CS 457/557 Functional Programming

Lecture 10

Drawing Regions

Pictures

- Drawing Pictures
  - Pictures are composed of Regions

    Regions are composed of shapes
  - Pictures add color and layering

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

- We need to use SOEGraphics, but SOEGraphics has its own Region datatype.

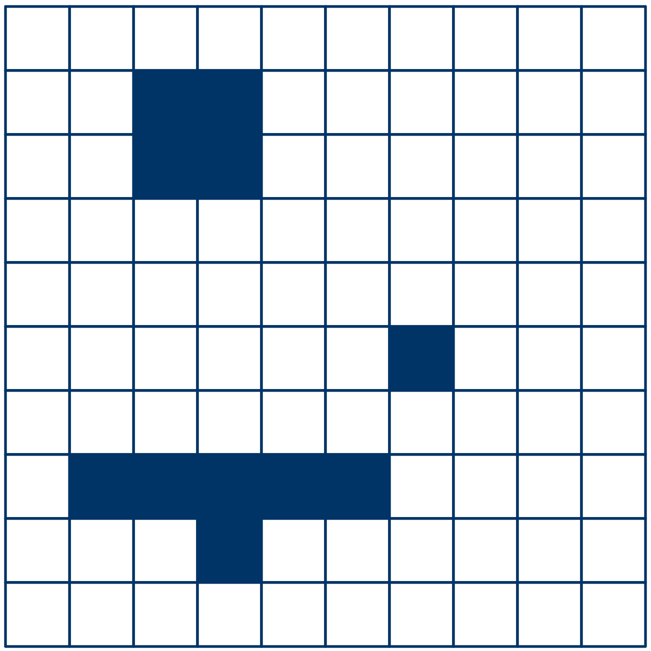
```
import SOEGraphics hiding (Region)
import qualified SOEGraphics as G (Region)
```

Recall the Region Datatype

```
data Region =
    Shape Shape                      -- primitive shape
  | Translate Vector Region          -- translated region
  | Scale     Vector Region          -- scaled region
  | Complement Region                -- inverse of a region
  | Region `Union` Region            -- union of regions
  | Region `Intersect` Region        -- intersection of regions
  | Empty
```

- How do we draw things like the intersection of two regions, or the complement of a region? These are hard things to do efficiently. Fortunately, the `G.Region` interface uses lower-level support to do this for us.

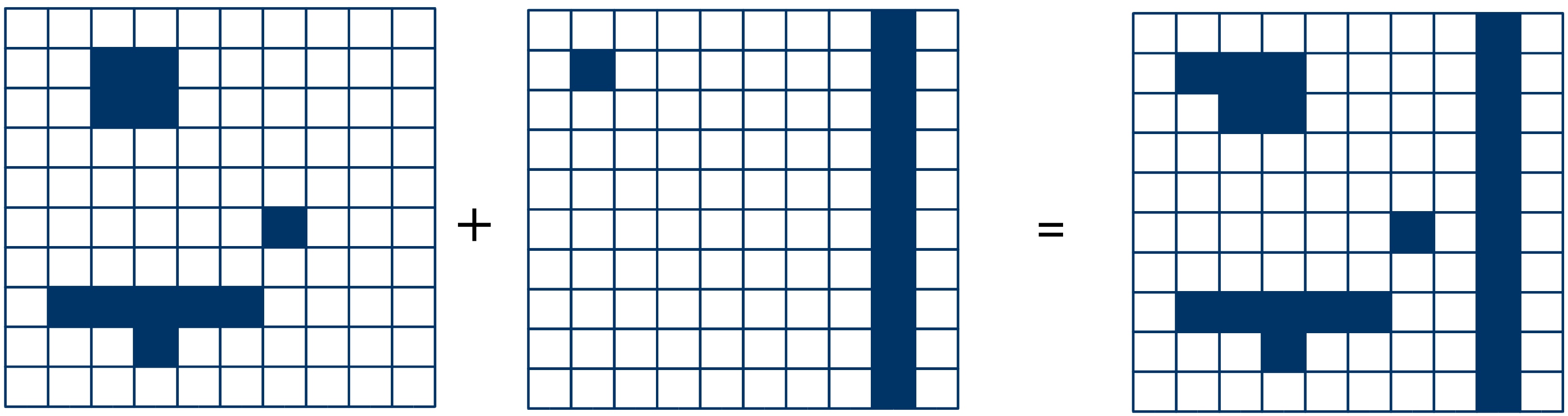
G.Region



- The `G.Region` datatype interfaces more directly to the underlying hardware. It is essentially a two-dimensional array or “bit-map”, storing a binary value for each pixel in the window.

Efficient Bit-Map Operations

- There is efficient low-level support for combining bit-maps using a variety of operators. For example, for union:



- These operations are fast, but data (space) intensive, and this space needs to be explicitly allocated and de-allocated, a job that seems easier in a much lower-level language.

G.Region Interface

```
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
```

These functions are defined in the SOEGraphics library module.

Drawing G.Region

- To draw things quickly, turn them into a `G.Region`, then turn the `G.Region` into a graphic object, and then use all the machinery we have built up so far to display the object.

```
drawRegionInWindow ::
       Window -> Color -> Region -> IO ()

drawRegionInWindow w c r =
   drawInWindow w
     (withColor c (drawRegion (regionToGRegion r)))
```

- All we need to define, then, is: `regionToGRegion`.
  - But first, let's define what it means to draw a picture.

Drawing Pictures

- Pictures combine multiple regions into one big picture. They provide a mechanism for placing one sub-picture on top of another.

```
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 ()
```

- Note that **p2** is drawn before **p1**, since we want **p1** to appear "over" **p2**.

Summary

- We have a rich calculus of **`Shape`**s, which we can draw, take the perimeter of, and tell if a point lies within.
- We defined a richer data type **`Region`**, which allows more complex compositions (intersection, complement, etc.).
  - We gave **`Region`** a mathematical semantics as a set of points in the 2-dimensional plane.
  - We defined some interesting operators like **`containsR`** which is the characteristic function for a region.
  - The rich nature of **`Region`** makes it hard to draw efficiently, so we use a lower level datatype **`G.Region`**, which relies on features like overwriting and explicit allocation and deallocation of memory.
  - We can think of **`Region`** as a high-level interface to **`G.Region`** that hides low-level details.
- We enriched things even further with the **`Picture`** type, which adds color and layering.

Turning a Region into a G.Region

Experiment with a subset of task to illustrate an efficiency problem.

Just consider rectangular shapes and scaling.

```
regToGReg0 :: Region -> G.Region
regToGReg0 (Shape (Rectangle sx sy))
      = createRectangle (trans(-sx/2,-sy/2))
                        (trans (sx/2,sy/2))
regToGReg0 (Scale (x,y) r)
      = regToGReg0 (scaleReg (x,y) r)
  where 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)
```

A Problem

- Consider

```
(Scale (x1,y1)
       (Scale (x2,y2)
              (Scale (x3,y3)
                     ... (Shape (Rectangle sx sy))
                      ... )))
```

- If the scaling is n levels deep, how many traversals does **`regToGReg1`** perform over the **`Region`** tree?

We've Seen This Before

- Believe it or not we have encountered this problem before. Recall the definition of **`reverse`**:

```
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]
  where []     ++ zs = zs
        (y:ys) ++ zs = y : (ys ++ zs)
```

- How did we solve this? We used an extra accumulating parameter:

```
reverse xs = revhelp xs []
  where revhelp [] zs     = zs
        revhelp (x:xs) zs = revhelp xs (x:zs)
```

- We can do the same thing for **`Regions`**.

Accumulate the Scaling Factor

```
regToGReg1 :: Region -> G.Region
regToGReg1 r = rToNR (1,1) r
  where rToGR :: (Float,Float) -> Region -> G.Region
        rToGR (x1,y1) (Shape (Rectangle sx sy))
              = createRectangle
                   (trans(-sx*x1/2,-sy*y1/2))
                   (trans (sx*x1/2,sy*y1/2))
        rToGR (x1,y1) (Scale (x2,y2) r)
              = rToGR (x1*x2,y1*y2) r
```

- To solve our original problem, repeat this for all the constructors of **`Region`** (not just **`Shape`** and **`Scale`**). We also need to handle translation as well as scaling.

Final Version

```
regToGReg2 :: Vector -> Vector -> Region -> G.Region
regToGReg2 loc sca (Shape s) = shapeToGRegion loc sca s
regToGReg2 (x,y) sca (Translate (u,v) r)
  = regToGReg2 (x+u, y+v) sca r
regToGReg2 loc (x,y) (Scale (u,v) r)
  = regToGReg2 loc (x*u, y*v) r
regToGReg2 loc sca Empty = createRectangle (0,0) (0,0)
regToGReg2 loc sca (r1 `Union` r2)
  = let gr1 = regToGReg2 loc sca r1
        gr2 = regToGReg2 loc sca r2
    in orRegion gr1 gr2
```

- Assuming, of course, that we can define:
  `shapeToGRegion :: Vector -> Vector -> Shape -> G.Region`
  and write rules for `Intersect`, `Complement` etc.

A Matter of Style

- While the function on the previous page shows how to solve the problem, there are several stylistic issues that could make it more readable and understandable.

- The style of defining a function by patterns becomes cluttered when there are many parameters (other than the one which has the patterns).

- The pattern of explicitly allocating and deallocating (bit-map) `G.Region`'s will be repeated in cases for intersection and for complement, so we should abstract it, and give it a name.

Abstract the Low-Level Bit-Map Details

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

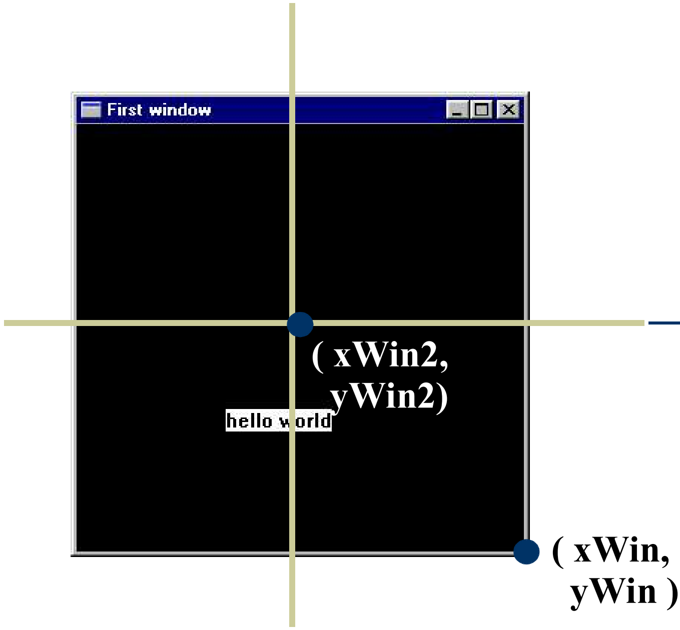
Redo with a Case Expression

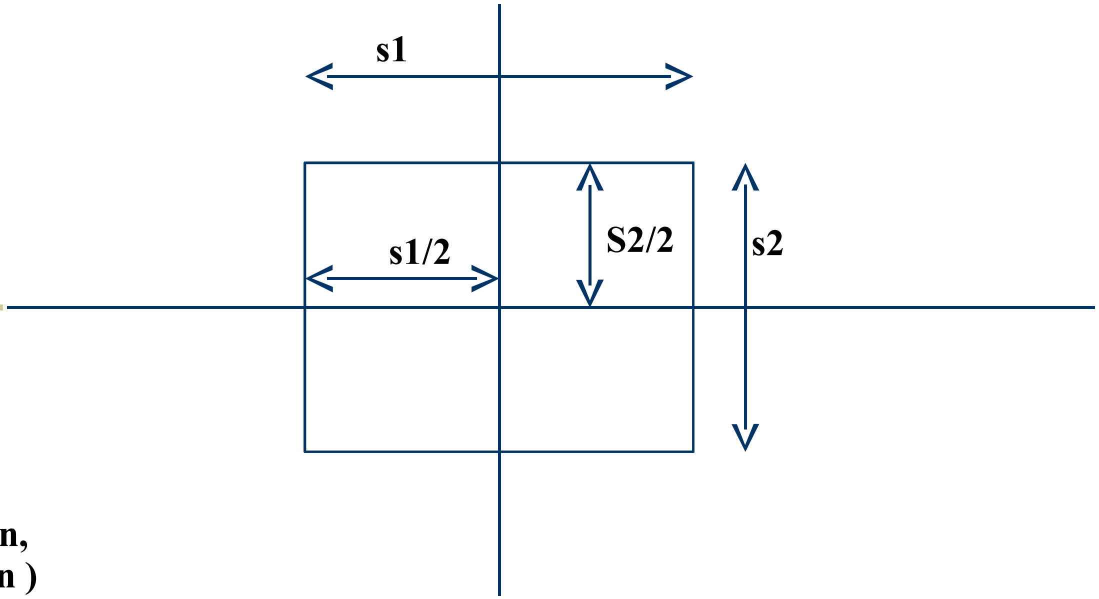
```
regToGReg :: Vector -> Vector -> Region -> G.Region
regToGReg (loc@(x,y)) (sca@(a,b)) shape =
 case shape of
  Shape s              -> shapeToGRegion loc sca s
  Translate (u,v) r -> regToGReg (x+u, y+v) sca r
  Scale (u,v) r        -> regToGReg loc (a*u, b*v) r
  Empty                -> createRectangle (0,0) (0,0)
  r1 `Union` r2        -> primGReg loc sca r1 r2 orRegion
  r1 `Intersect` r2 -> primGReg loc sca r1 r2 andRegion
  Complement r         -> primGReg loc sca winRect r diffRegion
    where winRect :: Region
          winRect = Shape (Rectangle
                     (pixelToInch xWin) (pixelToInch yWin))
regionToGRegion :: Region -> G.Region
regionToGRegion r = regToGReg (0,0) (1,1) r
```

Pattern renaming

Shape to G.Region: Rectangle

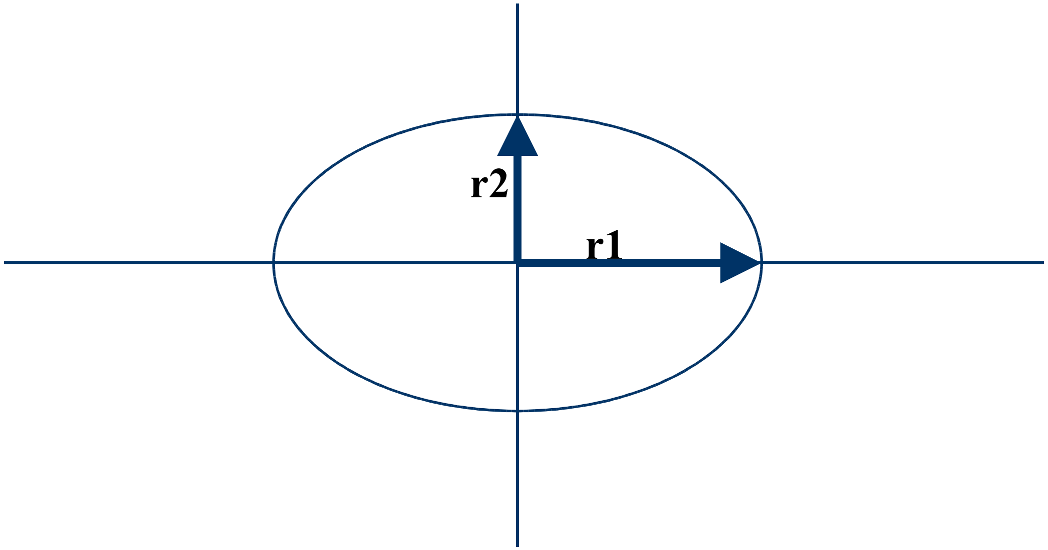
```
shapeToGRegion1
  :: Vector -> Vector -> Shape -> G.Region
shapeToGRegion1 (lx,ly) (sx,sy) (Rectangle s1 s2)
  = createRectangle (trans(-s1/2,-s2/2)) (trans (s1/2,s2/2))
    where trans (x,y) = ( xWin2 + inchToPixel ((x+lx)*sx),
                          yWin2 - inchToPixel ((y+ly)*sy) )
```





Ellipse

```
shapeToGRegion1 (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) )
```



Polygon and RtTriangle

```
shapeToGRegion1 (lx,ly) (sx,sy) (Polygon pts)
  = createPolygon (map trans pts)
    where trans (x,y) =
                ( xWin2 + inchToPixel ((x+lx)*sx),
                  yWin2 - inchToPixel ((y+ly)*sy) )
shapeToGRegion1 (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) )
```

A Matter of Style, 2

- shapeToGRegion1 has the same problems as regToGReg1
  - The extra parameters obscure the pattern matching.
  - There is a repeated pattern: we should give it a name.

```
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 pts      -> createPolygon (map trans pts)
  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) )
```

Drawing Pictures, Sample Regions

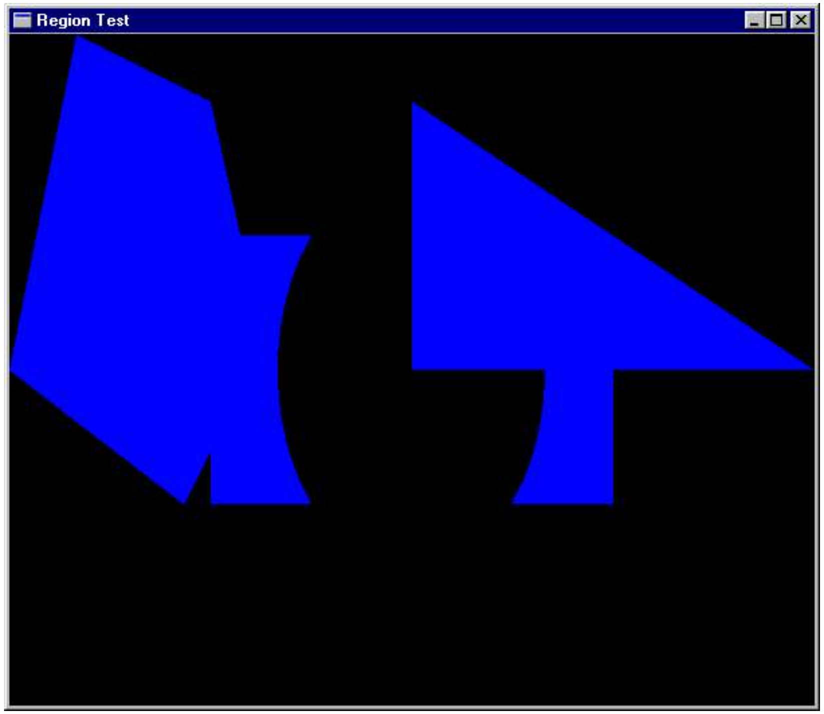
```
draw :: Picture -> IO ()
draw p = runGraphics (
         do w <- openWindow "Region Test" (xWin,yWin)
            drawPic w p
            spaceClose w
         )

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)] )
```

Sample Pictures

```
reg1 = r3 `Union`                     -- RtTriangle
       r1 `Intersect`                 -- Rectangle
       Complement r2 `Union`          -- Ellipse
       r4                             -- Polygon
```

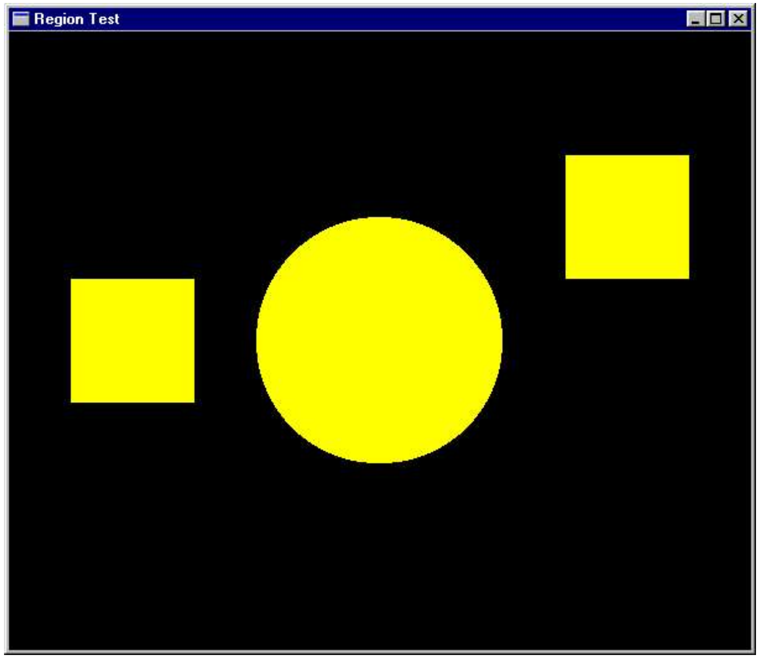
```
pic1 = Region Cyan reg1
Main1 = draw pic1
```



Recall the precedence of Union and Intersect

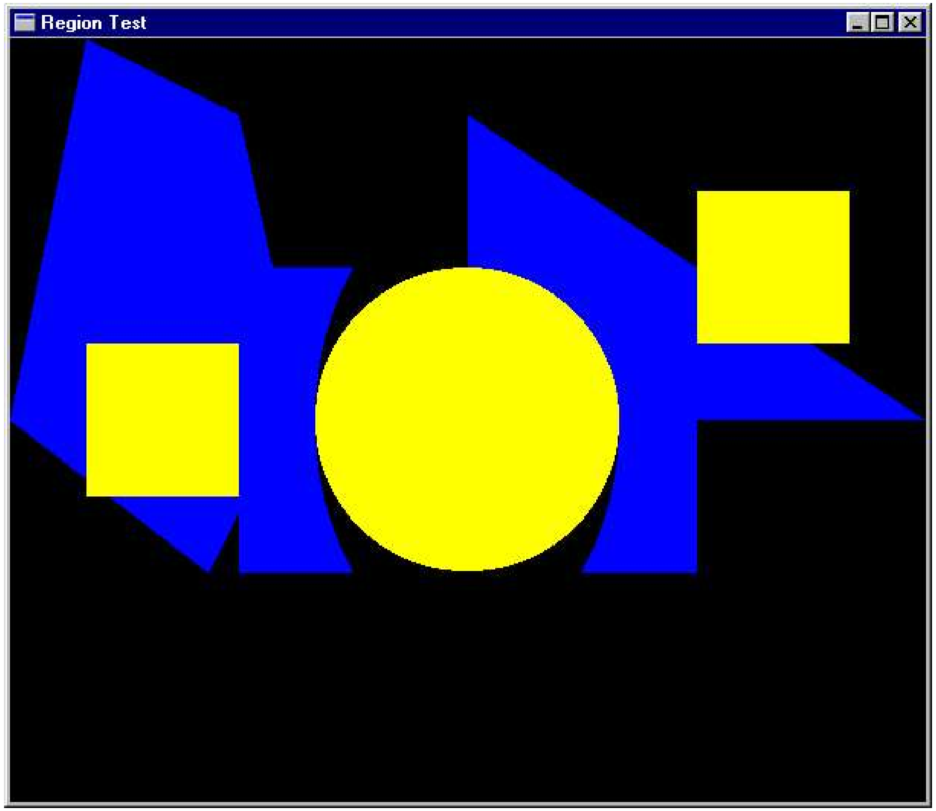
More Pictures

```
reg2 = let circle = Shape (Ellipse 0.5 0.5)
           square = Shape (Rectangle 1 1)
       in (Scale (2,2) circle)
          `Union` (Translate (2,1) square)
          `Union` (Translate (-2,0) square)
pic2 = Region Yellow reg2
main2 = draw pic2
```



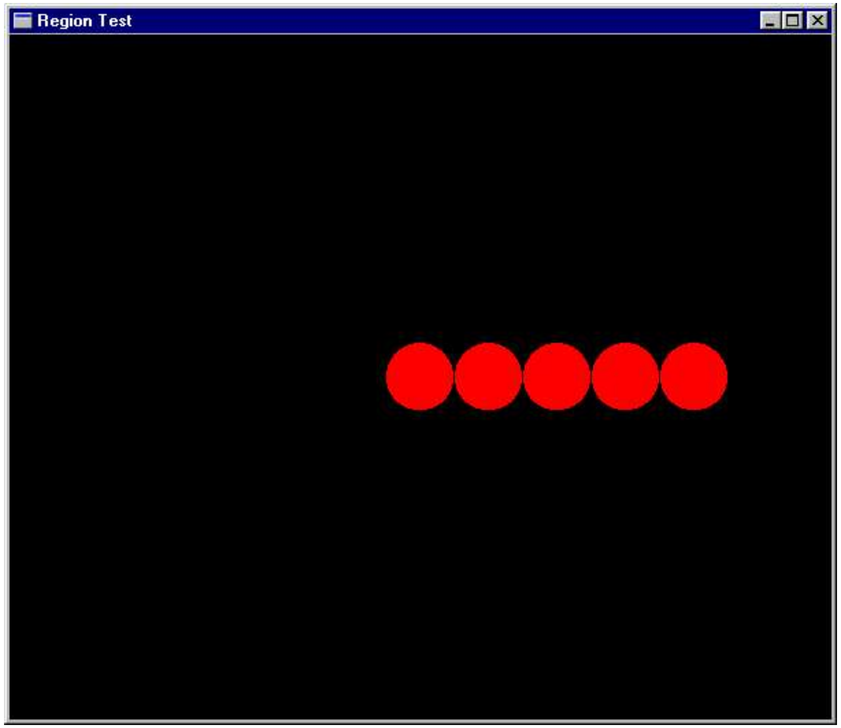
Another Picture

```
pic3 = pic2 `Over` pic1
main3 = draw pic3
```



Separate Computation From Action

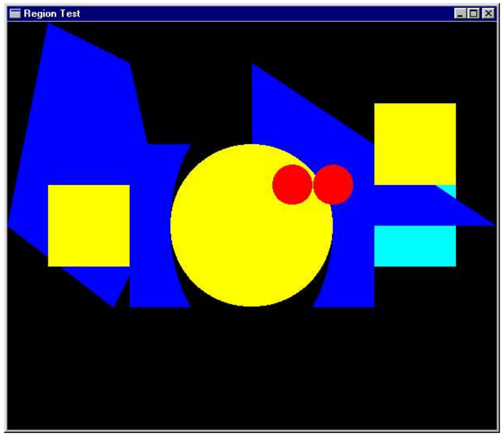
```
oneCircle   = Shape (Ellipse 1 1)
manyCircles
  = [ Translate (x,0) oneCircle | x <- [0,2..] ]
fiveCircles =
  foldr Union Empty
        (take 5 manyCircles)
pic4 = Region Magenta
        (Scale (0.25,0.25)
               fiveCircles)
main4 = draw pic4
```



Ordering Pictures

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

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




```
pic6 = pic4 `Over` pic2 `Over` pic1 `Over` pic5
pictToList pic6 --->
     [(Magenta,?), (Yellow,?),(Cyan,?),(Cyan,?)]
```

Recovers the Regions from top to bottom.

Possible because Picture is a datatype that can be analyzed.

Two ways of drawing a picture

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

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 ()
```

- Something interesting to prove:

```
drawPic w = sequence .
            (map (uncurry (drawRegionInWindow w))) .
            reverse .
            pictToList
```

Pictures that React

- Find the topmost Region in a Picture that "covers" the position of the mouse when a left button click appears.
- Search the picture list for the first Region that contains the mouse position.
- Re-arrange the list, bringing that one to the top.

```
adjust :: [(Color,Region)] -> Vertex ->
          (Maybe (Color,Region), [(Color,Region)])
adjust []           p = (Nothing, [])
adjust ((c,r):regs) p =
   if r `containsR` p
      then (Just (c,r), regs)
      else let (hit, rs) = adjust regs p
           in  (hit, (c,r) : rs)
```

Doing it Non-recursively

```
adjust2 regs p
  = case (break (\(_,r) -> r `containsR` p) regs)
    of
      (top,hit:rest) -> (Just hit, top++rest)
      (_,[])         -> (Nothing, [])
```

This is from the Prelude:

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

For example:

```
break even [1,3,5,4,7,6,12] = ([1,3,5],[4,7,6,12])
```

Putting it all Together

```
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)

draw2 :: Picture -> IO ()
draw2 pic = runGraphics (
           do w <- openWindow "Picture demo" (xWin,yWin)
              loop w (pictToList pic))
```

Try it Out

```
p1,p2,p3,p4 :: Picture
p1 = Region Magenta r1
p2 = Region Cyan r2
p3 = Region Green r3
p4 = Region Yellow r4

pic :: Picture
pic = foldl Over EmptyPic [p1,p2,p3,p4]
main = draw2 pic
```

A Matter of Style, 3

```
loop2 w regs
    = do clearWindow w
         sequence [ drawRegionInWindow w c r |
                    (c,r) <- reverse regs ]
         (x,y) <- getLBP w
         let aux (_,r) = r `containsR`
                         ( pixelToInch (x-xWin2),
                           pixelToInch (yWin2-y) )
         case (break aux regs) of
           (_,[])         -> closeWindow w
           (top,hit:bot) -> loop w (hit : (top++bot))
draw3 pic = runGraphics (
              do w <- openWindow "Picture demo" (xWin,yWin)
                 loop2 w (pictToList pic)
              )
```