Weekly Assignment 1

Advanced Programming 2014 @ DIKU

Martin Jørgensen
University of Copenhagen
Department of Computer Science
tzk173@alumni.ku.dk

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

September 7, 2014

Curves.hs

The Point

We've declared the `Point` type using `newtype`, as shown below in figure 1. It inherits the type class `Show`, which enables us to print it to the console.

```
import Text.Printf (printf)
```

Figure 1: `Point` declaration (../Curves.hs)

For ease of constructing points a constructor was defined as shown in figure 2, shown below.

```
  deriving (Show)
```

Figure 2: `Point` constructor (../Curves.hs)

Due to the imprecision of floating-point numbers we've limited the amount of meaningful decimal places to two. This was done by instancing the `Eq` type class for our `Point` type to reflect this, and is shown in figure 3 below.

```

-- Points are considered equal (sharing a location) if the difference between
-- their coordinates are less that 0.01.
```

Figure 3: Instantiating `Eq` for `Point` (../Curves.hs)

Furthermore, as will be discussed later, we will need the `Point` type to adhere to arithmetic operations. As such, we must make it apart of the `Num` type class. This is shown in figure 4.

```

-- Instancing Point under Num to use the + and - operators.
-- */abs/signum is nonsensical but included to shut up the compiler ^_^
instance Num (Point) where
  Point(ax, ay) + Point(bx, by) = Point(ax+bx, ay+by)
  Point(ax, ay) * Point(bx, by) = Point(ax*bx, ay*by) -- <3 Compiler warnings
  Point(ax, ay) - Point(bx, by) = Point(ax-bx, ay-by)
```

Figure 4: Instantiating `Num` for `Point` (../Curves.hs)

The Curve

We'd like to represent a curve in our program, which is essentially a sequence of points. So, we declared the `Curve` type as a list of `Points`. This is shown in figure 5 below.

```
  fromInteger i = Point (fromInteger i, fromInteger i)
```

Figure 5: Declaration of `Curve` (../Curves.hs)

For curves, we also have a convenience constructor function, shown below in figure 6.

```
  deriving (Show, Eq)
```

Figure 6: `Curve` constructor (../Curves.hs)

Manipulation Functions

Now, we turn to describe the ways in which we can manipulate these curves.

Connecting Curves

Connecting curves $a =< a_1, \ldots, a_n >$ and $b =< b_1, \ldots, b_m >$, we simply append b onto a forming the new curve c, such that $c =< a_1, \ldots, a_n, b_1, \ldots, b_m >$. Our implemenation of this is shown in figure 7 below.

```
curve p ps = Curve (p : ps)
```

Figure 7: Excerpt showing the `connect` function (../Curves.hs)

Rotating Curves

Rotation about the origin is given in 2 dimensions by the rotation matrix;

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{1}$$

Our implementation maps each point in the curve to a new curve by using a local function (`rotate'`) that calculates the rotation of a single point, also taking care not to recalculate anything unnecessary.

```
43 connect (Curve(xs)) (Curve(ys)) = Curve(xs ++ ys)
44
45 -- Rotate a curve around origin (0,0) by d degrees.
46 rotate :: Curve -> Double -> Curve
47 rotate (Curve(cs)) d = Curve (map (rotate' d) cs)
48   where rotate' :: Double -> Point -> Point
```

Figure 8: Excerpt showing the `rotate` function (../Curves.hs)

Since we are required to give the angle in degrees, but the trigonometric functions of Haskell take their arguments in radians, we must make sure to convert these properly beforehand. This conversion is apparent on line 48. Also, the formula above rotates counter-clockwise, and we'd like ours to rotate clockwise. So, we must make sure to negate the angle argument.

Translating Curves

To translate the curve, we must first calculate the difference between the given argument point and the curve starting-point. This delta-point is then used for an additive `map` over the entire sequence of points in the curve.

```


-- Translate a Curve around the plane.
```

Figure 9: Excerpt showing the `translate` function (../Curves.hs)

Reflecting Curves

To reflect a curve we pattern-match on the `Axis` argument, and then map each point in the curve by substracting the opposite component from twice the given offset. This is shown in figure 10 below.

```
data Axis = Vertical | Horizontal

-- Reflect a curve around an axis, the axis can be offset by offset "o".
```

Figure 10: Excerpt showing the `reflect` function (../Curves.hs)

Curve Bounding Box

In calculating the we made a local partial function (`cmp`) used to fold the points of the curve on each composant by the `min` function for the lower-left point and by the `max` function for the upper-right point.

```
reflect (Curve(ps)) Horizontal o   = Curve (map (\(Point(x,y)) -> Point(x, -y+2*o)) ps)

-- Calculate bounding box
```

Figure 11: Excerpt showing the `bbox` function (../Curves.hs)

Curve Width & Height

Calculating the width was done by retrieving $x_{m}in$ and $x_{m}ax$ from the bounding box, and returning the difference. That is, $x_{m}ax - x_{m}in$, as shown in figure 12 below. The same is done in calculating the height of the curve, so figure 13 should be pretty self-explanatory.

```
  where cmp f = \(Point(ax,ay)) (Point(bx,by)) -> Point(f ax bx, f ay by)

-- Get the width of the bounding box.
```

Figure 12: Excerpt showing the `width` function (../Curves.hs)

```
  where (Point(xmin,_), Point(xmax,_)) = bbox(c)

-- Get the height of the bounding box.
```

Figure 13: Excerpt showing the `height` function (../Curves.hs)

Making Lists From Curves

This is done entirely by pattern-matching. As apparent of figure 14 we simply grab the list contained within the `Curve` type.

```
where (Point(_,ymin), Point(_,ymax)) = bbox(c)
```

Figure 14: Excerpt showing the `toList` function (../Curves.hs)

Generate SVG

The `toSVG` function takes a Curve type and generates a string containing the SVG data. The `toFile` function takes a Curve object and and string with a filename and writes the string returned from `toSVG` to the file.

In `toSVG` we first try and convert the Curve into screen coordinates, where the origin is in the top left of the positive quadrant and not lower left as in a Kartesian coordinate system.

We generate the header of the SVG using a constant string where the height and width are calculated by taking the height/width of the curve and then adding offsets. The lines are created by taking the image height and subtracting the screen coordinates, to make up for the fact that the y-axis grows downwards in the screen cooridnate system, and not upwards.

The following imags have been generated from our test Curves using `toSVG`.

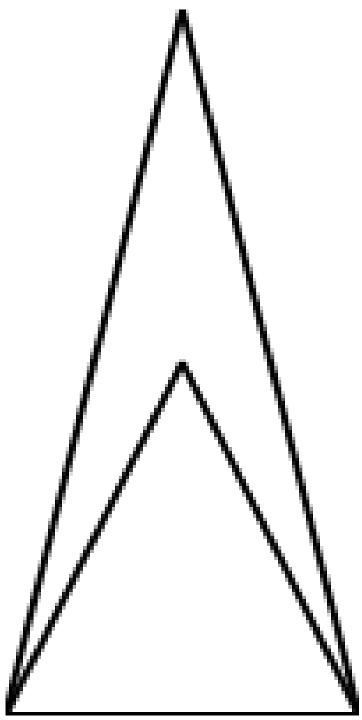

Figure 15: Two triangles drawn over one another.

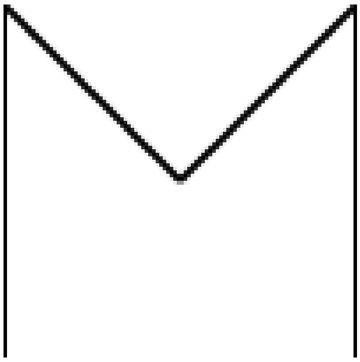

Figure 16: The letter “M”.

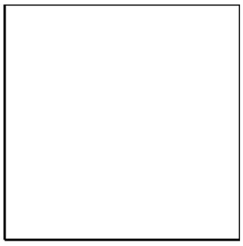

Figure 17: A square, translated slightly left, to test if the whitespace would stay.

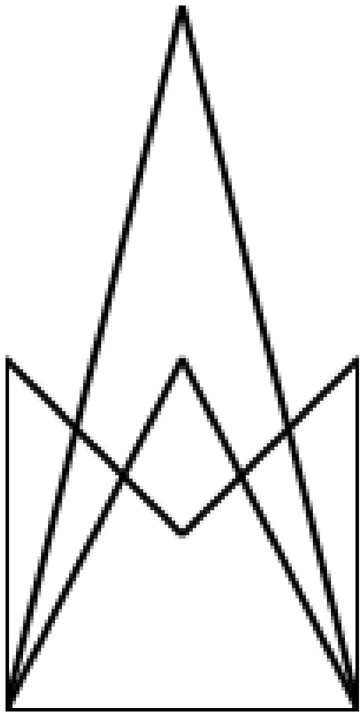

Figure 18: The Figure 15 and Figure 16 connected using the `Curves->connect` function we implemented.

Test with Hilbert Curves

We failed in producing a correct Hilbert curve. We did however see parts of it drawn to some extent, but failed to find the source of the problem. We do know for a fact that the conversion to screen coordinate system is flawed, but some of the issues may also stem from our manipulatory library functions.

Peano and Other Curves (Optional)

Extensions (Optional)