Geomeetrilised kujundid

- Geomeetriliste kujundite definitsioon:

```
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)
```


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)
```

Pildid

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

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

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


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 $O(n^2)$ 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:

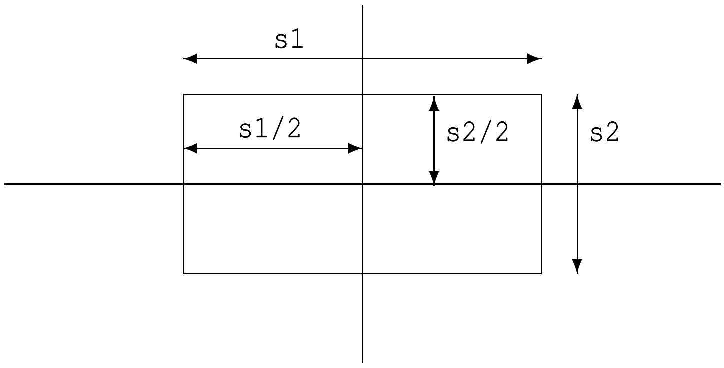
```
winRect :: Region
winRect = Shape (Rectangle (pixelToInch xWin)
                           (pixelToInch yWin))
```

- Jäänud on veel geomeetriliste kujundite teisendamine

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

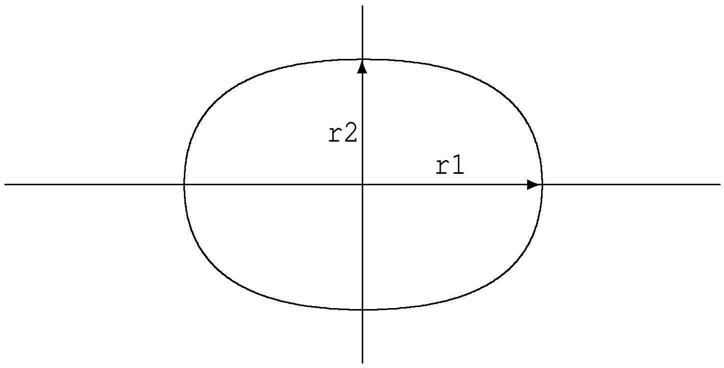

Ristkülik

```
shapeToGRegion (lx,ly) (sx,sy) (Rectangle s1 s2)
  = createRectangle (trans (-s1/2,-s2/2))
                    (trans ( s1/2, s2/2))
  where trans (x,y) = (xWin2 + inchToPixel((x+lx)*sx),
                       yWin2 - inchToPixel((y+ly)*sy))
```



Ellips

```
shapeToGRegion (lx,ly) (sx,sy) (Ellipse r1 r2)
  = createEllipse (trans (-r1,-r2)) (trans (r1,r2))
  where trans (x,y) = (xWin2 + inchToPixel((x+lx)*sx),
                       yWin2 - inchToPixel((y+ly)*sy))
```



Polügon ja täisnurkne kolmnurk

```
shapeToGRegion (lx,ly) (sx,sy) (Polygon vs)
  = createPolygon (map trans vs)
  where trans (x,y) = (xWin2 + inchToPixel((x+lx)*sx),
                       yWin2 - inchToPixel((y+ly)*sy))

shapeToGRegion (lx,ly) (sx,sy) (RtTriangle s1 s2)
  = createPolygon (map trans [(0,0),(s1,0),(0,s2)])
  where trans (x,y) = (xWin2 + inchToPixel((x+lx)*sx),
                       yWin2 - inchToPixel((y+ly)*sy))
```

Kujundite teisendamine uuesti

- Funktsioon `trans` kordub iga võrduse juures, kuna where-konstruktsioon käib ainult ühe võrduse kohta
- Võime defineerida globaalselt, aga siis peame andma lisaparameetritena nihke- ja skaleerimisfaktori
- Teine võimalus on kasutada case-avaldist:

```
shapeToGRegion (lx,ly) (sx,sy) s
  = case s of
      Rectangle s1 s2
        -> createRectangle (trans (-s1/2,-s2/2))
                           (trans ( s1/2, s2/2))
```


Kujundite teisendamine uuesti

```
      ...
      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
```