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 ---&gt; much less time spent in debugging
- Earlier detection of design errors ---&gt; 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 (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) } as follows:
zip vs (tail vs ++ [head vs])
New Version of \texttt{sides}

This leads to:

\[
\text{sides} \:: \text{[Vertex]} \rightarrow \text{[Side]}
\[
\text{sides} \text{ vs } = \text{zipWith distBetween vs (tail vs ++ [head vs])}
\]

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 \( r_2 \):

\[
p = 2\pi r_1 (1 - \sum s_i)
\]

where \( s_1 = \frac{1}{4} e^2 \)

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

\[
e = \sqrt{\frac{r_1^2 - r_2^2}{r_1}}
\]

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

\text{Scanl (scan from the left)}

\begin{itemize}
  \item Yes, using the predefined function \texttt{scanl}:
  \[
  \text{scanl} :: (a -> b -> b) -> b -> [a] -> [b]
  \]
  \[
  \text{scanl f seed [x:]} = f x seed : \text{scanl f newseed xs}
  \]
  \[
  \text{where newseed } = \text{ f } x \text{ seed}
  \]
  \item For example:
  \[
  \text{scanl (+) 0 [1,2,3] } \Rightarrow [ 0, 1, 3, 6 ] 
  \]
  \item Using \texttt{scanl}, the result we want is:
  \[
  \text{scanl aux s1 [2 ..] }
  \]
\end{itemize}

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

\[
\text{perimeter (Ellipse } r1 \ r2) \quad | \ r1 > r2 \ = \text{ellipsePerim } r1 \ r2 \\
| \ \text{otherwise } = \text{ellipsePerim } r2 \ r1
\]

where \texttt{ellipsePerim}:

\[
\text{let e } = \text{sqrt (r1^2 - r2^2) / r1} \]

\[
s = \text{scanl aux (0.25*e^2)} (\text{map intToFloat [2 ..]})
\]

\[
\text{aux s i } = \text{nextEl e s i}
\]

\[
\text{test x } = \text{x > epsilon}
\]

\[
\text{sSum } = \text{foldl (+) } 0 \ (\text{takeWhile test s})
\]

\[
\text{in 2*r1*pi*(1 - sSum)}
\]
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.

```haskell
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:
  ```haskell
  type Set a = a -> Bool
  ```
- For example:
  ```haskell
  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:
  ```haskell
  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:
  \[
  \text{containsR} :: \text{Region} \rightarrow \text{Coordinate} \rightarrow \text{Bool}
  \]
- Note that \( \text{containsR} \text{r} :: \text{Coordinate} \rightarrow \text{Bool} \), which is a characteristic function. So \( \text{containsR} \) "gives meaning to" regions.
- Another way to see this:
  \[
  \text{containsR} :: \text{Region} \rightarrow \text{Set Coordinate}
  \]
- We can define \( \text{containsR} \) recursively, using pattern matching over the structure of a \( \text{Region} \).
- 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 \( \text{containsS} \).

Let's define \( \text{containsS} \) first...

Rectangle

\[
\text{Rectangle \ s1 \ s2 \ 'containsS' \ (x,y)} = \begin{cases}
  \text{let } t1 = s1/2 \\
  \text{let } t2 = s2/2 \\
  \text{in } -t1 \leq x \land x \leq t1 \land -t2 \leq y \land y \leq t2
\end{cases}
\]

Ellipse

\[
\text{Ellipse } r1 \ r2 \ 'containsS' \ (x,y) = (x/r1)^2 + (y/r2)^2 \leq 1
\]

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...

\[
\text{isLeftOf} :: \text{Coordinate} \rightarrow \text{Ray} \rightarrow \text{Bool}
\]

\[
\text{isLeftOf } (px,py) 'isLeftOf' ((ax,ay),(bx,by)) = \begin{cases}
  \text{let } (s,t) = (px-ax, py-ay) \\
  \text{let } (u,v) = (px-bx, py-by) \\
  \text{in } s*v \geq t*u
\end{cases}
\]

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.

\[
\text{Polygon } \text{pts} \ 'containsS' \ p = \begin{cases}
  \text{let } \text{shiftpts} = \text{tail pts} ++ [\text{head pts}] \\
  \text{let } \text{leftOfList} = \text{map } (\text{isLeftOf } p) (\text{zip pts shiftpts}) \\
  \text{in and } \text{leftOfList}
\end{cases}
\]

Right Triangle

\[
\text{RtTriangle } s1 \ s2 \ 'containsS' \ p = \text{Polygon } [(0,0),(s1,0),(0,s2)] 'containsS' \ p
\]
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

```
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:

\[ (r_1 \cup r_2) \text{`containsR` } p \rightarrow (r_2 \cup r_1) \text{`containsR` } p \]

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)

```
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 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

```haskell
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

- These functions are defined in the SOE library module.

```
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...

```haskell
data NewRegion = Rect Side Side

regToNReg :: Region -> NewRegion
regToNReg (Shape (Rectangle sx sy)) = Rect sx sy
regToNReg (Scale (x,y) r) = regToNReg (Scale (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)
  scaleReg (x,y) (Empty) = Empty
  scaleReg (x,y) (OrRegion r1 r2) = r1 `Union` r2
  scaleReg (x,y) (AndRegion r1 r2) = r1 `Intersection` r2

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 (lx,ly) (Empty) = createRectangle (0,0) (0,0)
regToGReg loc (r1 `Union` r2) = let gr1 = regToGReg loc r1
g2 = regToGReg loc 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 Problem

- Consider

```haskell
(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`:
  ```haskell```
  ```haskell
  reverse [] = []
  reverse (x:xs) = (reverse xs) ++ [x]
  where [] ++ zs = zs
        (y:ys) ++ zs = y : (ys ++ zs)
  ```
  ```haskell
  How did we solve this? We used an extra accumulating parameter:
  ```haskell```
  ```haskell
  reverse xs = loop xs []
  where loop [] zs = zs
        loop (x:xs) zs = loop xs (x:zs)
  ```
  ```haskell
  We can do the same thing for `Regions`.
  ```haskell
  ```

- A good compiler (like GHC) really will implement this function call as a jump!

Accumulating the Scaling Factor

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

```
```

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

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

Case Expressions

```haskell
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+v*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
```

Drawing Pictures

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

A Better Definition

```haskell
($) :: (a->b) -> a -> b
f ($) 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

```haskell
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

```haskell
reg = r3 `Union` (r1 `Intersect` Complement r2 `Union` r4) -- RtTriangle
      -- Rectangle
      -- Ellipse
      -- Polygon
pic1 = Region Cyan reg
Main1 = draw pic1
```

A Region representing the whole graphics window
More Pictures

\[
\text{reg2 = let circle = Shape (Ellipse 0.5 0.5)} \\
\text{    square = Shape (Rectangle 1 1)} \\
\text{    in (Scale (2,2) circle)} \\
\text{    'Union' (Translate (2,1) square) \\
\text{    'Union' (Translate (-2,0) square) \\
\text{pic2 = Region Yellow reg2 \\
\text{main2 = draw pic2}}\]
\]

Another Picture

\[
\text{pic3 = pic2 `Over` pic1 \\
\text{main3 = draw pic3}}\]

Separating Computation From Action

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

Ordering Pictures

\[
\text{pictToList :: Picture -> [(Color,Region)]} \\
\text{pictToList EmptyPic = []} \\
\text{pictToList (Region c r) = [(c,r)]} \\
\text{pictToList (p1 `Over` p2) = pictToList p1 ++ pictToList p2 \}
\]

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

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

\[
\text{drawPic w EmptyPic = return ()} \\
\text{drawPic w (Region c r) = drawRegionInWindow w c r} \\
\text{drawPic w (p1 `Over` p2) = do drawPic w p2 \\
\text{    drawPic w p1 \\
\text{We'll have (much) more to say about this next week...}}\]

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.

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

We'll have (much) more to say about this next week...
### Doing it Non-recursively

From the Prelude:

```haskell
define `break`:: (a -> Bool) -> [a] -> ([a],[a])
For example:

```halve even [1,3,5,4,7,6,12] ➔ ([1,3,5],[4,7,6,12])

So:

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

### Putting it all Together

```haskell
loop `win` -> [Color,Region] -> IO ()
loop `win` `reg` =
    do `clearWindow` `win`
    sequence [ `drawRegionInWindow` `win` `c` `r` |
                (c, `r`) <- reverse `reg` ]
    (x, y) <- `getLBP` `win`
    let (px, py) = (pixelToInch (x - `xWin2`),
                    pixelToInch (yWin2 - y))
    let `testHit` (_, `r`) = `r` `containsR` (px, py)
    case (break `testHit` `reg`) of
        (Nothing, []) ➔ `closeWindow` `win`
        (top, `hit`: `bot`) ➔ loop `win` (hit : `bot`
            `reg`)
```

### A Matter of Style, Redux

```haskell
loop2 `win` `reg` =
    do `clearWindow` `win`
    sequence [ `drawRegionInWindow` `win` `c` `r` |
                (c, `r`) <- reverse `reg` ]
    (x, y) <- `getLBP` `win`
    let (px, py) = (pixelToInch (x - `xWin2`),
                    pixelToInch (yWin2 - y))
    let `testHit` (_, `r`) = `r` `containsR` (px, py)
    case (break `testHit` `reg`) of
        (Nothing, []) ➔ `closeWindow` `win`
        (top, `hit`: `bot`) ➔ loop2 `win` (hit : `bot`
            `reg`)
```

### Try it Out

```haskell
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
```

### 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:
  
  ```haskell
  type Animation a = Time -> a
  type Time = Float
  ```

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

- We could then execute (display) the previous animations. For example:
  ```haskell
  main1 :: IO ()
  main1 = animate "Animated Shape"
          (withColor Blue . shapeToGraphic . rubberBall)

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

 Definition of “animate”

```haskell
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...

Behaviors

- Preliminary definition:
  ```haskell
  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??)

Common Operations

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

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
  fromIntegral = lift0 . fromIntegral

instance Fractional a =>
  Fractional (Behavior a) where
  (/) = lift2 (/)
  fromRational = lift1 fromRational
  signum = lift1 signum
  abs = lift1 abs
  negate = lift1 negate
  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.

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
cond = lift3 ifFun

For example, a flashing color:

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

Type Class Magic

Furthermore, define time by:

time :: Behavior Time

For example, consider "time + 42":

time + 42

unfold two anonymous functions

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
e11 = lift1 Ellipse
red = lift0 Red
yellow = lift0 Yellow

we can redefine the red revolving ball as follows:

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

More Comparison Operators:

(*) :: Ord a => Behavior a -> Behavior a -> Behavior a
(*) = 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
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

d timeTrans (f t) (Beh a) = Beh (f t)

For example:

timeTrans (5*time) 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

```haskell
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.
```