Reaktiivsed pildid

- Piltide definitsioon:

```
data Picture = Region Color Region
             | Picture `Over` Picture
             | EmptyPic
             deriving Show
```

- Eesmärk on kirjutada programm, mis hiire vasaku nupu kliki korral toob viidatava regiooni pealmiseks; kui klikkida taustale, siis programm lõpetab töö.

Reaktiivsed pildid

- Piltide teisendamine regioonide listiks:

```
pictToList :: Picture -> [(Color,Region)]

pictToList  EmptyPic    = []
pictToList (Region c r) = [(c,r)]
pictToList (p1 `Over` p2)
         = pictToList p1 ++ pictToList p2
```

Reaktiivsed pildid

- Positsioonile vastava regiooni leidmine:

```
adjust :: [(Color,Region)] -> Coordinate ->
          (Maybe (Color,Region), [(Color,Region)])

adjust [] p = (Nothing, [])
adjust ((c,r):regs) p
       | r `containsR` p = (Just (c,r), regs)
       | otherwise       = (hit, (c,r) : rs)
       where (hit, rs) = adjust regs p
```


Reaktiivsed pildid

- Mitterekursiivne definitsioon:

```
adjust regs p
 = case (break (\(_,r) -> r `containsR` p) regs) of
      (top,hit:rest) -> (Just hit, top++rest)
      (_,[])         -> (Nothing, regs)
```

- Funktsioon `break` on eeldefineeritud:

```
break :: (a -> Bool) -> [a] -> ([a],[a])

break (>3) [1..6] ===> ([1,2,3],[4,5,6])
```

Reaktiivsed pildid

- Põhitsükkel:

```
loop :: Window -> [(Color,Region)] -> IO ()
loop w regs =
  do clearWindow w
     sequence_ [ drawRegionInWindow w c r
                                  | (c,r) <- reverse regs ]
     (x,y) <- getLBP w
     case (adjust regs (pixelToInch (x - xWin2),
                        pixelToInch (yWin2 - y))) of
       (Nothing,  _      ) -> closeWindow w
       (Just hit, newRegs) -> loop w (hit : newRegs)
```


Reaktiivsed pildid

- Pildi joonistamine:

```
draw2 :: String -> Picture -> IO ()
draw2 s p
  = runGraphics $
    do w <- openWindow s (xWin,yWin)
       loop w (pictToList p)
```

Tüübiklassid

- Aritmeetilised operaatorid:

```
(+) :: Int   -> Int   -> Int
(+) :: Float -> Float -> Float
```

  - Tüüp `(+) :: a -> a -> a` on liiga üldine!

- Polümorfne võrdus:

```
(==) :: a -> a -> Bool
```

  - Kuidas kontrollida näiteks funktsioonide võrdust?

- Polümorfne väljastus:

```
show :: a -> String
```

  - Funktsioon `show` on igal tüübil defineeritud erinevalt!

Tüübiklassid

- Ülemääratud operaatoreid defineeritakse *tüübiklassidega*

class [*context* =>] *classname tvar* **where** {*cbody*}

- Klassidefinitsiooni keha koosneb meetodide tüübisignatuuridest ja *default*-definitsioonidest.
- Näide — klass `Eq` (definitsioon prelüüdist):

```
class  Eq a  where
    (==), (/=)  :: a -> a -> Bool
    x /= y      =  not (x == y)
```

- Meetodide tüüpides esinevad tüübimuutujad on kitsendatud vastava klassikontekstiga.

```
(==) :: Eq a => a -> a -> Bool
```

Tüübiklassid

- Tüüpe saab kuulutada klassi kuuluvaks

  instance [*context* =>] *classname type* **where** {*ibody*}

- Deklareeritav tüüp peab olema kujul $tcon\ tvar_1 \ldots tvar_n$, kusjuures kõik tüübimuutujad peavad olema erinevad
- Deklaratsiooni keha koosneb meetodide definitsioonidest
- Default-definitsiooniga meetodi pole vaja (aga võib) uuesti defineerida

Tüübiklassid

- Näide — binaarpuude võrdus:

```
data IntBtree = Lf Int | Br IntBtree IntBtree

instance Eq IntBtree where
       Lf i1 ==  Lf i2      = i1 == i2
  Br t11 t12 == Br t21 t22  = t11 == t21 &&  t12 == t22
           _ == _           = False
```

- Näide — listide võrdus (defineeritud prelüüdis):

```
instance Eq a => Eq [a] where
      [] == []      = True
  (x:xs) == (y:ys) = x == y && xs == ys
       _ == _       = False
```

Tüübiklassid

- Näide — põõsaste võrdus:

```
data Bush a = One a | Two (Bush a) (Bush a)
                    | Many [Bush a]

instance Eq a => Eq (Bush a) where
      One x == One y     =     x == y
  Two x1 x2 == Two y1 y2 =    x1 == y1 && x2 == y2
    Many xs == Many ys   =    xs == ys
          _ == _         =    False
```

- On kasutatud *kolme* erinevat võrdust!!!

```
(==) :: a -> a -> Bool
(==) :: Bush a -> Bush a -> Bool
(==) :: [Bush a] -> [Bush a] -> Bool
```


Tüübiklassid

- Klassidest võib mõelda, kui predikaatidest üle tüüpide
- Tüübid, mis rahuldavad neid predikaate, omavad “lisa funktsionaalsust”
- Klasside deklaratsioonid defineerivad mis liiki “lisa funktsionaalsusega” on tegemist
- “Isendite” deklaratsioonid defineerivad selle “lisa funktsionaalsuse” konreetse tüübi jaoks

Tüübiklassid

- Klassikuuluvuse tuletamine:

```
data Bool = False | True
          deriving (Eq,Ord,Ix,Enum,Read,Show,Bounded)
data Maybe a = Nothing | Just a
             deriving (Eq, Ord, Read, Show)
data Either a b = Left a | Right b
                deriving (Eq, Ord, Read, Show)
data Ordering = LT | EQ | GT
          deriving (Eq,Ord,Ix,Enum,Read,Show,Bounded)
```

- `deriving` abil saab tuletada ainult eeldefineeritud klasse `Eq`, `Ord`, `Enum`, `Bounded`, `Show` ja `Read`.

Klass `Ord`

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
    compare x y | x == y    =  EQ
                | x <= y    =  LT
                | otherwise =  GT
     x <= y     = compare x y /= GT
     x <  y     = compare x y == LT
     x >= y     = compare x y /= LT
     x >  y     = compare x y == GT
     max x y    = if x >= y then x else y
     min x y    = if x <  y then x else y
```

Klass `Enum`

```
class  Enum a  where
    toEnum         :: Int -> a
    fromEnum       :: a -> Int
    enumFrom       :: a -> [a]              -- [n..]
    enumFromThen   :: a -> a -> [a]         -- [n,n'..]
    enumFromTo     :: a -> a -> [a]         -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]    -- [n,n'..m]

succ, pred :: Enum a => a -> a
succ       =  toEnum . (+1) . fromEnum
pred       =  toEnum . (subtract 1) . fromEnum
```


Arvudega seotud klassid

- Klass `Num`

```
class  (Eq a, Show a) => Num a  where
     (+), (-), (*)     :: a -> a -> a
     negate            :: a -> a
     abs, signum       :: a -> a
     fromInteger       :: Integer -> a

     x - y             =  x + negate y
```

- Klass `Real`

```
class  (Num a, Ord a) => Real a  where
    toRational       ::  a -> Rational
```

Arvudega seotud klassid

- Klass `Integral`

```
class  (Real a, Enum a) => Integral a  where
     quot, rem        :: a -> a -> a
     div, mod         :: a -> a -> a
     quotRem, divMod  :: a -> a -> (a,a)
     toInteger        :: a -> Integer
```

- Klass `Fractional`

```
class  (Num a) => Fractional a  where
    (/)              :: a -> a -> a
    recip            :: a -> a
    fromRational     :: Rational -> a
```

- Lisaks on eeldefineeritud `RealFrac`, `Floating`, `RealFloat`

Klassid `Show` ja `Read`

```
type  ShowS    = String -> String

class  Show a  where
    showsPrec :: Int ->  a  -> ShowS
    showList  ::        [a] -> ShowS
    show      :: a -> String
    show x    =  showsPrec 0 x ""


class  Read a  where ...

read :: (Read a) => String -> a
```

Ratsionaalarvud

```
module Rats ( Rat((:/:)) ) where
infix 7 :/:
data Rat = Integer :/: Integer

instance Show Rat where
   showsPrec p (a :/: b) s = show a ++ "/" ++ show b ++ s

mkRat :: Rat -> Rat
mkRat (x :/: 0) = error ("Invalid rational number "  ++
                   "(in mkRat) " ++ show (x :/: 0) ++ "\n")
mkRat (0 :/: y) = 0 :/: 1
mkRat (x :/: y) = (a `div` d) :/: (b `div` d)
       where {a = (signum y) * x; b = abs y; d = gcd a b}
```


Ratsionaalarvud

```
compareRat ::   (Integer -> Integer -> Bool)
              -> Rat -> Rat -> Bool
compareRat r (a :/: b) (c :/: d)
   | b == 0    = errRat (a :/: 0)
   | d == 0    = errRat (c :/: 0)
   | otherwise = r (a*d) (b*c)
      where errRat (x :/: y) = error ("Invalid rational "
                                 ++ "number (in compareRat) "
                                 ++ show (x :/: y) ++ "\n")

instance Eq   Rat where   (==)  = compareRat (==)
instance Ord  Rat where   (<=)  = compareRat (<=)
```

Ratsionaalarvud

```
instance Num Rat where
   (a :/: b) + (c :/: d) = mkRat ((a*d + c*b) :/: (b*d))
   (a :/: b) - (c :/: d) = mkRat ((a*d - c*b) :/: (b*d))
   (a :/: b) * (c :/: d) = mkRat ((a*c) :/: (b*d))
   negate (a :/: b)      = mkRat ((-a) :/: b)
   fromInteger x         = x :/: 1

instance Real Rat where
   toRational (a :/: b)  = a % b

instance Fractional Rat where
   (a :/: b) / (c :/: d) = mkRat ((a*d) :/: (b*c))
   fromRational x  = mkRat (numerator x :/: denominator x)
```