Advanced Programming Handout 6

Functional Animation

Homework Preview…

- The main part of this week's homework assignment will be to write your own animation based on SOE chapter 13.
- There will also be a more structured warm-up exercise.
- Make sure that you can run the SOE graphics demos.

Teams

- This assignment, and most likely all the rest, will be carried out in teams of two.
- We'll finalize teams in class on Wednesday. (Pair up sooner if you like.)
- Team members can work together on at most one weekly assignment and one of the two larger projects.

Pair programming

- Programming in teams of two is strongly advocated by proponents of "Extreme Programming" (and its many variants)
- Rules:
  - All programming sessions are "shoulder to shoulder": two people at one screen
  - Both must understand and agree with every line of code
  - Switch drivers from time to time

Disadvantages of Pair Programming

- Coordination overhead
  - Have to get two people physically together to do anything
- Slower
  - Uses two people to do one person's job

Advantages of Pair Programming

- Dramatic increase in code quality
  - Verbalizing ideas leads to deeper understanding
  - Discourages quick hacks
  - Result is often better than either programmer could have achieved *even by spending twice as long!*

Advantages of Pair Programming

- Not that much slower
  - Fewer thinkos ---> much less time spent in debugging
  - Earlier detection of design errors ---> much less time spent in massive reorganizations
  - People's energy levels have different cycles
- Continual opportunities to hone skills and learn new tricks

Perimeters of Shapes

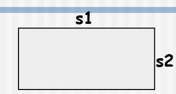
(SOE Chapter 6)

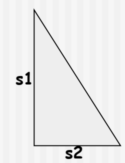
Shapes

```
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)
```

The Perimeter of a Shape





- To compute the perimeter we need a function with four equations (1 for each `Shape` constructor).
- The first three are easy …

```
perimeter :: Shape -> Float
perimeter (Rectangle  s1 s2) = 2*(s1+s2)
perimeter (RtTriangle s1 s2) =
                      s1 + s2 + sqrt (s1^2+s2^2)
perimeter (Polygon pts)       =
                      foldl (+) 0 (sides pts)
                      -- or: sumList (sides pts)
```

- This assumes that we can compute the lengths of the sides of a polygon. This shouldn't be too difficult since we can compute the distance between two points with `distBetween`.

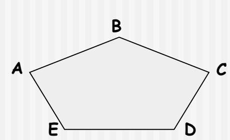
Recursive Def'n of `Sides`

```
sides        :: [Vertex] -> [Side]
sides  []     = []
sides (v:vs) = aux v vs
  where
    aux v1 (v2:vs') = distBetween v1 v2 : aux v2 vs'
    aux vn []       = distBetween vn v  : []

    -- i.e. aux vn [] = [distBetween vn v]
```

- But can we do better? Can we remove the direct recursion, as a seasoned functional programmer might?

Visualize What's Happening



- The list of vertices is: `vs = [A,B,C,D,E]`
- We need to compute the distances between the pairs of points `(A,B)`, `(B,C)`, `(C,D)`, `(D,E)`, and `(E,A)`.
- Can we compute these pairs as a list?
  `[(A,B),(B,C),(C,D),(D,E),(E,A)]`
- Yes, by "zipping" the two lists:
  `[A,B,C,D,E]` and `[B,C,D,E,A]`
  as follows:
  `zip vs (tail vs ++ [head vs])`

New Version of `sides`

This leads to:

```
sides    :: [Vertex] -> [Side]
sides vs = zipWith distBetween vs
                   (tail vs ++ [head vs])
```

Perimeter of an Ellipse

There is one remaining case: the *ellipse*. The perimeter of an ellipse is given by the summation of an infinite series. For an ellipse with radii r_1 and r_2:

$$p = 2\pi r_1(1 - \Sigma\, s_i)$$

where $s_1 = 1/4\ e^2$

$$s_i = s_{i-1} \frac{(2i-1)(2i-3)\ e^2}{4i^2} \quad \text{for } i > 1$$

$$e = \text{sqrt}\ (r_1^2 - r_2^2)\ /\ r_1$$

Given s_i, it is easy to compute s_{i+1}.

Computing the Series

```
nextEl:: Float -> Float -> Float -> Float
nextEl e s i = s*(2*i-1)*(2*i-3)*(e^2) / (4*i^2)
```

$$s_{i+1} = s_i \frac{(2i-1)(2i-3)\ e^2}{4i^2}$$

Now we want to compute `[s1,s2,s3, …]`.
To fix `e`, let's define:

```
aux s i = nextEl e s i
```

So, we would like to compute:

```
[s1,
 s2 = aux s1 2,
 s3 = aux s2 3 = aux (aux s1 2) 3,
 s4 = aux s3 4 = aux (aux (aux s1 2) 3) 4,
 ...
]
```

Can we capture this pattern?

Scanl (scan from the left)

- Yes, using the predefined function `scanl:`

```
scanl :: (a -> b -> b) -> b -> [a] -> [b]
scanl f seed  []     = seed : []
scanl f seed (x:xs) = seed : scanl f newseed xs
      where newseed =  f x seed
```

- For example:

```
scanl (+) 0 [1,2,3]
➔ [ 0,
    1 = (+) 0 1,
    3 = (+) 1 2,
    6 = (+) 3 3 ]
➔ [ 0, 1, 3, 6 ]
```

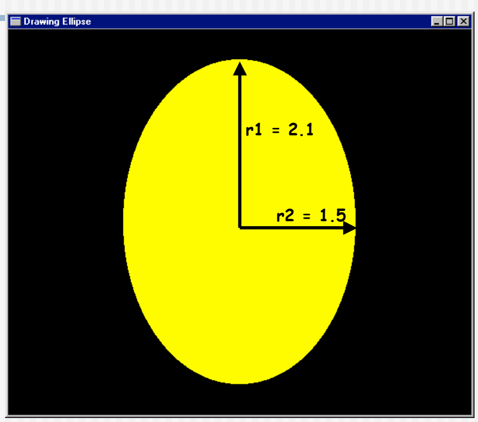
- Using `scanl`, the result we want is:

```
scanl aux s1 [2 ..]
```

Sample Series Values

```
[s1 = 0.122449,
 s2 = 0.0112453,
 s3 = 0.00229496,
 s4 = 0.000614721,
 s5 = 0.000189685,
 ...]
```

Note how quickly the values in the series get smaller …



Putting it all Together

```
perimeter (Ellipse r1 r2)
   | r1 > r2    = ellipsePerim r1 r2
   | otherwise = ellipsePerim r2 r1
   where ellipsePerim r1 r2
           = let e = sqrt (r1^2 - r2^2) / r1
                 s = scanl aux (0.25*e^2)
                           (map intToFloat [2..])
                 aux s i = nextEl e s i
                 test x = x > epsilon
                 sSum = foldl (+) 0 (takeWhile test s)
             in 2*r1*pi*(1 - sSum)
```

A Module of Regions

SOE Chapter 8

The Region Data Type

- A *region* is an area on the two-dimensional Cartesian plane.
- It is represented by a tree-like data structure.

```
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

type Vector = (Float, Float)
```

Questions about Regions

- What is the strategy for writing functions operating on regions?
- Is there a fold-function for regions?
  - How many parameters does it have?
  - What is its type?
- Can one define infinite regions?
- *What does a region mean?*

Sets and Characteristic Functions

- How can we represent an infinite set in Haskell? E.g.:
  - the set of all even numbers
  - the set of all prime numbers
- We could use an infinite list, but then searching it might take a long time! (Membership becomes semi-decidable.)
- The *characteristic function* for a set containing elements of type `z` is a function of type `z -> Bool` that indicates whether or not a given element is in the set. Since that information completely characterizes a set, we can use it to represent a set:

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

- For example:

```
even   :: Set Integer          -- i.e., Integer -> Bool
even x = (x `mod` 2) == 0
```

Combining Sets

- If sets are represented by characteristic functions, then how do we represent the:
  - *union* of two sets?
  - *intersection* of two sets?
  - *complement* of a set?
- In-class exercise – define the following Haskell functions:

```
union      s1 s2 =
intersect s1 s2 =
complement    s   =
```

- We will use these later to define similar operations on regions.

Semantics of Regions

The "meaning" (or "denotation") of a region can be expressed as its characteristic function -- i.e.,

a region denotes the set of points contained within it.

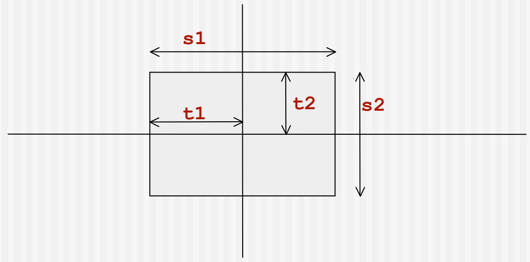
Characteristic Functions for Regions

- We define the meaning of regions by a function:
```
containsR :: Region -> Coordinate -> Bool
type Coordinate = (Float, Float)
```
- Note that `containsR r :: Coordinate -> Bool`, which is a characteristic function. So `containsR` "gives meaning to" regions.
- Another way to see this:
```
containsR :: Region -> Set Coordinate
```
- We can define `containsR` recursively, using pattern matching over the structure of a `Region`.
- Since the base cases of the recursion are primitive shapes, we also need a function that gives meaning to primitive shapes; we will call this function `containsS`.

Let's define `containsS` first…

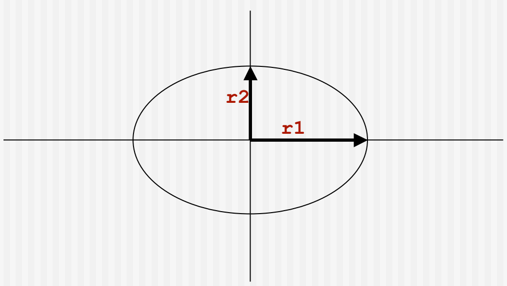
Rectangle

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



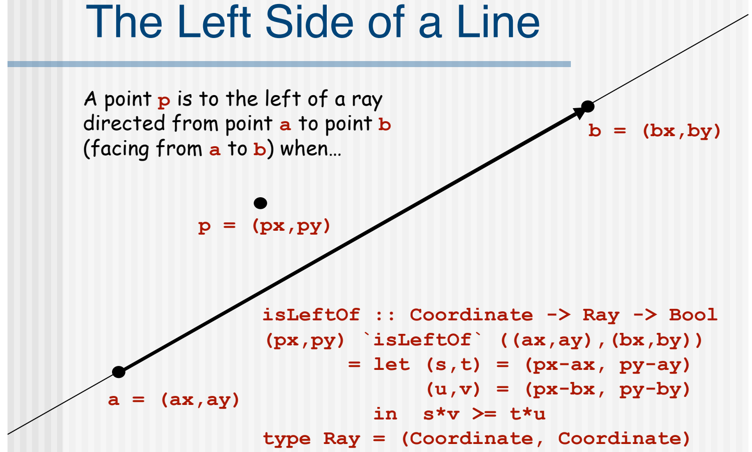
Ellipse

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



The Left Side of a Line

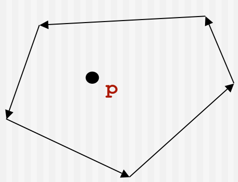
A point **p** is to the left of a ray directed from point **a** to point **b** (facing from **a** to **b**) when…




```
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
type Ray = (Coordinate, Coordinate)
```

Polygon

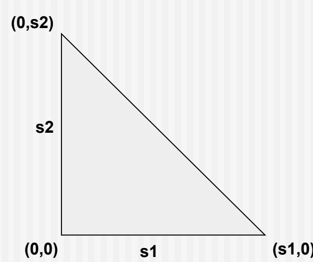
A point **p** is contained within a (convex) polygon if it is to the left of every side, when the vertices are oriented in counter-clockwise order.




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

Right Triangle

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



Putting it all Together

```
containsS :: Shape -> Vertex -> Bool
Rectangle s1 s2 `containsS` (x,y)
   = let t1 = s1/2; t2 = s2/2
     in -t1<=x && x<=t1 && -t2<=y && y<=t2
Ellipse r1 r2 `containsS` (x,y)
   = (x/r1)^2 + (y/r2)^2 <= 1
Polygon pts `containsS` p
   = let shiftpts    = tail pts ++ [head pts]
         leftOfList = map isLeftOfp (zip pts shiftpts)
         isLeftOfp p' = isLeftOf p p'
     in and leftOfList
RtTriangle s1 s2 `containsS` p
   = Polygon [(0,0),(s1,0),(0,s2)] `containsS` p
```

Defining `containsR`

```
containsR :: Region -> Vertex -> Bool
Shape s `containsR` p
         = s `containsS` p
Translate (u,v) r `containsR` (x,y)
         = r `containsR` (x-u,y-v)
Scale (u,v) r      `containsR` (x,y)
         = r `containsR` (x/u,y/v)
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
```

Applying the Semantics

Having defined the meanings of regions, what can we use them for?

- In Chapter 10, we use the `containsR` function to test whether a mouse click falls within a region.
- We can also use the interpretation of regions as characteristic functions to reason about abstract properties of regions. E.g., we can show (by calculation) that `Union` is commutative, in the sense that:

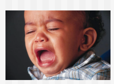
  for any regions `r1` and `r2` and any vertex `p` ,

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

  (and vice versa)

  This is very cool: Instead of having a separate "program logic" for reasoning about properties of programs, we can prove many interesting properties directly by *calculation* on Haskell program texts.

Unfortunately, we will not have time to pursue this topic further this semester.



Drawing Regions

(SOE Chapter 10)

Pictures

- Drawing Pictures
  - Pictures are composed of Regions (which are composed of Shapes)
  - Pictures add color and layering

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

Digression on Importing

- We need to use SOE for drawing things on the screen, but SOE has its own Region datatype, leading to a name clash when we try to import both SOE and our Region module.
- We can work around this as follows:

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

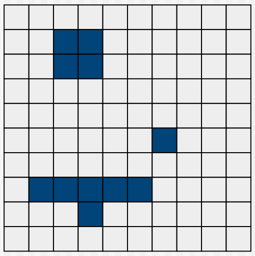
- The effect of these declarations is that all the names from SOE *except* Region can be used in unqualified form, and we can say G.Region to refer to the one from SOE.

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 to do efficiently. Fortunately, the `G.Region` interface uses lower-level support to do this for us.

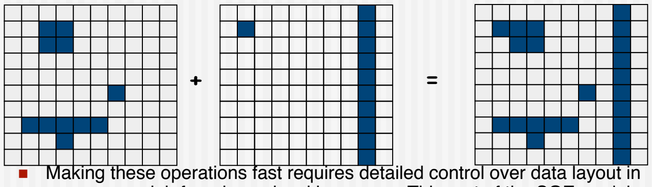
G.Region



- 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:



- Making these operations fast requires detailed control over data layout in memory -- a job for a lower-level language. This part of the SOE module is therefore just a "wrapper" for an external library (probably written in C or C++).

G.Region Interface


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

andRegion       :: G.Region -> G.Region -> IO G.Region
orRegion        :: G.Region -> G.Region -> IO G.Region
xorRegion       :: G.Region -> G.Region -> IO G.Region
diffRegion      :: G.Region -> G.Region -> IO G.Region
deleteRegion    :: G.Region -> IO ()

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

These functions are defined in the SOE library module.

Drawing G.Region

- To render things involving intersections and unions quickly, we perform these calculations in a `G.Region`, then turn the `G.Region` into a graphic object, and then use the machinery we have seen in earlier chapters to display the object.

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

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

- To finish this off, we still need to define `regionToGRegion`.
- But first let's complete the big picture by writing the (straightforward) function that uses `drawRegionInWindow` to draw Pictures.

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`.

Now back to the code for rendering Regions as G.Regions...

Turning a Region into a G.Region

Let's first experiment with a simplified variant of the problem to illustrate an efficiency issue...

```
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)
  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)
```

instead of G.Region

omitting cases for other Region constructors

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 `regToNReg` perform over the `Region` tree?

We've Seen This Before

- We have encountered this problem before in a different setting. Recall the naive 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 = loop xs []
  where loop [] zs     = zs
        loop (x:xs) zs = loop xs (x:zs)
```

N.b.: A good compiler (like GHC) really will implement this function call as a jump!

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

Accumulating the Scaling Factor

```
regToNReg2 :: Region -> NewRegion
regToNReg2 r = rToNR (1,1) r
  where 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
```

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

Final Version

accumulated scaling

accumulated translation

```
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) sca r
regToGReg loc sca Empty
  = createRectangle (0,0) (0,0)
regToGReg loc sca (r1 `Union` r2)
  = let gr1 = regToGReg loc sca r1
        gr2 = regToGReg loc sca r2
    in orRegion gr1 gr2
```

To finish, we need to write similar clauses for `Intersect`, `Complement` etc. and define

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

A Matter of Style

- While the function on the previous page does the job correctly, there are several stylistic issues that could make it more readable and understandable.
- For one thing, the style of defining a function by patterns becomes cluttered when there are many parameters (other than the one which has the patterns).
- For another, 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.

Abstracting Out a Common Pattern

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

Case Expressions

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

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

Pattern renaming

A Region representing the whole graphics window

Drawing Pictures

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

A Better Definition

```
($) :: (a->b) -> a -> b
f ($) x = f x

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

In effect, we've introduced a second syntax for application, with lower precedence than the standard one

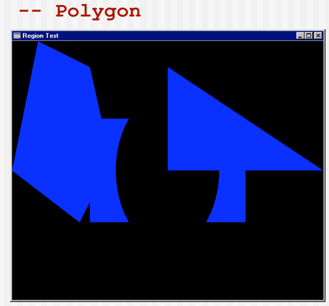
Some Sample Regions

```
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

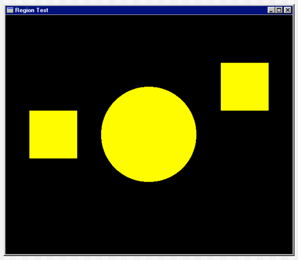
```
reg = r3 `Union`             -- RtTriangle
      (r1 `Intersect`        -- Rectangle
      Complement r2 `Union`  -- Ellipse
      r4)                    -- Polygon

pic1 = Region Cyan reg
Main1 = draw pic1
```



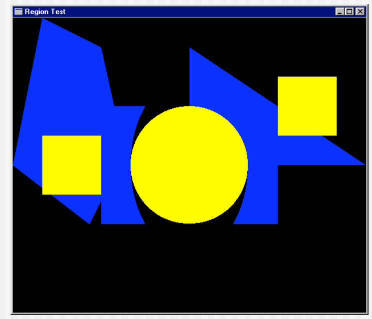
More Pictures

```
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
```



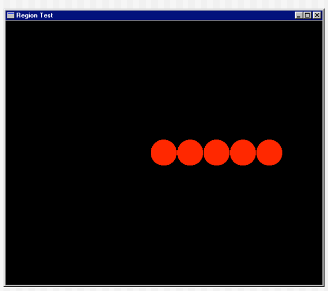
Another Picture

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



Separating Computation From Action

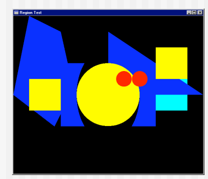
```
oneCircle   = Shape (Ellipse 1 1)
manyCircles = [ Translate (x,0) oneCircle | x <- [0,2..] ]
fiveCircles = foldr Union Empty (take 5 manyCircles)
pic4 = Region Magenta
       (Scale (0.25,0.25)
              fiveCircles)
main4 = draw pic4
```



Ordering Pictures

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

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



Lists the Regions in a Picture from top to bottom.

(Note that this is possible because Picture is a datatype that can be analyzed. Would not work with, e.g., a characteristic function representation.)

A Suggestive Analogy

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

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

We'll have (much) more to say about this next week...

Pictures that React

- Goal: Find the topmost Region in a Picture that "covers" the position of the mouse when the left button is clicked.
- Implementation: Search the picture (represented as a list) for the first Region that contains the mouse position.
- Then (just for fun) re-arrange the list, bringing that one to the top.

```
adjust :: [(Color,Region)] -> Vertex ->
          (Maybe(Color,Region), [(Color,Region)])
               selected picture          reordered list
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

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])
```

So:

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

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)
```

A Matter of Style, Redux

```
loop2 w regs
    = do clearWindow w
         sequence [ drawRegionInWindow w c r |
                    (c,r) <- reverse regs ]
         (x,y) <- getLBP w
         let (px,py) = (pixelToInch (x-xWin2),
                        pixelToInch (yWin2-y))
         let testHit (_,r) = r `containsR` (px,py)
         case (break testHit 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)
```

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 = draw3 pic
```

A Module of Simple Animations

SOE Chapter 13

Motivation

- In the abstract, an *animation* is a continuous, time-varying image.
- In practice, it is a sequence of static images displayed in succession so rapidly that it looks continuous.
- Our goal is to present to the programmer an abstract view of animations that hides the practical details.
- In addition, we will generalize animations to be continuous, time-varying quantities of *any* value, not just images.

Representing Animations

- As usual, we will use our most powerful tool, *functions,* to represent animations:

```
type Animation a = Time -> a
type Time = Float
```

- Examples:

```
rubberBall :: Animation Shape
rubberBall t = Ellipse (sin t) (cos t)

revolvingBall :: Animation Region
revolvingBall t = let ball = Shape (Ellipse 0.2 0.2)
                  in Translate (sin t, cos t) ball

planets :: Animation Picture
planets t =  let p1 = Region Red (Shape (rubberBall t))
                 p2 = Region Yellow (revolvingBall t)
             in p1 `Over` p2

tellTime :: Animation String
tellTime t = "The time is: " ++ show t
```

An Animator

- Suppose we had a function:

```
animate :: String -> Animation Graphic -> IO ( )
```

- We could then execute (display) the previous animations. For example:

```
main1 :: IO ( )
main1 = animate "Animated Shape"
                (withColor Blue . shapeToGraphic .
                 rubberBall)

main2 :: IO ( )
main2 = animate "Animated Text"
                (text (100,200) . tellTime)
```

Definition of "animate"

```
animate :: String -> Animation Graphic -> IO ( )

animate title anim = runGraphics $
  do w <- openWindowEx title (Just (0,0)) (Just
  (xWin,yWin))

  drawBufferedGraphic (Just 30)
      t0 <- timeGetTime
      let loop =
        do t <- timeGetTime
        let ft = intToFloat (word32ToInt (t-t0)) / 1000
        setGraphic w (anim ft)
        getWindowTick w
        loop
      loop
```

See text for details...

Common Operations

- We can define many operations on animations based on the underlying type. For example, for Pictures:

```
emptyA :: Animation Picture
emptyA t = EmptyPic
overA :: Animation Picture
         -> Animation Picture
         -> Animation Picture
overA a1 a2 t = a1 t `Over` a2 t
overManyA :: [Animation Picture] -> Animation Picture
overManyA = foldr overA emptyA
```

- We can do a similar thing for Shapes, etc.
- Also, for numeric animations, we could define functions like addA, multA, and so on.
- But there is a better way... **Type Classes!!**

...naturally

Behaviors

- Preliminary definition:

```
newtype Behavior a  =  Beh (Time -> a)
```

- Here `newtype` creates a single-argument datatype with (time and space) efficiency the same as a simple `type` declaration.

(So what is the difference??)

Behaviors

- We need to use `newtype` here because type synonyms are not allowed in type class instance declarations

Numeric Animations

```
instance Num a =>
         Num (Behavior a) where
    (+) = lift2 (+)
    (*) = lift2 (*)
    negate = lift1 negate
    abs = lift1 abs
    signum = lift1 signum
    fromInteger = lift0 . fromInteger

instance Fractional a =>
           Fractional (Behavior a)
where
    (/) = lift2 (/)
    fromRational = lift0 . fromRational
```

```
instance Floating a =>
       Floating (Behavior a)
where
pi       = lift0 pi
sqrt     = lift1 sqrt
exp      = lift1 exp
log      = lift1 log
sin      = lift1 sin
cos      = lift1 cos
tan      = lift1 tan
etc.
```

...where the lifting functions are defined by:

```
  lift0 :: a -> Behavior a
  lift0 x = Beh (\t -> x)

  lift1 :: (a -> b) -> (Behavior a -> Behavior b)
  lift1 f (Beh a) = Beh (\t -> f (a t))

  lift2 :: (a -> b -> c) -> (Behavior a -> Behavior b -> Behavior c)
  lift2 g (Beh a) (Beh b) = Beh (\t -> g (a t) (b t))
```

Type Class Magic

- Furthermore, define *time* by:

```
time :: Behavior Time
time = Beh (\t -> t)
```

- For example, consider "time + 42":

```
time + 42
```

➔ unfold overloaded defs of time, (+), and 42

```
(lift2 (+)) (Beh (\t -> t)) (Beh (\t -> 42))
```

➔ `unfold lift2`

```
(\ (Beh a) (Beh b) -> Beh (\t -> a t + b t) )
      (Beh (\t -> t))
      (Beh (\t -> 42))
```

➔ unfold anonymous function

```
Beh (\t -> (\t -> t) t + (\t -> 42) t )
```

➔ unfold two anonymous functions

```
Beh (\t -> t + 42)
```

- The magic of type classes!!

New Type Classes

- In addition to using existing type classes such as `Num`, we can define new ones. For example:

```
class Combine a where
      empty  :: a
      over   :: a -> a -> a

instance Combine Picture where
      empty  = EmptyPic
      over   = Over

instance Combine a => Combine (Behavior a) where
      empty  = lift0 empty
      over   = lift2 over

overMany :: Combine a => [a] -> a
overMany = foldr over empty
```

Hiding More Detail

- We have not yet hidden all the "practical" detail – in particular, *time* itself.
- But through more aggressive lifting...

```
reg     = lift2 Region
shape   = lift1 Shape
ell     = lift2 Ellipse
red     = lift0 Red
yellow  = lift0 Yellow
translate (Beh a1, Beh a2) (Beh r)          -- note complexity here
        = Beh (\t -> Translate (a1 t, a2 t) (r t))
```

we can redefine the red revolving ball as follows:

```
revolvingBallB :: Behavior Picture
revolvingBallB =
    let ball = shape (ell 0.2 0.2)
   in reg red (translate (sin time, cos time) ball)
```

More Liftings

- Comparison operators:

```
(>*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
(>*) = lift2 (>)
```

- Conditional behaviors:

```
ifFun :: Bool -> a -> a -> a
ifFun p c a = if p then c else a

cond :: Behavior Bool -> Behavior a -> Behavior a
         -> Behavior a
cond = lift3 ifFun
```

- For example, a flashing color:

```
flash :: Behavior Color
flash = cond (sin time >* 0) red yellow
```

Time Travel

- A function for translating a behavior through time:

```
timeTrans :: Behavior Time -> Behavior a -> Behavior a
timeTrans (Beh f) (Beh a) = Beh (a . f)
```

- For example:

```
timeTrans (2*time) anim                  -- double speed
timeTrans (5+time) anim `over` anim      -- one anim 5 sec
                                            behind another
timeTrans (negate time) anim             -- go backwards
```

- Any kind of animation can be time transformed:

```
flashingBall :: Behavior Picture
flashingBall =
    let ball = shape (ell 0.2 0.2)
   in reg (timeTrans (8*time) flash)
             (translate (sin time, cos time) ball)
```

Final Example

```
revolvingBalls :: Behavior Picture
   revolvingBalls
     = overMany [ timeTrans (time + t*pi/4) flashingBall
                                   | t <- map lift0 [0..7] ]
```

See the text for one other example:
a kaleidoscope program.