Assignment 6 solutions

Test driver

```
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
```

Type classes

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

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

Here is a definition for a triangle, in terms of the coordinates of its three vertices:

```haskell
data Triangle = Triangle { ax, ay, bx, by, cx, cy :: Float }
  deriving (Show, Eq)
```

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.

```haskell
instance Shape Triangle where
  area (Triangle ax ay bx by cx cy) =
    abs ((ax*(by-cy) + bx*(cy-ay) + cx*(ay-by))/2)
  bump (Triangle ax ay bx by cx cy) =
    Triangle (ax+1) (ay+1) (bx+1) (by+1) (cx+1) (cy+1)
```

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! We can encode that as a generic function on any instance:

```haskell
bumpPreservesArea :: Shape a => a -> Bool
bumpPreservesArea shape =
  closeEnough (area shape) (area (bump shape))
  where closeEnough n1 n2 = abs (n1 - n2) < 0.0001
```

Here are some examples:

```haskell
data Circle = Circle { centerX, centerY, radius :: Float }
  deriving Show

data Rectangle = Rectangle { x1, y1, x2, y2 :: Float }
  deriving Show
```

```haskell
instance Shape Circle where
  area (Circle x y r) = pi * r * r
  bump (Circle x y r) = Circle (x+1) (y+1) r

instance Shape Rectangle where
  area (Rectangle x1 y1 x2 y2) = abs (x1 - x2) * abs (y1 - y2)
  bump (Rectangle x1 y1 x2 y2) = Rectangle (x1+1) (y1+1) (x2+1) (y2+1)
```

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

Laziness

```haskell
powersOf :: Num a => a -> [a]
powersOf n = map (n^) [0..]

merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys

runningSums xs = 0 : zipWith (+) xs (runningSums xs)
```

Threading state

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

prettyPrint :: Show a => String -> Tree a -> String
prettyPrint indent Leaf = indent ++ "- *\n"
prettyPrint indent (Branch v Leaf Leaf) =
  indent ++ "- " ++ show v ++ "\n"
```

```haskell
prettyPrint indent (Branch v l r) =
  indent ++ "- " ++ show v ++ "\n" ++ prettyPrint tab l ++ prettyPrint tab r
  where tab = indent ++ "  |"

printTree :: Show a => Tree a -> IO ()
printTree = putStrLn . prettyPrint ""

data Tree a
  = Leaf
  | Branch { value :: a, left, right :: Tree a }
  deriving (Show)
```

And here's a sample tree we used before:

```haskell
sample1 :: Tree String
sample1 =
  Branch "A"
    (Branch "K"
       (Branch "M"
          Leaf
          (Branch "Q" Leaf Leaf))
       (Branch "P" Leaf Leaf))
    Leaf

data Seed = Seed { unSeed :: Integer }
  deriving (Eq, Show)

rand :: Seed -> (Integer, Seed)
rand (Seed s) = (s', Seed s')
  where
    s' = (s * 16807) `mod` 0x7FFFFFFF

withCounter :: a -> Int -> ((a, Int), Int)
withCounter value n = ((value, n), n+1)

inject :: a -> a -> (a, a)
inject value next = (next, value)
```