Lihtne graafika

- Graafiline “Hello, World”

```
import SOEGraphics
main1
  = runGraphics (
    do w <- openWindow
            "My First Graphics Program" (300,300)
       drawInWindow w
           (text (100,200) "Hello Graphics World")
       k <- getKey w
       closeWindow w
    )
```

Lihtne graafika

- Graafika käske:

```
type Title = String
type Size  = (Int,Int)
type Point = (Int,Int)

runGraphics  :: IO () -> IO ()
openWindow   :: Title -> Size -> IO Window
drawInWindow :: Window -> Graphic -> IO ()
text         :: Point -> String -> Graphic
getKey       :: Window -> IO Char
closeWindow  :: Window -> IO ()
```

Lihtne graafika

- Graafiliste objektide joonistamise käske:

```
ellipse     :: Point -> Point -> Graphic
line        :: Point -> Point -> Graphic
polyline    :: [Point] -> Graphic
polygon     :: [Point] -> Graphic
```

- Graafiliste objektide "värvimine":

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White

withColor :: Color -> Graphic -> Graphic
```

Lihtne graafika

- Näide:

```
pic1 = withColor Red (ellipse (150,150) (300,200))
pic2 = withColor Blue
         (polyline [(100,50),(200,50),
                    (200,250),(100,250),(100,50)])

spaceClose w = do k <- getKey w
                  if k == ' ' then closeWindow w
                              else spaceClose w

main2 = runGraphics $
    do w <- openWindow
              "Some Graphics Figures" (300,300)
       drawInWindow w pic1
       drawInWindow w pic2
       spaceClose w
```

Geomeetriliste kujundite joonistamine

- Geomeetriliste kujundite andmetüüp:

```
data Shape = Rectangle Side Side
           | Ellipse Radius Radius
           | RtTriangle Side Side
           | Polygon [Vertex]
     deriving Show

type Radius = Float
type Side   = Float
type Vertex = (Float,Float)

square s = Rectangle s s
circle r = Ellipse r r
```

Geomeetriliste kujundite joonistamine

- `Rectangle`, `Ellipse`, `RtTriangle` ei sõltu paigutusest, aga `Polygon` asukoht on definitsiooniga määratud.
- Asukohast sõltumatuted kujundite aknasse joonistamisel paigutame nad akna keskele; so. kujundite joonistamisel on koordinaatsüsteemi algpunkt akna keskel.
- Samal ajal, graafika-teegis kasutatava koordinaadistiku algpunkt on akna vasak ülemine nurk.
- Kujundite esitamisel kasutatakse pikkusühikuna tolli; graafika-teegis aga pikseleid.
- Vaja defineerida teisendusfunktsioonid ühest koordinaatsüsteemist teise!

Geomeetriliste kujundite joonistamine

- Ühikute teisendamine

```
inchToPixel  :: Float -> Int
inchToPixel x = round (100*x)

pixelToInch  :: Int -> Float
pixelToInch n = fromIntegral n / 100
```

- Koordinaatide teisendamine

```
xWin, yWin :: Int
xWin = 600
yWin = 500

xWin2, yWin2 :: Int
xWin2 = xWin `div` 2
yWin2 = yWin `div` 2
```

Geomeetriliste kujundite joonistamine

- Koordinaatide teisendamine (järg)

```
trans :: Vertex -> Point
trans (x,y) = ( xWin2 + inchToPixel x,
                yWin2 - inchToPixel y )

transList :: [Vertex] -> [Point]
transList = map trans
```

Geomeetriliste kujundite joonistamine

- Kujundite teisendamine graafikaks

```
shapeToGraphic :: Shape -> Graphic
shapeToGraphic (Rectangle s1 s2)
        = let s12 = s1/2
              s22 = s2/2
          in polygon (transList [(-s12,-s22),(-s12,s22),
                                 (s12,s22),(s12,-s22)])
shapeToGraphic (Ellipse r1 r2)
        = ellipse (trans (-r1,-r2)) (trans (r1,r2))
shapeToGraphic (RtTriangle s1 s2)
        = polygon (transList [(0,0),(s1,0),(0,s2)])
shapeToGraphic (Polygon pts)
        = polygon (transList pts)
```

Geomeetriliste kujundite joonistamine

```
sh1,sh2,sh3,sh4 :: Shape

sh1 = Rectangle 3 2
sh2 = Ellipse 1 1.5
sh3 = RtTriangle 3 2
sh4 = Polygon [(-2.5,2.5), (-1.5,2.0),
               (-1.1,0.2), (-1.7,-1.0),
               (-3.0,0)]
```

Geomeetriliste kujundite joonistamine

```
main0
  = runGraphics $
    do w <- openWindow "Drawing Shapes" (xWin,yWin)
       drawInWindow w (withColor Red
                                 (shapeToGraphic sh1))
       drawInWindow w (withColor Blue
                                 (shapeToGraphic sh2))
       spaceClose w
```

Geomeetriliste kujundite joonistamine

```
type ColoredShapes = [(Color,Shape)]

shs :: ColoredShapes
shs  = [(Red,sh1),    (Blue,sh2),
        (Yellow,sh3), (Magenta,sh4)]

drawShapes :: Window -> ColoredShapes -> IO ()
drawShapes w [] = return ()
drawShapes w ((c,s):cs)
  = do drawInWindow w (withColor c (shapeToGraphic s))
       drawShapes w cs

main1 = runGraphics $
    do w <- openWindow "Drawing Shapes" (xWin,yWin)
       drawShapes w shs
       spaceClose w
```

Geomeetriliste kujundite joonistamine

```
conCircles = map circle [2.4,2.1 .. 0.3]

coloredCircles =
  zip [Black, Blue, Green, Cyan, Red,
               Magenta, Yellow, White]
      conCircles

main2
  = runGraphics $
    do w <- openWindow "Bull's Eye" (xWin,yWin)
       drawShapes w coloredCircles
       spaceClose w
```

Regioonid

- Andmetüüp Shape võimaldab esitada lihtsaid geomeetrilisi kujundeid, mida me saame ka lihtsasti graafiliselt joonistada
- Kuidas . . .
  - konstrueerida lihtsamatest kujunditest keerulisemaid?
  - lahendada “probleemi”, et osad kujundid (ellips, ristkülik ja täisnurkne kolmnurk) on tasandi koordinaatteljestiku suhtes tsentreeritud?

Regioonid

- Regiooni definitsioon:

```
data Region = Shape Shape
            | Translate Vector Region
            | Scale     Vector Region
            | Complement Region
            | Region `Union` Region
            | Region `Intersect` Region
            | Empty
     deriving Show

type Vector = (Float,Float)
```

Regioonide “tähendus”

- Regioon on tasandil asetsevate punktide hulk
- Tihti (peaaegu alati?) lõpmatu
- Regioonidele formaalse “tähenduse” andmiseks vaja efektiivset moodust lõpmatute hulkade esitamiseks
- Kasutame karakteristlikke funktsioone

```
type Set a = a -> Bool
```

- Näide:

```
evenInts :: Set Int
evenInts x = (x `mod` 2) == 0
```

Operatsioone hulkadel

- Ülesanne — kuidas leida:
  - kahe hulga ühendit?

    ```
    union :: Set a -> Set a -> Set a
    x `union` y =
    ```

  - kahe hulga ühisosa?

    ```
    intersect :: Set a -> Set a -> Set a
    x `intersect` y =
    ```

  - vastandhulka?

    ```
    complement :: Set a -> Set a
    complement x =
    ```

Regioonide karakteristlikud funktsioonid

- Etteantud koordinaadi kuulumine regiooni:

```
type Coordinate = (Float,Float)
containsR :: Region -> Coordinate -> Bool
```

- Paneme tähele, et `containsR r` on karakteristlik funktsioon; ehk

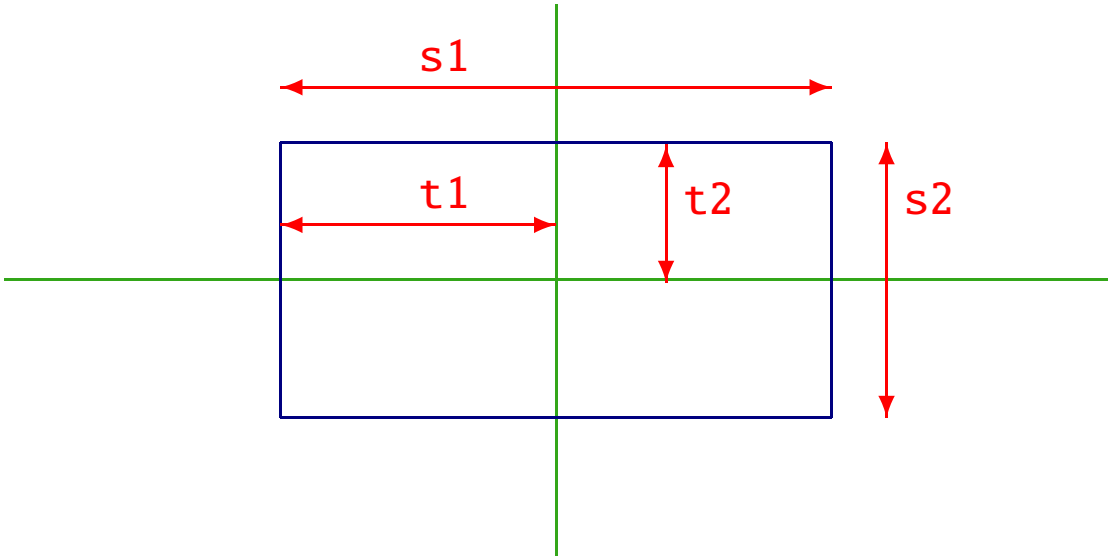
```
containsR :: Region -> Set Coordinate
```

- Defineerime rekursiooniga üle `Region` andmetüübi.
- Rekursiooni baasjuhuks on kujundite andmetüüp `Shape`, mille jaoks defineerime eraldi funktsiooni:

```
containsS :: Shape -> Coordinate -> Bool
```

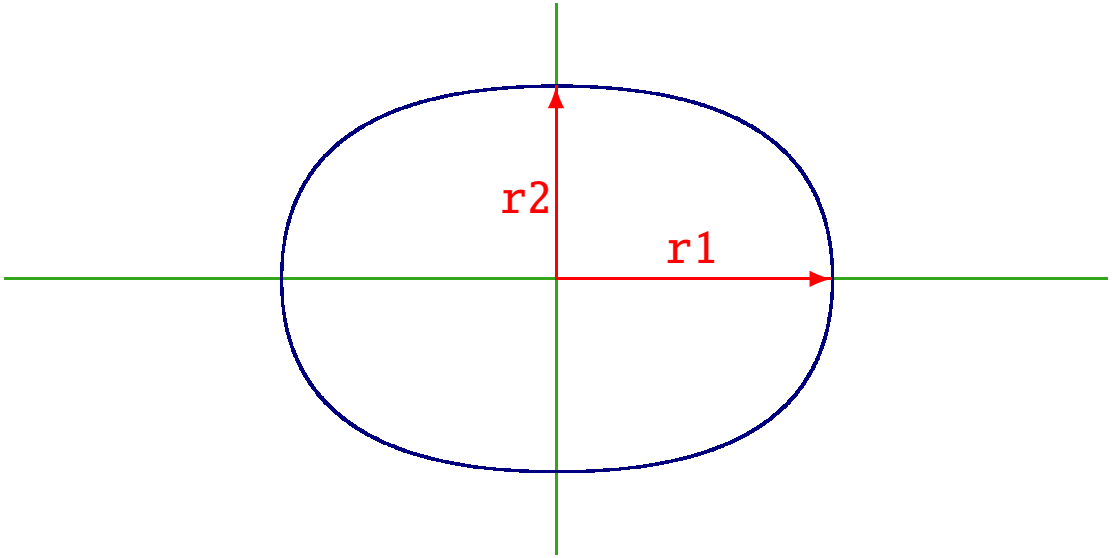
Ristkülik

```
(Rectangle s1 s2) `containsS` (x,y)
   = let t1 = s1/2
         t2 = s2/2
     in -t1<=x && x<=t1 && -t2<=y && y<=t2
```



Ellips

```
(Ellipse r1 r2) `containsS` (x,y)
                    = (x/r1)^2 + (y/r2)^2 <= 1
```



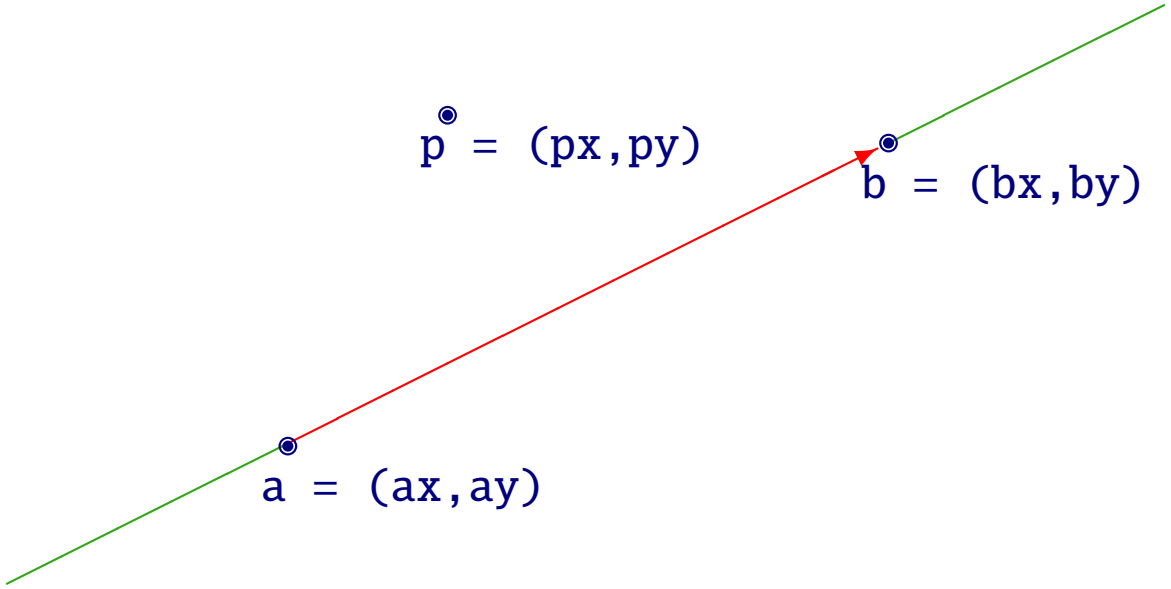
Polügon

Punkt p asub (kumera) polügoni sees, kui ta asub “vasakul pool” igast vastupäeva orienteeritud polügoni küljest

```
(Polygon pts) `containsS` p
   = let shiftpts     = tail pts ++ [head pts]
         leftOfList   = map isLeftOfp (zip pts shiftpts)
         isLeftOfp p' = isLeftOf p p'
     in and leftOfList
```

Joone “vasak pool”

Punktist a punkti b mineva vektori poolt tekitatud joonest vasakul asumine



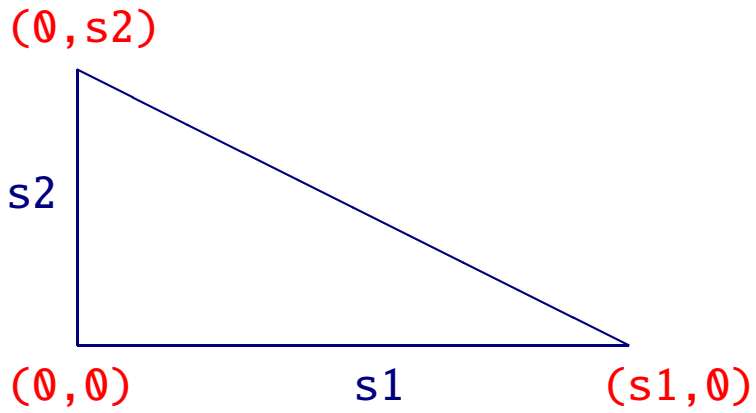
Joone “vasak pool”

```
type Ray = (Coordinate, Coordinate)

isLeftOf :: Coordinate -> Ray -> Bool
(px,py) `isLeftOf` ((ax,ay),(bx,by))
       = let (s,t) = (px-ax, py-ay)
             (u,v) = (px-bx, py-by)
         in  s*v >= t*u
```

Täisnurkne kolmnurk

```
(RtTriangle s1 s2) `containsS` p
   = (Polygon [(0,0),(s1,0),(0,s2)]) `containsS` p
```



Koordinaadi kuulumine regiooni

```
(Shape s) `containsR` p = s `containsS` p
(Translate (u,v) r) `containsR` (x,y)
                        = let p = (x-u,y-v) in r `containsR` p
(Scale (u,v) r) `containsR` (x,y)
                        = let p = (x/u,y/v) in r `containsR` p
(Complement r) `containsR` p
                        = not (r `containsR` p)
(r1 `Union` r2)     `containsR` p
                        = r1 `containsR` p || r2 `containsR` p
(r1 `Intersect` r2) `containsR` p
                        = r1 `containsR` p && r2 `containsR` p
Empty `containsR` p     = False
```

Regioonide algebra

- Iga `r1`, `r2` ja `r3` korral

  ```
  (r1 `Union` (r2 `Union` r3)) `containsR` p
  ```

  siis ja ainult siis kui

  ```
  ((r1 `Union` r2) `Union` r3) `containsR` p
  ```

- . . . , ehk `Union` on assotsiatiivne:

  ```
  r1 `Union` (r2 `Union` r3)
                === (r1 `Union` r2) `Union` r3
  ```

- Tõestus on lihtne arvutus . . .

Assotsiatiivsuse tõestus

```
(r1 `Union` (r2 `Union` r3)) `containsR` p
==> (r1 `containsR` p) ||
        ((r2 `Union` r3) `containsR` p)
==> (r1 `containsR` p) ||
        ((r2 `containsR` p) || (r3 `containsR` p))
==> ((r1 `containsR` p) || (r2 `containsR` p)) ||
         (r3 `containsR` p)
==> ((r1 `Union` r2) `containsR` p) ||
         (r3 `containsR` p)
==> ((r1 `Union` r2) `Union` r3) `containsR` p
```

NB! Tõestus sõltub (||) assotsiatiivsusest (mille saab ka tõstada analoogselt)

Teisi omadusi

- `Union` ja `Intersect` on assotsiatiivsed
- `Union` ja `Intersect` on kommutatiivsed
- `Union` ja `Intersect` on distributiivsed
- Iga `r` korral

```
r `Union` Empty === r
r `Intersect` Complement Empty === r
r `Union` Complement r === Complement Empty
r `Intersect` Complement r === Empty
```

Pildid

- Pildid on konstrueeritud regioonidest
  - Regioonid omakorda geomeetrilistest kujunditest
- Pildid lisavad regioonidele värvid

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

Pildid

- Piltide joonistamine:

```
drawPic :: Window -> Picture -> IO ()

drawPic w (Region c r)   = drawRegionInWindow w c r
drawPic w (p1 `Over` p2) = do { drawPic w p2
                              ; drawPic w p1
                              }
drawPic w EmptyPic       = return ()
```

Regioonide joonistamine

- Regioonide joonistamisel kasutame `SOEGraphics` teegis defineeritud vahendeid
- `SOEGraphics` defineerib oma graafilise regiooni abstraktse tüübi, mis kannab ka nime `Region`
- Nimekonfliktide vältimiseks peame selle importima kvalifitseeritult

```
import SOEGraphics hiding (Region)
import qualified SOEGraphics as G (Region)
```

- Nüüd saame graafilisi regioone kasutada nimega `G.Region`

Graafilised regioonid

- Funktsioonid graafiliste regioonidega:

```
createRectangle :: Point -> Point -> G.Region
createEllipse   :: Point -> Point -> G.Region
createPolygon   :: [Point] -> G.Region

andRegion       :: G.Region -> G.Region -> G.Region
orRegion        :: G.Region -> G.Region -> G.Region
xorRegion       :: G.Region -> G.Region -> G.Region
diffRegion      :: G.Region -> G.Region -> G.Region

drawRegion      :: G.Region -> Graphic
```

Regioonide joonistamine

- Vaja teisendada meie regioonid graafilisteks regioonideks

```
regionToGRegion :: Region -> G.Region
```

- Siis on regioonide joonistamine lihtne:

```
drawRegionInWindow :: Window -> Color -> Region -> IO ()
drawRegionInWindow w c r
             = drawInWindow w
                 (withColor c
                    (drawRegion (regionToGRegion r)))
```

Regiooni teisendamine graafiliseks

- Esimene (lihtsustatud) katse:

```
data NewRegion = Rect Side Side

regToNReg :: Region -> NewRegion
regToNReg (Shape (Rectangle sx sy)) = Rect sx sy
regToNReg (Scale (x,y) r)  = regToNReg (scaleReg (x,y) r)
scaleReg (x,y) (Shape (Rectangle sx sy))
                           = Shape (Rectangle (x*sx) (y*sy))
scaleReg (x,y) (Scale s r) = Scale s (scaleReg (x,y) r)
```

Regiooni teisendamine graafiliseks

- Antud lahendus on väga ebaefektiivne

```
(Scale (x1,y1)
       (Scale (x2,y2)
                 ... (Shape (Rectangle sx sy))
                 ... ))
```

- Kui meil on vaja teisendada regiooni, mida on skaleeritud n korda, siis `regToNReg` teeb $\mathcal{O}(n^2)$ regioonipuu läbivaatust!
- Probleem (ja ka lahendus) on sama mis listide ümberpööramisel

Listide ümberpööramine

- Lihtne, kuid ebaefektiivne definitsioon:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- Akumuleeriva parameetriga (lineaarse keerukusega) versioon:

```
reverse xs = rev [] xs
   where rev acc []     = acc
         rev acc (x:xs) = rev (x:acc) xs
```

Regiooni teisendamine graafiliseks

- Teine (lihtsustatud) katse:

```
regToNReg :: Region -> NewRegion
regToNReg r = rToNR (1,1) r

rToNR :: (Float,Float) -> Region -> NewRegion
rToNR (x1,y1) (Shape (Rectangle sx sy))
                        = Rect (x1*sx) (y1*sy)
rToNR (x1,y1) (Scale (x2,y2) r)
                        = rToNR (x1*x2,y1*y2) r
```

- Lõppversioonis tuleb lisaks skaleerimisfaktorile akumuleerida ka nihutusfaktorit

Regiooni teisendamine graafiliseks

```
regionToGRegion :: Region -> G.Region
regionToGRegion r = regToGReg (0,0) (1,1) r

regToGReg :: Vector -> Vector -> Region -> G.Region
regToGReg loc sca (Shape s)
       = shapeToGRegion loc sca s
regToGReg loc (sx,sy) (Scale (u,v) r)
       = regToGReg loc (sx*u,sy*v) r
regToGReg (lx,ly) (sx,sy) (Translate (u,v) r)
       = regToGReg (lx+u*sx,ly+v*sy) (sx,sy) r
```

Regiooni teisendamine graafiliseks

- Kuna tühja graafilist regiooni `SOEGraphics` ei defineeri, siis kautame "tühja ristkülikut"

```
regToGReg loc sca Empty
        = createRectangle (0,0) (0,0)
```

- Ühendi teisendamine:

```
regToGReg loc sca (r1 `Union` r2)
        = let gr1 = regToGReg loc sca r1
              gr2 = regToGReg loc sca r2
          in  orRegion gr1 gr2
```

- Kuna ühisosa (ja ka täiend) on analoogilised, siis on mõistlik defineerida seda ühist malli esitav abifunktsioon

Regiooni teisendamine graafiliseks

```
primGReg loc sca r1 r2 op
      = let gr1 = regToGReg loc sca r1
            gr2 = regToGReg loc sca r2
        in  op gr1 gr2

regToGReg loc sca (r1 `Union` r2)
      = primGReg loc sca r1 r2 orRegion
regToGReg loc sca (r1 `Intersect` r2)
      = primGReg loc sca r1 r2 andRegion
regToGReg loc sca (Complement  r)
      = primGReg loc sca winRect r diffRegion
```

Regiooni teisendamine graafiliseks

- Regiooni täiendi teisendamiseks on vaja kogu tasandit hõlmavat graafilist regiooni
- Kuna täpselt sellist graafilist regiooni ei ole, siis kasutame sellena akna suurust ristkülikut:

```
winRect :: Region
winRect = Shape (Rectangle (pixelToInch xWin)
                           (pixelToInch yWin))
```

- Jäänud on veel geomeetriliste kujundite teisendamine

```
shapeToGRegion :: Vector -> Vector -> Shape -> G.Region
```

Kujundite teisendamine

```
shapeToGRegion (lx,ly) (sx,sy) s
  = case s of
      Rectangle s1 s2
        -> createRectangle (trans (-s1/2,-s2/2))
                           (trans ( s1/2, s2/2))
      Ellipse r1 r2
        -> createEllipse (trans (-r1,-r2))
                         (trans ( r1, r2))
      Polygon vs
        -> createPolygon (map trans vs)
      RtTriangle s1 s2
        -> createPolygon (map trans [(0,0),(s1,0),(0,s2)])
  where trans (x,y) = (xWin2 + inchToPixel (lx+x*sx),
                       yWin2 - inchToPixel (ly+y*sy))
```

Piltide joonistamine

- Näide:

```
draw :: String -> Picture -> IO ()
draw s p = runGraphics $
           do w <- openWindow s (xWin,yWin)
              drawPic w p
              spaceClose w
r1 = Shape (Rectangle 3 2)
r2 = Shape (Ellipse 1 1.5)
r3 = Shape (RtTriangle 3 2)
r4 = Shape (Polygon [(-2.5,2.5), (-3.0,0),
         (-1.7,-1.0),(-1.1,0.2), (-1.5,2.0)])
```

Piltide joonistamine

- Näide (jätkub):

```
xUnion :: Region -> Region -> Region
p1 `xUnion` p2 = (p1 `Intersect` Complement p2)
                 `Union`
                 (p2 `Intersect` Complement p1)
reg1 = r3 `xUnion`
          (r1 `Intersect` Complement r2
                                    `Union` r4)
pic1 = Region Blue reg1
```