CS 457/557 Functional Programming

Lecture 4
Drawing Shapes
Recall the Shape Datatype

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

type Vertex = (Float,Float)
type Side   = Float
type Radius = Float
```
Properties of Shapes

- Note that some shapes are position independent:
  - Rectangle Side Side
  - RtTriangle Side Side
  - Ellipse Radius Radius

- On the other hand, a Polygon [Vertex] is defined in terms of where it appears in the plane.

- A shape’s Size and Radius are measured in inches.

- On the other hand, the graphics drawing mechanism of Ch. 3 was based on pixels.
Considerations

• Where do we draw position-independent shapes?
  – Randomly?
  – In the upper left corner (the window origin)?
  – In the middle of the window?

• We will choose the last option above, by defining the middle of the window as the origin of a standard Cartesian coordinate system.

• So our new coordinate system has both a different notion of “origin” (middle vs. top-left) and of “units” (inches vs. pixels).

• We will need to define coercions between these two coordinate systems.
Coordinate Systems

Window Coordinate System

Shape Coordinate System

(0,0)  (0,1)  (200,0)

(0,0)  (1,0)

(0,200)  (200,200) pixels

or

(0,1)  (1,-1) inches
Units Coercion

\[
\text{inchToPixel} :: \text{Float} \rightarrow \text{Int} \\
\text{inchToPixel} \ x = \text{round} \ (100 \times x)
\]

\[
\text{pixelToInch} :: \text{Int} \rightarrow \text{Float} \\
\text{pixelToInch} \ n = \text{fromIntegral} \ n \div 100
\]

Note: simpler alternative to book's definition.
Translation Coercion

\[ x_{\text{Win}}, y_{\text{Win}} :: \text{Int} \]
\[ x_{\text{Win}} = 600 \]
\[ y_{\text{Win}} = 500 \]

\[ x_{\text{Win}2}, y_{\text{Win}2} :: \text{Int} \]
\[ x_{\text{Win}2} = x_{\text{Win}} \div 2 \]
\[ y_{\text{Win}2} = y_{\text{Win}} \div 2 \]

\[ \text{trans} :: \text{Vertex} \rightarrow \text{Point} \]
\[ \text{trans} (x, y) = (x_{\text{Win}2} + \text{inchToPixel} x, y_{\text{Win}2} - \text{inchToPixel} y) \]
Translating Points

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

transList :: [Vertex] -> [Point]
transList [] = []
transList (p:ps) = trans p : transList ps

-- or:
transList vs = [trans p | p <- vs]
Translating Shapes

\[
\text{shapeToGraphic} :: \text{Shape} \rightarrow \text{Graphic}
\]

\[
\text{shapeToGraphic} (\text{Rectangle } s_1 \ s_2) = \text{let } s_{12} = s_1/2 \ \\
\quad \quad s_{22} = s_2/2 \ \\
\quad \quad \text{in polygon (transList } [(s_{12},-s_{22}),(-s_{12},s_{22}), \ \\
\quad \quad \quad (s_{12},s_{22}), (-s_{12},-s_{22})])
\]

\[
\text{shapeToGraphic} (\text{Ellipse } r_1 \ r_2) = \text{ellipse (trans } (-r_1,-r_2)) \text{ (trans } (r_1,r_2))
\]

\[
\text{shapeToGraphic} (\text{RtTriangle } s_1 \ s_2) = \text{polygon (transList } [(0,0),(s_1,0),(0,s_2)])
\]

\[
\text{shapeToGraphic} (\text{Polygon } \text{pts}) = \text{polygon (transList } \text{pts})
\]
Some Test Shapes

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)]
Drawing Shapes

```haskell
main10 = runGraphics (  
do w <- openWindow "Drawing Shapes" (xWin,yWin)  
drawInWindow w  
   (withColor Red (shapeToGraphic sh1))  
drawInWindow w  
   (withColor Blue (shapeToGraphic sh2))  
   spaceClose w  
)
```
The Result
Drawing Multiple Shapes

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
        =do withColor c (shapeToGraphic s)
drawShapes w cs
Multiple Shapes, cont’d

```haskell
main11
  = runGraphics ( DO w <- openWindow
                  "Drawing Shapes"
                  (xWin,yWin)
                  drawShapes w shs
                  spaceClose w )
```
Retrospect

- Can distinguish three different types.

```haskell
data Shape = Polygon [Vertex] | ...

  » “Transparent” -- can both construct and pattern match.
  » Represents geometric abstraction (no graphical meaning)
```

```haskell
type Graphic

polygon :: [Point] -> Graphic

  » Abstract type – can construct instances, but not inspect them.
  » Can modify/combine with special operators like `withColor`
  » Expressed in graphics coordinate system.
```

```haskell
type IO ()

drawPolygon :: [Vertex] -> IO ()

  » (We didn't choose to define functions like this.)
  » Even more abstract; can only be sequenced and executed.
```