Assignment 6

due at 23:59 on Wed Nov 1 (80 points)

Save all your functions into one file called `a06.hs` and submit it to this dropbox.

Type classes

In the notes, we defined types for `Circle` and `Rectangle` that were instances of this `Shape` class:

```
class Shape a where
  area :: a -> Float
  bump :: a -> a
```

1. Write a `data` definition for a `Triangle` class, with a constructor also called `Triangle`. Its arguments should be six `Float` values, representing the `x`, `y` coordinates of the triangle's three vertices. Your definition should automatically derive the instances for `Show` and `Eq`.

2. Instantiate the `Shape` class for the `Triangle` type. To compute the area of a triangle, use the formula at `http://www.mathopenref.com/coordtrianglearea.html`. (Often you'll see the area of a triangle written as $\frac{bh}{2}$ where `b` is the length of the base and `h` is the height. But that formula presumes that you know the base and height, which can be difficult to calculate from three arbitrary coordinates.)

   Here are some examples using `Triangle` with the two functions in `Shape`:

   ```
   ghci> area (Triangle 15 15 30 25 15 35)
   150.0
   ghci> area (Triangle 25 15 31 (-5) 41 40)
   235.0
   ghci> bump (Triangle 25 15 31 (-5) 41 40)
   Triangle {ax = 26.0, ay = 16.0, bx = 32.0, by = -4.0, cx = 42.0, cy = 41.0}
   ```

   You can compare your `area` results to those given by the interactive gadget on the page I linked for the formula.

3. Just like the Monoid type class has some associated laws that instances should obey, we can define a law for `Shape`: the area of a shape should not be affected by bumping it to a new position! Encode that law as a Boolean function that works on any `Shape` instance:

   ```
   bumpPreservesArea :: Shape a => a -> Bool
   ```

You will compare the area to the 'bumped' area, and if they are "close enough", return True. (Close enough can be defined as the absolute value of the difference should be less than 0.001, for example.)

```
ghci> bumpPreservesArea (Circle 3.4 5.5 1.8)
True
ghci> bumpPreservesArea (Rectangle 3 5 8 9)
True
ghci> bumpPreservesArea (Triangle 0 0 1 9 2 10)
True
```

Laziness

Here we will use laziness to create and process (potentially) infinite data structures.

4. Create a function `powersOf` with the following signature that generates *all* the non-negative integer powers of the given base.

   ```
   powersOf :: Num a => a -> [a]
   ```

   Here are examples of the powers of two and powers of three:

   ```
   ghci> take 10 $ powersOf 2
   [1,2,4,8,16,32,64,128,256,512]
   ghci> take 10 $ powersOf 3
   [1,3,9,27,81,243,729,2187,6561,19683]
   ghci> powersOf 3 !! 20
   3486784401
   ```

5. Create a function `merge` with the following signature. It will take two lists of ordered elements – each list should be in sorted order already. Then it will merge the elements together into a new list, *maintaining* sorted order.

   ```
   merge :: Ord a => [a] -> [a] -> [a]
   ```

   Here is an example with finite lists:

   ```
   ghci> merge [3,7,18] [2,8,10,24]
   [2,3,7,8,10,18,24]
   ```

   But it should also work if one of the lists is infinite (note that 16 appears twice in the result, because it's in the range `[10..20]` *and* it's a power of two):

   ```
   ghci> take 20 $ merge [10..20] (powersOf 2)
   [1,2,4,8,10,11,12,13,14,15,16,16,17,18,19,20,32,64,128,256]
   ```

Or if both lists are infinite:

```
ghci> take 20 $ merge (powersOf 3) (powersOf 2)
[1,1,2,3,4,8,9,16,27,32,64,81,128,243,256,512,729,1024,2048,2187]
```

Threading state

6. In the notes for 19 October, we wrote a function `preorderState` to thread a state through a pre-order tree traversal.

```
preorderState :: (a -> s -> (b,s)) -> Tree a -> s -> (Tree b, s)
preorderState gen Leaf s0 = (Leaf, s0)
preorderState gen (Branch value left right) s0 =
  (Branch newValue newLeft newRight, s3)
  where (newValue, s1) = gen value s0
        (newLeft, s2) = preorderState gen left s1
        (newRight, s3) = preorderState gen right s2
```

Change that to make `postorderState` on the same `Tree` type. It should behave like this:

```
ghci> printTree sample1
- "A"
  |- "K"
  |  |- "M"
  |  |  |- *
  |  |  |- "Q"
  |  |- "P"
  |- *

ghci> (t0, _) = postorderState withCounter sample1 1
ghci> printTree t0
- ("A",5)
  |- ("K",4)
  |  |- ("M",2)
  |  |  |- *
  |  |  |- ("Q",1)
  |  |- ("P",3)
  |- *

ghci> (t0, _) = postorderState inject sample1 "Z"
ghci> printTree t0
- "K"
  |- "P"
  |  |- "Q"
```

```
|  |  |- *
|  |  |- "Z"
|  |- "M"
|- *
```

where `inject`, `withCounter`, `printTree`, and so forth are defined as in the notes.

Test code

```haskell
import Control.Monad.State
main = flip execStateT (0,0) $ do
    -- Triangles
    let t0 = Triangle 15 15 30 25 15 35
        t1 = Triangle 25 15 31 (-5) 41 40
    verifyF "1.01" 150 $ area t0
    verifyF "1.02" 235 $ area t1
    verify  "1.03" (Triangle 26 16 32 (-4) 42 41) $ bump t1
    assert  "1.04" $ bumpPreservesArea t0
    assert  "1.05" $ bumpPreservesArea t1
    -- powersOf
    verify "2.01" [128,256,512,1024,2048,4096] $ take 6 $ drop 7 $ powersOf 2
    verify "2.02" [729,2187,6561,19683] $ take 4 $ drop 6 $ powersOf 3
    verify "2.03" [1,5,25,125,625] $ take 5 $ powersOf 5
    -- merge
    verify "3.01" [2,3,7,8,10,18,24] $ merge [3,7,18] [2,8,10,24]
    verify "3.02" [1,2,4,5,6,7,8,8,16,32] $ take 10 $ merge [5..8] (powersOf 2)
    verify "3.03" [64,81,128,243,256,512,729,1024] $ take 8 $ drop 10
      $ merge (powersOf 3) (powersOf 2)
  where
    say = liftIO . putStrLn
    correct (k, n) = (k+1, n+1)
    incorrect (k, n) = (k, n+1)
    assert s = verify s True
    verify :: (Show a, Eq a) => String -> a -> a -> StateT (Int,Int) IO ()
    verify = verify' (==)
    verifyF = verify' (\x y -> abs(x-y) < 0.00001)
    verify' :: (Show a) => (a -> a -> Bool) -> String -> a -> a ->
               StateT (Int,Int) IO ()
    verify' eq tag expected actual
      | eq expected actual = do
          modify correct
          say $ " OK " ++ tag
      | otherwise = do
          modify incorrect
          say $ "ERR " ++ tag ++ ": expected " ++ show expected
```

```
            ++ " got " ++ show actual
-- End of test driver
```