Homework

Preliminaries

Before starting this part of the assignment,

1. Install the following packages

```
$ cabal install quickcheck
```

2. Learn to read the documentation

To complete this homework, download this file as plain text and answer each question, filling in code where it says `"TODO"`. Your code must typecheck against the given type signatures. Feel free to add your own tests to this file to exercise the functions you write. Submit your homework by sending this file, filled in appropriately, to cse230@goto.ucsd.edu with the subject "HW3"; you will receive a confirmation email after submitting. Please note that this address is unmonitored; if you have any questions about the assignment, post to Piazza.

```
{-# LANGUAGE TypeSynonymInstances, FlexibleContexts, NoMonomorphismRestriction, OverlappingI
module Solution where
import qualified Hw3 as H
import qualified Data.Map as Map

import Control.Monad.State
import Control.Monad.Error
import Control.Monad.Writer

import Test.QuickCheck hiding ((===))
import Control.Monad (forM, forM_)
import Data.List (transpose, intercalate, nubBy, sortBy)

import Control.Applicative

quickCheckN n = quickCheckWith $ stdArgs { maxSuccess = n}
```

Problem 0: All About You

Tell us your name, email and student ID, by replacing the respective strings below

```
myName  = "Write Your Name  Here"
myEmail = "Write Your Email Here"
mySID   = "Write Your SID   Here"
```

Problem 1: Binary Search Trees Revisited

Recall the old type of binary search trees from HW2.

– > data BST k v = Emp – > | Bind k v (BST k v) (BST k v) – > deriving (Show)

```
toBinds :: H.BST t t1 -> [(t, t1)]
toBinds H.Emp            = []
toBinds (H.Bind k v l r) = toBinds l ++ [(k,v)] ++ toBinds r
```

The following function tests whether a tree satisfies the binary-search-order invariant.

```
isBSO ::  Ord a => H.BST a b -> Bool
isBSO H.Emp            = True
isBSO (H.Bind k v l r) = all (< k) lks && all (k <) rks && isBSO l && isBSO r
  where lks = map fst $ toBinds l
        rks = map fst $ toBinds r
```

Finally, to test your implementation, we will define a type of operations over trees

```
data BSTop k v = BSTadd k v | BSTdel k
                 deriving (Eq, Show)
```

and a function that constructs a tree from a sequence of operations

```
ofBSTops ::  Ord k => [BSTop k v] -> H.BST k v
ofBSTops    = foldr doOp H.Emp
  where doOp (BSTadd k v) = bstInsert k v
        doOp (BSTdel k)   = bstDelete k
```

and that constructs a reference `Map` from a sequence of operations

```
mapOfBSTops ::  Ord k => [BSTop k a] -> Map.Map k a
mapOfBSTops = foldr doOp Map.empty
  where doOp (BSTadd k v) = Map.insert k v
        doOp (BSTdel k)   = Map.delete k
```

and functions that generate an arbitrary BST operations

```
keys :: [Int]
keys = [0..10]

genBSTadd, genBSTdel, genBSTop ::  Gen (BSTop Int Char)
genBSTadd = liftM2 BSTadd (elements keys) (elements ['a'..'z'])
genBSTdel = liftM BSTdel (elements keys)
genBSTop  = frequency [(5, genBSTadd), (1, genBSTdel)]
```

(a) Insertion

Write an insertion function

```
bstInsert :: (Ord k) => k -> v -> H.BST k v -> H.BST k v
bstInsert k v t = balance $ bstInsertRaw k v t

bstInsertRaw :: (Ord k) => k -> v -> H.BST k v -> H.BST k v
bstInsertRaw k v H.Emp = H.Bind k v H.Emp H.Emp
bstInsertRaw k v (H.Bind k' v' l r)
    | k < k'    = H.Bind k' v' (bstInsert k v l) r
    | k > k'    = H.Bind k' v' l (bstInsert k v r)
    | otherwise = H.Bind k v l r
```

such that `bstInsert k v t` inserts a key `k` with value `v` into the tree `t`. If `k` already exists in the input tree, then its value should be *replaced* with `v`. When you are done, your code should satisfy the following QC properties.

```
prop_insert_bso :: Property
prop_insert_bso = forAll (listOf genBSTadd) $ \ops ->
                    isBSO (ofBSTops ops)

prop_insert_map = forAll (listOf genBSTadd) $ \ops ->
                    toBinds (ofBSTops ops) == Map.toAscList (mapOfBSTops ops)
```

(b) Deletion

Write a deletion function for BSTs of this type:

```
bstDelete k t = balance $ bstDeleteRaw k t

maxRight :: (Ord k) => H.BST k v -> Maybe (H.BST k v, k, v)
maxRight H.Emp                = Nothing
maxRight (H.Bind k v l H.Emp) = Just (l, k, v)
maxRight (H.Bind k v l r  )   = case maxRight r of
                                  Just (t, k', v') -> Just (H.Bind k v l t, k', v')

bstDeleteRaw :: (Ord k) => k -> H.BST k v -> H.BST k v
bstDeleteRaw k H.Emp                          = H.Emp
bstDeleteRaw k (H.Bind k' v l r) | k < k'    = H.Bind k' v (bstDeleteRaw k l) r
                                 | k > k'    = H.Bind k' v l (bstDeleteRaw k r)
                                 | otherwise = case maxRight l of
                                                 Nothing         -> r
                                                 Just (l', k, v) -> H.Bind k v l' r
```

such that `bstDelete k t` removes the key `k` from the tree `t`. If `k` is absent from the input tree, then the tree is returned unchanged as the output. When you are done, your code should satisfy the following QC properties.

```
prop_delete_bso :: Property
prop_delete_bso = forAll (listOf genBSTop) $ \ops ->
                    isBSO (ofBSTops ops)

prop_delete_map = forAll (listOf genBSTop) $ \ops ->
                    toBinds (ofBSTops ops) == Map.toAscList (mapOfBSTops ops)
```

(c) Balanced Trees

The following function determines the `height` of a BST

```
height (H.Bind _ _ l r) = 1 + max (height l) (height r)
height H.Emp            = 0
```

We say that a tree is *balanced* if

```
isBal (H.Bind _ _ l r) = isBal l && isBal r && abs (height l - height r) <= 2
isBal H.Emp            = True
```

Write a balanced tree generator

```
genBal :: Gen (H.BST Int Char)
genBal = mkBalanced . nubBy eqKeys . sortBy cmpKeys <$> listOf genBSTadd
  where
    cmpKeys (BSTadd k1 _) (BSTadd k2 _) = compare k1 k2
    eqKeys  (BSTadd k1 _) (BSTadd k2 _) = k1 == k2
    mkBalanced [] = H.Emp
    mkBalanced [BSTadd k v] = H.Bind k v H.Emp H.Emp
    mkBalanced xs = let (ys, (BSTadd k v):zs) = splitAt (length xs `div` 2) xs
                        l = mkBalanced ys
                        r = mkBalanced zs
                    in H.Bind k v l r
```

such that

```
prop_genBal = forAll genBal isBal
```

(d) Height Balancing (** Hard **)

Rig it so that your insert and delete functions *also* create balanced trees. That is, they satisfy the properties

```
prop_insert_bal ::  Property
prop_insert_bal = forAll (listOf genBSTadd) $ isBal . ofBSTops

prop_delete_bal ::  Property
prop_delete_bal = forAll (listOf genBSTop) $ isBal . ofBSTops

rotateL :: (Ord k) => H.BST k v -> H.BST k v
rotateL H.Emp = H.Emp
rotateL t@(H.Bind k v l r)
    = case r of
        H.Emp -> t
        H.Bind k' v' l' r' -> H.Bind k' v' (H.Bind k v l l') r'

rotateR :: (Ord k) => H.BST k v -> H.BST k v
rotateR H.Emp = H.Emp
rotateR t@(H.Bind k v l r)
    = case l of
        H.Emp -> t
        H.Bind k' v' l' r' -> H.Bind k' v' l' (H.Bind k v r' r)
```

```haskell
balanceFactor H.Emp = 0
balanceFactor (H.Bind _ _ l r) = height l - height r

balance :: (Ord k) => H.BST k v -> H.BST k v
balance H.Emp = H.Emp
balance t@(H.Bind k v l r) = go $ balanceFactor t
  where
    go n
      -- left-right
      | n > 1 && (balanceFactor l) < 0
        = rotateR $ H.Bind k v (rotateL l) r
      -- left-left
      | n > 1
        = rotateR t
      -- right-left
      | n < -1 && (balanceFactor r) > 0
        = rotateL $ H.Bind k v l (rotateR r)
      -- right-right
      | n < -1
        = rotateL t
      | otherwise = t
```

Problem 2: Circuit Testing

Credit: UPenn CIS552

For this problem, you will look at a model of circuits in Haskell.

Signals

A *signal* is a list of booleans.

```haskell
newtype Signal = Sig [Bool]
```

By convention, all signals are infinite. We write a bunch of lifting functions that lift boolean operators over signals.

```haskell
lift0 ::  Bool -> Signal
lift0 a = Sig $ repeat a

lift1 ::  (Bool -> Bool) -> Signal -> Signal
lift1 f (Sig s) = Sig $ map f s
```

```haskell
lift2 ::  (Bool -> Bool -> Bool) -> (Signal, Signal) -> Signal
lift2 f (Sig xs, Sig ys) = Sig $ zipWith f xs ys

lift22 :: (Bool -> Bool -> (Bool, Bool)) -> (Signal, Signal) -> (Signal,Signal)
lift22 f (Sig xs, Sig ys) =
  let (zs1,zs2) = unzip (zipWith f xs ys)
  in (Sig zs1, Sig zs2)

lift3 :: (Bool->Bool->Bool->Bool) -> (Signal, Signal, Signal) -> Signal
lift3 f (Sig xs, Sig ys, Sig zs) = Sig $ zipWith3 f xs ys zs
```

Simulation

Next, we have some helpers that can help us simulate a circuit by showing how it behaves over time. For testing or printing, we truncate a signal to a short prefix

```haskell
truncatedSignalSize = 20
truncateSig bs = take truncatedSignalSize bs

instance Show Signal where
  show (Sig s) = show (truncateSig s) ++ "..."

trace :: [(String, Signal)] -> Int -> IO ()
trace desc count = do
  putStrLn   $ intercalate " " names
  forM_ rows $ putStrLn . intercalate " " . rowS
  where (names, wires) = unzip desc
        rows           = take count . transpose . map (\ (Sig w) -> w) $ wires
        rowS bs        = zipWith (\n b -> replicate (length n - 1) ' ' ++ (show (binary b))) names bs

probe :: [(String,Signal)] -> IO ()
probe desc = trace desc 1

simulate :: [(String, Signal)] -> IO ()
simulate desc = trace desc 20
```

Testing support (QuickCheck helpers)

Next, we have a few functions that help to generate random tests

```haskell
instance Arbitrary Signal where
  arbitrary = do
    x      <- arbitrary
```

```
    Sig xs <- arbitrary
    return $ Sig (x : xs)

arbitraryListOfSize n = forM [1..n] $ \_ -> arbitrary
```

To check whether two values are equivalent

```
class Agreeable a where
  (===) :: a -> a -> Bool

instance Agreeable Signal where
  (Sig as) === (Sig bs) =
    all (\x->x) (zipWith (==) (truncateSig as) (truncateSig bs))

instance (Agreeable a, Agreeable b) => Agreeable (a,b) where
  (a1,b1) === (a2,b2) = (a1 === a2) && (b1 === b2)

instance Agreeable a => Agreeable [a] where
  as === bs = all (\x->x) (zipWith (===) as bs)
```

To convert values from boolean to higher-level integers

```
class Binary a where
  binary :: a -> Integer

instance Binary Bool where
  binary b = if b then 1 else 0

instance Binary [Bool] where
  binary = foldr (\x r -> (binary x) + 2 *r) 0
```

And to probe signals at specific points.

```
sampleAt n (Sig b) = b !! n
sampleAtN n signals = map (sampleAt n) signals
sample1 = sampleAt 0
sampleN = sampleAtN 0
```

Basic Gates

The basic gates from which we will fashion circuits can now be described.

```
or2 ::  (Signal, Signal) -> Signal
or2 = lift2 $ \x y -> x || y

xor2 :: (Signal, Signal) -> Signal
xor2 = lift2 $ \x y -> (x && not y) || (not x && y)

and2 :: (Signal, Signal) -> Signal
and2 = lift2 $ \x y -> x && y

imp2 ::  (Signal, Signal) -> Signal
imp2 = lift2 $ \x y -> (not x) || y

mux :: (Signal, Signal, Signal) -> Signal
mux = lift3 (\b1 b2 select -> if select then b1 else b2)

demux :: (Signal, Signal) -> (Signal, Signal)
demux args = lift22 (\i select -> if select then (i, False) else (False, i)) args

muxN :: ([Signal], [Signal], Signal) -> [Signal]
muxN (b1,b2,sel) = map (\ (bb1,bb2) -> mux (bb1,bb2,sel)) (zip b1 b2)

demuxN :: ([Signal], Signal) -> ([Signal], [Signal])
demuxN (b,sel) = unzip (map (\bb -> demux (bb,sel)) b)
```

Basic Signals

Similarly, here are some basic signals

```
high = lift0 True
low  = lift0 False

str   ::  String -> Signal
str cs = Sig $ (map (== '1') cs) ++ (repeat False)

delay ::  Bool -> Signal -> Signal
delay init (Sig xs) = Sig $ init : xs
```

Combinational circuits

NOTE When you are asked to implement a circuit, you must **ONLY** use the above gates or smaller circuits built from the gates.

For example, the following is a *half-adder* (that adds a carry-bit to a single bit).

```
halfadd :: (Signal, Signal) -> (Signal, Signal)
halfadd (x,y) = (sum,cout)
  where sum   = xor2 (x, y)
        cout  = and2 (x, y)
```

Here is a simple property about the half-adder

```
prop_halfadd_commut b1 b2 =
  halfadd (lift0 b1, lift0 b2) === halfadd (lift0 b2, lift0 b1)
```

We can use the half-adder to build a full-adder

```
fulladd (cin, x, y) = (sum, cout)
  where (sum1, c1)  = halfadd (x,y)
        (sum, c2)   = halfadd (cin, sum1)
        cout        = xor2 (c1,c2)

test1a = probe [("cin",cin), ("x",x), ("y",y), ("  sum",sum), ("cout",cout)]
  where cin        = high
        x          = low
        y          = high
        (sum,cout) = fulladd (cin, x, y)
```

and then an n-bit adder

```
bitAdder :: (Signal, [Signal]) -> ([Signal], Signal)
bitAdder (cin, [])   = ([], cin)
bitAdder (cin, x:xs) = (sum:sums, cout)
  where (sum, c)     = halfadd (cin,x)
        (sums, cout) = bitAdder (c,xs)

test1 = probe [("cin",cin), ("in1",in1), ("in2",in2), ("in3",in3), ("in4",in4),
               ("  s1",s1), ("s2",s2), ("s3",s3), ("s4",s4), ("c",c)]
  where
    cin = high
    in1 = high
    in2 = high
    in3 = low
    in4 = high
    ([s1,s2,s3,s4], c) = bitAdder (cin, [in1,in2,in3,in4])
```

The correctness of the above circuit is described by the following property that compares the behavior of the circuit to the *reference implementation* which is an integer addition function

```
prop_bitAdder_Correct ::  Signal -> [Bool] -> Bool
prop_bitAdder_Correct cin xs =
  binary (sampleN out ++ [sample1 cout]) == binary xs + binary (sample1 cin)
  where (out, cout) = bitAdder (cin, map lift0 xs)
```

Finally, we can use the bit-adder to build an adder that adds two N-bit numbers

```
adder :: ([Signal], [Signal]) -> [Signal]
adder (xs, ys) =
   let (sums,cout) = adderAux (low, xs, ys)
   in sums ++ [cout]
   where
     adderAux (cin, [], [])     = ([], cin)
     adderAux (cin, x:xs, y:ys) = (sum:sums, cout)
                                  where (sum, c) = fulladd (cin,x,y)
                                        (sums,cout) = adderAux (c,xs,ys)
     adderAux (cin, [], ys)     = adderAux (cin, [low], ys)
     adderAux (cin, xs, [])     = adderAux (cin, xs, [low])

test2 = probe [ ("x1", x1), ("x2",x2), ("x3",x3), ("x4",x4),
                (" y1",y1), ("y2",y2), ("y3",y3), ("y4",y4),
                (" s1",s1), ("s2",s2), ("s3",s3), ("s4",s4), (" c",c) ]
  where xs@[x1,x2,x3,x4] = [high,high,low,low]
        ys@[y1,y2,y3,y4] = [high,low,low,low]
        [s1,s2,s3,s4,c]  = adder (xs, ys)
```

And we can specify the correctness of the adder circuit by

```
prop_Adder_Correct ::  [Bool] -> [Bool] -> Bool
prop_Adder_Correct l1 l2 =
  binary (sampleN sum) == binary l1 + binary l2
  where sum = adder (map lift0 l1, map lift0 l2)
```

Problem: Subtraction

1. Using `prop_bitAdder_Correct` as a model, write a specification for a single-bit subtraction function that takes as inputs a N-bit binary number and a single bit to be subtracted from it and yields as outputs an N-bit binary number. Subtracting one from zero should yield zero.

```
prop_bitSubtractor_Correct ::  Signal -> [Bool] -> Bool
prop_bitSubtractor_Correct bin xs =
  binary (sampleN sum) == max 0 (binary xs - binary (sample1 bin))
  where (sum,bout) = bitSubtractor (bin, map lift0 xs)
```

2. Using the `bitAdder` circuit as a model, define a `bitSubtractor` circuit that implements this functionality and use QC to check that your behaves correctly.

```
snot :: Signal -> Signal
snot a = xor2 (high, a)

halfsub :: (Signal, Signal) -> (Signal, Signal)
halfsub (x,y) = (sum,bout)
  where sum   = xor2 (x, y)
        bout  = and2 (y, snot x)

bitSubtractor :: (Signal, [Signal]) -> ([Signal], Signal)
bitSubtractor (bin, [])   = ([], bin)
bitSubtractor (bin, x:xs) = ((and2 (sum, snot bout)):sums, bout)
  where (sum, b)     = halfsub (x,bin)
        (sums, bout) = bitSubtractor (b,xs)
```

Problem: Multiplication

3. Using `prop_Adder_Correct` as a model, write down a QC specification for a `multiplier` circuit that takes two binary numbers of arbitrary width as input and outputs their product.

```
prop_Multiplier_Correct ::  [Bool] -> [Bool] -> Bool
prop_Multiplier_Correct x y =
  (binary x * binary y) == (binary $ sampleN $ multiplier (map lift0 x, map lift0 y))
```

4. Define a `multiplier` circuit and check that it satisfies your specification. (Looking at how adder is defined will help with this, but you'll need a little more wiring. To get an idea of how the recursive structure should work, think about how to multiply two binary numbers on paper.)

```
multiplier :: ([Signal], [Signal]) -> [Signal]
multiplier (xs, [])   = map (const low) xs
multiplier (xs, y:ys) = adder (map (curry and2 y) xs, low : multiplier (xs, ys))
```