Homework

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
module Solution where

import Control.Applicative hiding (empty, (<|>))
import Control.Monad.State hiding (when)
import Text.Parsec hiding (State, between)
import Text.Parsec.Combinator hiding (between)
import Text.Parsec.Char
import Text.Parsec.String
import Data.Foldable (Foldable, foldMap)
import qualified Data.Foldable as F
import Data.Monoid
import Data.Map (findWithDefault, insert, empty)
import Test.QuickCheck

import qualified Hw2 as H
```

This week's homework is presented as a literate Haskell file, just like the lectures. This means that every line beginning with `>` is interpreted as Haskell code by the compiler, while every other line is ignored. (Think of this as the comments and code being reversed from what they usually are.)

You can load this file into `ghci` and compile it with `ghc` just like any other Haskell file, so long as you remember to save it with a `.lhs` suffix.

To complete this homework, download this file as plain text and answer each question, filling in code where noted (i.e. where it says `error "TBD"`).

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 "HW2"; you will receive a confirmation email after submitting.

Before starting this assignment:

1. Install `parsec` via the command `cabal install parsec`
2. Learn to read the documentation
3. Download the test files test.imp, fact.imp, abs.imp, times.imp.

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: All About `foldl`

Define the following functions by filling in the "error" portion:

1. Describe `foldl` and give an implementation:

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl f b []     = b
myFoldl f b (x:xs) = myFoldl f (f b x) xs
```

2. Using the standard `foldl` (not `myFoldl`), define the list reverse function:

```
myReverse :: [a] -> [a]
myReverse = foldl (flip (:)) []
```

3. Define `foldr` in terms of `foldl`:

```
myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr f b = foldl (flip f) b . reverse
```

4. Define `foldl` in terms of the standard `foldr` (not `myFoldr`):

```
myFoldl2 :: (a -> b -> a) -> a -> [b] -> a
myFoldl2 f b = foldr (flip f) b . reverse
```

5. Try applying `foldl` to a gigantic list. Why is it so slow? Try using `foldl'` (from Data.List) instead; can you explain why it's faster?

Part 2: Binary Search Trees

Define a `delete` function for BSTs of this type:

```
popGreatest :: (Ord k) => H.BST k v -> (Maybe (k, v), H.BST k v)
popGreatest H.Emp            = (Nothing, H.Emp)
popGreatest (H.Bind k v l r) = case popGreatest r of
                                 (Nothing, _) -> (Just (k, v), l)
                                 (kv, r')     -> (kv, H.Bind k v l r')

delete :: (Ord k) => k -> H.BST k v -> H.BST k v
delete k H.Emp                        = H.Emp
delete k (H.Bind k' v l r) | k < k'    = H.Bind k' v (delete k l) r
                           | k > k'    = H.Bind k' v l (delete k r)
                           | otherwise = case popGreatest l of
                                           (Nothing, _)        -> r
                                           (Just (k2, v2), l') -> H.Bind k2 v2 l' r

mapKeysToSelf :: H.BST k v -> H.BST k k
mapKeysToSelf H.Emp            = H.Emp
mapKeysToSelf (H.Bind k _ l r) = H.Bind k k (mapKeysToSelf l) (mapKeysToSelf r)

instance Foldable (H.BST k) where
 foldMap f H.Emp            = mempty
 foldMap f (H.Bind _ v l r) = foldMap f l `mappend` foldMap f r `mappend` f v

isBST :: (Ord k) => H.BST k k -> Bool
isBST H.Emp            = True
isBST (H.Bind k _ l r) = F.all (< k) l && F.all (> k) r && isBST l && isBST r

genBST :: Int -> Int -> Int -> Gen (H.BST Int Int)
genBST lb ub sz | sz <= 0   = return H.Emp
                | lb >= ub  = return H.Emp
                | otherwise = do k <- choose (lb, ub)
                                 l <- genBST lb      (k - 1)  (sz `div` 2)
                                 r <- genBST (k + 1) ub       (sz `div` 2)
                                 return $ H.Bind k k l r

instance Arbitrary (H.BST Int Int) where
  arbitrary = sized $ genBST 0 100

someBSTs = sample $ sized $ genBST 0 100

checkBSTGen = quickCheck (isBST :: H.BST Int Int -> Bool)
```

```
isBSTDeleted t k = isBST $ delete k t

deleteProp = forAll arbitrary $ \t -> forAll (arbitrary :: Gen Int) $ \k -> isBSTDeleted t k

checkBSTDelete = quickCheck deleteProp
```

Part 3: An Interpreter for WHILE

Next, you will use monads to build an evaluator for a simple *WHILE* language. In this language, we will represent different program variables as

– > type Variable = String

– Programs in the language are simply values of the type

– > data Statement = – > Assign Variable Expression – x = e – > | If Expression Statement Statement – if (e) {s1} else {s2} – > | While Expression Statement – while (e) {s} – > | Sequence Statement Statement – s1; s2 – > | Skip – no-op – > deriving (Show)

– where expressions are variables, constants or – binary operators applied to sub-expressions

– > data Expression = – > Var Variable – x – > | Val Value – v – > | Op Bop Expression Expression – > deriving (Show)

– and binary operators are simply two-ary functions

– > data Bop = – > Plus – (+) :: Int -> Int -> Int – > | Minus – (-) :: Int -> Int -> Int – > | Times – (*) :: Int -> Int -> Int – > | Divide – (/) :: Int -> Int -> Int – > | Gt – (>) :: Int -> Int -> Bool – > | Ge – (>=) :: Int -> Int -> Bool – > | Lt – (<) :: Int -> Int -> Bool – > | Le – (<=) :: Int -> Int -> Bool – > deriving (Show)

– > data Value = – > IntVal Int – > | BoolVal Bool – > deriving (Show)

We will represent the *store* i.e. the machine's memory, as an associative map from `Variable` to `Value`

– > type Store = Map Variable Value

Note: we don't have exceptions (yet), so if a variable is not found (eg because it is not initialized) simply return the value `0`. In future assignments, we will add this as a case where exceptions are thrown (the other case being type errors.)

We will use the standard library's `State` monad to represent the world-transformer. Intuitively, `State s a` is equivalent to the world-transformer `s -> (a, s)`. See the above documentation for more details. You can ignore the bits about `StateT` for now.

Expression Evaluator

First, write a function

– > evalE :: Expression -> State Store Value

that takes as input an expression and returns a world-transformer that returns a value. Yes, right now, the transformer doesnt really transform the world, but we will use the monad nevertheless as later, the world may change, when we add exceptions and such.

Hint: The value `get` is of type `State Store Store`. Thus, to extract the value of the "current store" in a variable `s` use `s <- get`.

```
intOp :: (Int -> Int -> Int) -> H.Value -> H.Value -> H.Value
intOp op (H.IntVal x) (H.IntVal y) = H.IntVal $ x `op` y
intOp _  _            _             = H.IntVal 0

boolOp :: (Int -> Int -> Bool) -> H.Value -> H.Value -> H.Value
boolOp op (H.IntVal x) (H.IntVal y) = H.BoolVal $ x `op` y
boolOp _  _            _            = H.BoolVal False

-- Note: we don't have exceptions yet, so if a variable is not found,
-- simply return value 0. (Future homework: add this as a case where
-- exceptions would be thrown. Ditto type errors.)
evalE :: H.Expression -> State H.Store H.Value
evalE (H.Var x)      = get >>= return . findWithDefault (H.IntVal 0) x
evalE (H.Val v)      = return v
evalE (H.Op o e1 e2) = (return $ semantics o) `ap` evalE e1 `ap` evalE e2
   where semantics H.Plus   = intOp (+)
         semantics H.Minus  = intOp (-)
         semantics H.Times  = intOp (*)
         semantics H.Divide = intOp (div)
         semantics H.Gt     = boolOp (>)
         semantics H.Ge     = boolOp (>=)
         semantics H.Lt     = boolOp (<)
         semantics H.Le     = boolOp (<=)
```

Statement Evaluator

Next, write a function

```
evalS :: H.Statement -> State H.Store ()
```

that takes as input a statement and returns a world-transformer that returns a unit. Here, the world-transformer should in fact update the input store

appropriately with the assignments executed in the course of evaluating the `Statement`.

Hint: The value `put` is of type `Store -> State Store ()`. Thus, to "update" the value of the store with the new store `s'` do `put s`.

```
evalS w@(H.While e s)    = evalS (H.If e (H.Sequence s w) H.Skip)
evalS H.Skip             = return ()
evalS (H.Sequence s1 s2) = evalS s1 >> evalS s2
evalS (H.Assign x e )    = do v <- evalE e
                              m <- get
                              put $ insert x v m
                              return ()
evalS (H.If e s1 s2) = do v <- evalE e
                          case v of H.BoolVal True  -> evalS s1
                                    H.BoolVal False -> evalS s2
                                    _               -> return ()
```

In the `If` case, if `e` evaluates to a non-boolean value, just skip both the branches. (We will convert it into a type error in the next homework.) Finally, write a function

```
execS :: H.Statement -> H.Store -> H.Store
execS s = execState (evalS s)
```

such that `execS stmt store` returns the new `Store` that results from evaluating the command `stmt` from the world `store`. **Hint:** You may want to use the library function

```
execState :: State s a -> s -> s
```

When you are done with the above, the following function will "run" a statement starting with the `empty` store (where no variable is initialized). Running the program should print the value of all variables at the end of execution.

```
run :: H.Statement -> IO ()
run stmt = do putStrLn "Output Store:"
              putStrLn $ show $ execS stmt empty
```

Here are a few "tests" that you can use to check your implementation.

– > w_test = (Sequence (Assign "X" (Op Plus (Op Minus (Op Plus (Val (IntVal 1)) (Val (IntVal 2))) (Val (IntVal 3))) (Op Plus (Val (IntVal 1)) (Val (IntVal 3))))) (Sequence (Assign "Y" (Val (IntVal 0))) (While (Op Gt (Var "X") (Val

(IntVal 0))) (Sequence (Assign “Y” (Op Plus (Var “Y”) (Var “X”))) (Assign “X” (Op Minus (Var “X”) (Val (IntVal 1))))))))

– > w_fact = (Sequence (Assign “N” (Val (IntVal 2))) (Sequence (Assign “F” (Val (IntVal 1))) (While (Op Gt (Var “N”) (Val (IntVal 0))) (Sequence (Assign “X” (Var “N”)) (Sequence (Assign “Z” (Var “F”)) (Sequence (While (Op Gt (Var “X”) (Val (IntVal 1))) (Sequence (Assign “F” (Op Plus (Var “Z”) (Var “F”))) (Assign “X” (Op Minus (Var “X”) (Val (IntVal 1)))))) (Assign “N” (Op Minus (Var “N”) (Val (IntVal 1)))))))))

As you can see, it is rather tedious to write the above tests! They correspond to the code in the files `test.imp` and `fact.imp`. When you are done, you should get

```
ghci> run w_test
Output Store:
fromList [("X",IntVal 0),("Y",IntVal 10)]

ghci> run w_fact
Output Store:
fromList [("F",IntVal 2),("N",IntVal 0),("X",IntVal 1),("Z",IntVal 2)]
```

Problem 4: A Parser for WHILE

It is rather tedious to have to specify individual programs as Haskell values. For this problem, you will use parser combinators to build a parser for the WHILE language from the previous problem.

Parsing Constants

First, we will write parsers for the `Value` type

```
valueP :: Parser H.Value
valueP = intP <|> boolP
```

To do so, fill in the implementations of

```
intP :: Parser H.Value
intP = many1 digit >>= return . H.IntVal . read
```

Next, define a parser that will accept a particular string `s` as a given value `x`

```
constP :: String -> a -> Parser a
constP s x = string s >> return x
```

and use the above to define a parser for boolean values where `"true"` and `"false"` should be parsed appropriately.

```
boolP :: Parser H.Value
boolP = (constP "true" $ H.BoolVal True) <|> (constP "false" $ H.BoolVal True)
```

Continue to use the above to parse the binary operators

```
opP :: Parser H.Bop
opP =     constP "+"  H.Plus
      <|> constP "-"  H.Minus
      <|> constP "*"  H.Times
      <|> constP "/"  H.Divide
      <|> constP ">"  H.Gt
      <|> constP ">=" H.Ge
      <|> constP "<"  H.Lt
      <|> constP "<=" H.Le
```

Parsing Expressions

Next, the following is a parser for variables, where each variable is one-or-more uppercase letters.

```
varP :: Parser H.Variable
varP = many1 upper
```

Use the above to write a parser for `Expression` values

```
variableExpr :: Parser H.Expression
variableExpr = varP >>= return . H.Var

valExpr :: Parser H.Expression
valExpr = valueP >>= return . H.Val

parenExpr :: Parser H.Expression
parenExpr = do string "("
               e <- exprP
               string ")"
               return e

baseExpr :: Parser H.Expression
baseExpr = valExpr <|> variableExpr <|> parenExpr
```

```haskell
-- This is contorted to avoid left recursion.
-- One should be able to avoid this using:
--  - the Expr module in Parsec
--  - the chain* functions in Parsec
--  - the try combinator
exprOp :: H.Expression -> Parser H.Expression
exprOp e1 = do op <- opP
               spaces
               e2 <- exprP
               return $ H.Op op e1 e2

exprP :: Parser H.Expression
exprP = do e1 <- baseExpr
           spaces
           exprOp e1 <|> return e1
```

Parsing Statements

Next, use the expression parsers to build a statement parser

```haskell
assignStmt :: Parser H.Statement
assignStmt = do v <- varP
                spaces; string ":="; spaces
                e <- exprP
                return $ H.Assign v e

ifStmt :: Parser H.Statement
ifStmt = do string "if"; spaces
            e <- exprP
            spaces; string "then"; spaces
            s1 <- statementP
            spaces; string "else"; spaces
            s2 <- statementP
            spaces; string "endif"
            return $ H.If e s1 s2

whileStmt :: Parser H.Statement
whileStmt = do string "while"; spaces
               e <- exprP
               spaces; string "do"; spaces
               s <- statementP
               spaces; string "endwhile"
               return $ H.While e s

skipStmt :: Parser H.Statement
```

```
skipStmt = string "skip" >> return H.Skip

baseStmt :: Parser H.Statement
baseStmt = assignStmt <|> ifStmt <|> whileStmt <|> skipStmt

seqStatement :: H.Statement -> Parser H.Statement
seqStatement s1 = do char ';'; spaces
                     s2 <- statementP
                     return $ H.Sequence s1 s2

statementP :: Parser H.Statement
statementP = do s1 <- baseStmt
                seqStatement s1 <|> return s1
```

When you are done, we can put the parser and evaluator together in the end-to-end interpreter function

```
runFile s = do p <- parseFromFile statementP s
               case p of
                 Left err   -> print err
                 Right stmt -> run stmt

printStore :: H.Store -> IO ()
printStore e = do putStrLn "Environment:"
                  putStrLn $ show e
```

When you are done you should see the following at the ghci prompt

```
ghci> runFile "test.imp"
Output Store:
fromList [("X",IntVal 0),("Y",IntVal 10)]

ghci> runFile "fact.imp"
Output Store:
fromList [("F",IntVal 2),("N",IntVal 0),("X",IntVal 1),("Z",IntVal 2)]
```