A short report, report.pdf, explaining your code, and containing an assessment of your implementation, including what this assessment is based upon. In the report, you should also:

1. Discuss how you ensure that the user of your module cannot construct empty curves.
2. Tell us how we can run your code and reproduce your test results.

What is empty curve

What is an assessment

What the basis of assessment is

What tests

1. Point

1.1 Data type declaration

data Point = Point Double Double deriving (Show)

The Point data type is implemented using a pair of Doubles representing the coordinates of the point on the XOY axis. The first Double denotes the coordinates of the OX axis while the second the coordinates on the OY axis.

1.2 Related Functions

1. point :: (Double, Double) -> Point

point (x,y) = Point x y

This function takes a tuple of Doubles and constructs a Point from them. The two numbers represent the OX and OY coordinates of the point, in that order.

1. instance Eq Point where

p1 == p2 = abs (pointX p1 - pointX p2) < 0.01 && abs (pointY p1 - pointY p2) < 0.01

In this instance declaration we specify that the type Point is an instance of Eq and we have implemented the definition of “==” on these types. Namely, when two Points are compared only the first 2 digits after the decimal points are considered. This has been implemented by considering all points with a difference less than 0.01 between their respective coordinates (OX and OY) as being equal.

1. pointX :: Point -> Double

pointX (Point x \_) = x

pointY :: Point -> Double

pointY (Point \_ y) = y

The functions pointX and pointY return a Double representing the coordinate of the Point on the OX axis and OY axis respectively.

1. rotatePoint:: Double -> Point -> Point

rotatePoint angle (Point x y) = point (x \* cos angle - y \* sin angle, x \* sin angle + y \* cos angle)

The function rotatePoint rotates counter-clockwise a given point with a given degree about the origin. The first parameter is a Double that represents the desired rotation angle in radians. Since the functions in Haskell used radians we have implemented a function that converts from degrees to radians which will be described in Utilities below.

The formula for rotating a point around the origin is derived from linear algebra by the use of a rotation matrix. In XOY the rotation matrix is of the form:

By multiplying the vector formed of the OX, OY coordinates with the rotation matrix we obtain the new coordinates for the rotated points.:

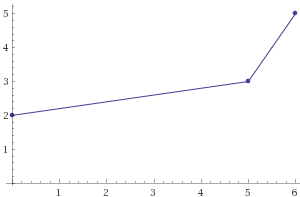
The direction of rotation is counter clockwise because the angle is positive, and the above matrix represents the counter-clockwise rotation matrix.

The new coordinates of the rotated points are given by:

Which is the formula used in the rotatePoint function.

1. translatePoint :: Point -> Point -> Point

translatePoint (Point vx vy) (Point x y) = point (x + vx, y + vy)

The function translatePoint translates a point from the plane by the direction and distance of a given vector. The first paramenter is a Point which is merely a representation of a vector, namely the OX and OY coordinates of the vector , where vx, vy are in the function above. The second parameter represents the point in the plane we would like to translate by the aforementioned vector . To translate the point we merely add the two vectors consisting of their coordinates: .

1. Curve

2.1 Data type declaration

curve :: Point -> [Point] -> Curve

curve = Curve ???

The curve data type has a point representing the starting point of the curve and a list of points which denote the subsequent points following the starting point (not including the starting point) which define the curve. The curve will be a line drawn through all these points, from the starting point and going through all the points from the list in the order of their appearance.

In the figure above there is a curve having the starting point (0,2) and the subsequent points (5,3), (6,5).

2.2 Related functions

1. instance Eq Curve where

(Curve s1 l1) == (Curve s2 l2) = s1 == s2 && l1 == l2

In a similar manner with the Point data structure, we make Curve an instance of Eq and describe the behavior of “==” applied to the Curve data. That is, two curves are considered equal if their starting points are equal and the lists are equal as well (having same elements in the same order).

1. connect :: Curve -> Curve -> Curve

connect (Curve s1 l1) (Curve s2 l2) = curve s1 (l1++s2:l2)

The function connect takes two Curves and returns a Curve formed from the first Curve parameter (having the same starting Point s1), a line connecting its last point with the first point of the second Curve parameter (which is merely a concatenation of lists as seen above, with the addition of the starting point of the second Curve).

1. rotate :: Curve -> Double -> Curve

rotate (Curve s l) degAngle = curve (rotatePoint (toRadians degAngle) s ) (map (rotatePoint (toRadians degAngle)) l)