br 8 [], Br 9 []]]


--           3
--         / | \
--        2  5  10
--           |  | \
--           6  8  9
```

Defineeri Eq tüübiklass võrdsuse kontrollimiseks. Näited:

```
Main> puu == puu
True
Main> Br 3 [Br 1 []] == Br 3 [Br 2 []]
False
```

Sisend-väljund Idrises

- Idrise puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random : () → Int`
- Lahendus: `IO` liides
  - 'IO a' tüüpi väärtus — „masin mis arvutab `a` tüüpi väärtuse“
  - `pure : a → IO a` — masin tagastab 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 = if x `mod` 2 == 1 then
                         pure "paaritu"
                     else
                         pure "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 = if x `mod` 2 == 1 then
                          pure "paaritu"
                      else
                          pure "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 = ...
```

do-süntaks II

Näide

```
proc : IO ()
proc = do
   s ← getLine
   let n : Int
       n = read s
       n2 : Int
       n2 = 2*n
   putStrLn ("Kaks␣korda␣" ++ s ++ "␣on␣" ++ show n2)
```

Do-süntaks algab `do`-ga, millele järgnevad *järjest töödeldavad* laused.

- Laused mustriga $x \leftarrow p$, (kui 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 : IO ()
main = do
   putStrLn "Kirjuta midagi!"
   xs ← getLine
   if (xs=="")
    then putStr "Sõnakuulmatu!"
    else do
       putStrLn "Tänan!"
       putStrLn ("Kirjutasid: " ++ xs)
```

Ülesanne

Kirjuta järgnev Pythoni programm Idrises:

```
name = input("Sisesta nimi: ")
print("Tere,", name, "!")
```

Idrises:

```
main : IO ()
main = ?main_rhs
```

Ülesanne

Kirjuta järgnev Idrise kood ilma do-süntaksita:

```
rnd : IO Int32
rnd = do x ← randomRIO (1,6)
         printLn x
         if x==6
            then do x' ← rnd
                    pure (x+x')
            else
                 pure x
```

Ülesanne

Teame, et `List` on `Monad`. Ehk saame do-süntaksit kasutada `IO` asemel `List`ide peal.

Mida teeb järgnev kood?

```
bla : List Int
bla =
    do x ← [1..3]
       y ← [10,20,30]
       pure (x+y)
```