Assignment 3 solutions

```
{-# LANGUAGE FlexibleContexts #-}
```

Test driver

```
import Control.Monad.RWS
import Control.Monad.State
import System.IO
main = do
  createGraphs  -- You can remove this line
  flip execStateT (0,0) $ do
    -- Ex 1 depth
    verify "1.01 depth" 2 $ depth tree1
    verify "1.02 depth" 4 $ depth tree2
    verify "1.03 depth" 2 $ depth expr1
    verify "1.04 depth" 0 $ depth (Leaf "a")
    -- Ex 2 listLeaves
    verify "2.01 listLeaves" "ABCD"      $ listLeaves tree1
    verify "2.02 listLeaves" [3,4,5,7,9] $ listLeaves tree2
    verify "2.03 listLeaves" [1,2,3]     $ listLeaves expr1
    verify "2.04 listLeaves" [99]        $ listLeaves $ Leaf 99
    -- Ex 3 mirrorTree
    verify "3.01 mirrorTree" "DCBA"      $ listLeaves $ mirrorTree tree1
    verify "3.02 mirrorTree" [9,7,5,4,3] $ listLeaves $ mirrorTree tree2
    verify "3.03 mirrorTree" [3,2,1]     $ listLeaves $ mirrorTree expr1
    verify "3.04 mirrorTree" [99]        $ listLeaves $ mirrorTree $ Leaf 99
    -- Ex 4 mapLeaves
    verify  "4.01 mapLeaves" "BCDE"          $ listLeaves $ mapLeaves succ tree1
    verify  "4.02 mapLeaves" "@ABC"          $ listLeaves $ mapLeaves pred tree1
    verify  "4.03 mapLeaves" [9,16,25,49,81] $ listLeaves $ mapLeaves (^2) tree2
    verifyF "4.04 mapLeaves" 22.245312 $ sum $ listLeaves $ mapLeaves (**2.5) expr1
    -- Ex 5 mapBranches
    verify "5.01 mapBranches" (Leaf 6) $ mapBranches (+1) (Leaf 6)
    verify "5.02 mapBranches" (Branch 9 (Leaf 6) (Leaf 7)) $
             mapBranches (+1) (Branch 8 (Leaf 6) (Leaf 7))
    -- Ex 6
    verifyF "6.01 calculate" (pi/2) $ calculate Divide   pi 2
    verifyF "6.02 calculate" (pi+2) $ calculate Add      pi 2
    verifyF "6.03 calculate" (pi*2) $ calculate Multiply pi 2
    verifyF "6.04 calculate" (pi-2) $ calculate Subtract pi 2
    verifyF "6.05 calculate" (pi^2) $ calculate Power    pi 2
```

```
    -- Ex 7
    verifyF "7.01 interpret"  9.0 $ interpret expr1
    verifyF "7.02 interpret" 27.0 $ interpret $ mapLeaves (+1.5) expr1
    verifyF "7.03 interpret"  6.0 $ interpret $ mapBranches (const Multiply) expr1
    -- Ex 8
    verify  "8.01 expr2" 3 $ depth expr2
    verify  "8.02 expr2" [pi,7,19.1,18.2,6,0.3] $ listLeaves expr2
    verifyF "8.03 expr2" 21.477394 $ interpret expr2
    verifyF "8.04 expr2" 21.480368 $ interpret $ mirrorTree expr2
    verifyF "8.05 expr2" 379.6297 $ interpret $ mapLeaves (+1) expr2
    verifyF "8.06 expr2" 53.741592 $ interpret $ mapBranches (const Add) expr2
  where
    say = liftIO . putStrLn
    correct (k, n) = (k+1, n+1)
    incorrect (k, n) = (k, n+1)
    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
```

Provided code

Here is the tree data type that we developed in class. The type variable `lv` stands for the type of values stored at the leaves. The type variable `bv` stands for the type of values stored at the branches.

```
data Tree lv bv
  = Leaf {leafValue :: lv}
  | Branch {branchValue :: bv, left, right :: Tree lv bv}
  deriving (Show, Eq)

tree1 :: Tree Char Int
tree1 =
  Branch 1
```

```
    (Branch 3 (Leaf 'A') (Leaf 'B'))
    (Branch 5 (Leaf 'C') (Leaf 'D'))

tree2 :: Tree Int Char
tree2 =
  Branch 'A'
    (Leaf 3)
    (Branch 'B'
      (Branch 'C'
        (Leaf 4)
        (Branch 'D'
          (Leaf 5)
          (Leaf 7)))
      (Leaf 9))
```

Generic tree operations

1. depth

```
depth :: Tree lv bv -> Int
depth (Leaf _) = 0
depth (Branch _ left right) = 1 + max (depth left) (depth right)
```

2. listLeaves

```
listLeaves :: Tree lv bv -> [lv]
listLeaves (Leaf x) = [x]
listLeaves (Branch _ left right) = listLeaves left ++ listLeaves right
```

3. mirrorTree

```
mirrorTree :: Tree lv bv -> Tree lv bv
mirrorTree (Leaf x) = Leaf x
mirrorTree (Branch y left right) = Branch y (mirrorTree right) (mirrorTree left)
```

4. mapLeaves

```
mapLeaves :: (lv1 -> lv2) -> Tree lv1 bv -> Tree lv2 bv
mapLeaves f (Leaf x) = Leaf (f x)
mapLeaves f (Branch y left right) =
  Branch y (mapLeaves f left) (mapLeaves f right)
```

5. mapBranches

```
mapBranches :: (bv1 -> bv2) -> Tree lv bv1 -> Tree lv bv2
mapBranches f (Leaf x) = Leaf x
mapBranches f (Branch y left right) =
  Branch (f y) (mapBranches f left) (mapBranches f right)
```

Trees to represent arithmetic expressions

```
data ArithOp = Add | Subtract | Multiply | Divide | Power
  deriving (Show, Eq)
```

6. calculate

```
calculate :: ArithOp -> Float -> Float -> Float
calculate Add x y = x + y
calculate Subtract x y = x - y
calculate Multiply x y = x * y
calculate Divide x y = x / y
calculate Power x y = x ** y
```

7. interpret

```
interpret :: Tree Float ArithOp -> Float
interpret (Leaf x) = x
interpret (Branch op left right) =
  calculate op (interpret left) (interpret right)

expr1 :: Tree Float ArithOp
expr1 =
  Branch Multiply
    (Branch Add (Leaf 1) (Leaf 2))
    (Leaf 3)
```

8. expr2

```
expr2 :: Tree Float ArithOp
expr2 =
  Branch Power
    (Branch Subtract
       (Branch Multiply (Leaf pi) (Leaf 7))
       (Leaf 19.1))
    (Branch Divide
       (Leaf 18.2)
       (Branch Add (Leaf 6) (Leaf 0.3)))
```

Extra

You don't necessarily need to look at this section, but I'll describe what it does in case you're curious. The function `graphviz` takes a tree and converts it into a string in a language called GraphViz that can then be fed to a command-line tool to produce images. All the diagrams of trees on the assignment handout page were produced from this Haskell program and GraphViz.

```
graphviz :: (Show lv, Show bv) => Tree lv bv -> String
graphviz tree = surround $ snd $ execRWS (traverse tree) () 0
  where
    surround s = "digraph {\n" ++ s ++ "}\n"
    quote a = "\"" ++ show a ++ "\""
    increment :: MonadState Int m => (Int -> m a) -> m Int
    increment f = do
      i <- get
      put (i+1)
      f i
      return i
    node i = "node" ++ show i
    traverse (Leaf x) = increment $ \i ->
      tell $ node i ++ " [shape=rectangle, label=" ++ quote x ++ "]\n"
    traverse (Branch y left right) = increment $ \i -> do
      j <- traverse left
      k <- traverse right
      tell $
        node i ++ " [label=" ++ quote y ++ "]\n" ++
        node i ++ " -> " ++ node j ++ "\n" ++
        node i ++ " -> " ++ node k ++ "\n"

createGraphs = mapM_ each
  [ ("expr1", graphviz expr1)
  , ("expr2", graphviz expr2)
  , ("tree1", graphviz tree1)
  , ("tree2", graphviz tree2)
  , ("tree1mirror", graphviz (mirrorTree tree1))
  , ("tree2mirror", graphviz (mirrorTree tree2))
  , ("tree2map", graphviz (mapLeaves (^2) tree2))
  , ("tree2map2", graphviz (mapBranches (succ . succ) tree2))
  ]
  where
    each (name, dot) =
      withFile ("a03" ++ name ++ ".dot") WriteMode $ flip hPutStr dot
```