# Homework 5 Due May 8th 11:59 pm

Submission rules:

- **No late days may be used**
- You must submit a single .hs file with the following name: <firstName>-<lastName>-hw5.hs. Failure to do so will result in -10 points.
- You will lose 10 points if you put a module statement at the top of the file.
- You will lose 10 points for any import statements you have in your file and will automatically miss any problems you used an imported function on.
- If your file doesn't compile you will lose 10 points and miss any problems that were causing the compilation errors.
- You must use the skeleton file provided and must not alter any type signature. If you alter a type signature you will automatically miss that problem.

# Problems

## Problem 1 Expression Tree (25 pts)

Given the following definitions:

```
type Identifier = String

data Expr = Num Integer
          | Var Identifier
          | Let { var :: Identifier, value :: Expr, body :: Expr }
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr

type Env = Identifer -> Integer

emptyEnv :: Env
emptyEnv = \s -> error ("unbound " ++ s)

extendEnv :: Env -> Identifier -> Integer -> Env
extendEnv oldEnv s n = \s' -> if s == s' then n else oldEnv s'
```

First instance this type into Show using the following examples as a guide, note you do not need to worry about operator precedence (i.e. no parenthesis necessary):

```
show (Num 1)                                    => 1
show (Var "x")                                  => x
show (Let "x" (Num 3) (Add (Var "x") (Num 5))) => let x = 3 in x + 5 end
show (Add (Num 1) (Add (Num 2) (Var "x")))     => 1 + 2 + x
show (Sub (Num 1) (Sub (Num 2) (Var "x")))     => 1 - 2 - x
show (Mul (Num 1) (Mul (Num 2) (Var "x")))     => 1 * 2 * x
show (Div (Num 1) (Div (Num 2) (Var "x")))     => 1 / 2 / x
```

Now define the following function:

```
evalInEnv :: Env -> Expr -> Integer
```

Which computes the arithmetic value corresponding the to the given *Expr* in the given *Env*.

This can then be used to define *eval* which evaluates expressions without free variables:

```
eval :: Expr -> Integer
eval e = evalInEnv emptyEnv e
```

For example:

```
eval (Let "x" (Num 3) (Add (Var "x") (Num 5)))                    ==> 8
eval (Let "x" (Num 1) (Let "x" (Var "x") (Add (Var "x") (Num 1)))) ==> 2
eval (Let "x" (Var "x") (Var "x"))                                ==> *** Exception: unbour
```

## Problem 2 Diagonalization (35 pts)

What does it mean for a set to be countably infinite? A set is countably infinite if you can put it in 1-to-1 correspondence with the natural numbers. Are the rational numbers countably infinite? When you first think about you might say no because you can keep writing an infinite amount of rationals with the numerator 1 then an infinite amount with the numerator 2 and so on. Hmm but is there a trick to make the enumerable? How can we then put them in 1-to-1 correspondence with the natural numbers? Well lets look at how we could write out the rational numbers:

| | | | | | |
|---|---|---|---|---|---|
| 1/1 | 2/1 | 3/1 | 4/1 | 5/1 | ... |
| 1/2 | 2/2 | 3/2 | 4/2 | 5/2 | ... |
| 1/3 | 2/3 | 3/3 | 4/3 | 5/3 | ... |
| 1/4 | 2/4 | 3/4 | 4/4 | 5/4 | ... |
| 1/5 | 2/5 | 4/5 | 4/5 | 5/5 | ... |
| . | . | . | . | . | . |
| . | . | . | . | . | . |
| . | . | . | . | . | . |

The question remains, how could you enumerate this if each column and row is infinite? Well if you instead go diagonal from right to left then you can

enumerate every value! For example the first few values after this diagonalization are:

1/1, 2/1, 1/2, 3/1 2/2 1/3, 4/1 3/2 2/3 1/4, ...

Thus your task is to create the following function:

```
diag :: [[a]] -> [a]
```

This function takes in a lists of lists where each inner lists represents a row in a matrix as above and returns a flat list that represents the diagonalization of that matrix. All lists should be treated as potentially **infinite**!

To test this code you can use the following functions:

```
rlist :: [[Double]]
rlist = [ [i / j | i <- [1..] ] | j <- [1..] ]

qlist1 :: [[String]]
qlist1 = [ [ show i ++ "/" ++ show j | i <- [1..] ] | j <- [1..] ]

qlist2 :: [[String]]
qlist2 = [ [ fracString i j | i <- [1..] ] | j <- [1..] ]

-- Pretty print a fraction
fracString :: (Show p, Integral p) => p -> p -> String
fracString num den
    | denominator == 1 = show numerator
    | otherwise        = show numerator ++ "/" ++ show denominator
    where
        c           = gcd num den
        numerator   = num `div` c
        denominator = den `div` c

-- Take an n by n block from the top of a list of lists
block :: Int -> [[a]] -> [[a]]
block n xss = map (take n) (take n xss)

blockExample :: Bool
blockExample = block 5 qlist2 ==
    [
        ["1","2","3","4","5"],
        ["1/2","1","3/2","2","5/2"],
        ["1/3","2/3","1","4/3","5/3"],
        ["1/4","1/2","3/4","1","5/4"],
        ["1/5","2/5","3/5","4/5","1"]
    ]

diagTest :: Bool
```

```
diagTest = take 20 (diag qlist2) ==
    [
        "1",
        "2","1/2",
        "3","1","1/3",
        "4","3/2","2/3","1/4",
        "5","2","1","1/2","1/5",
        "6","5/2","4/3","3/4","2/5"
    ]
```

The order of the output must exactly match the order shown in the examples.

## Problem 3 (10 pts)

Define the following function:

```
argmin :: Ord a => (t -> a) -> [t] -> t
```

*argmin*, takes a function $f$ and a list $ls$. The function $f$ takes ane element of $ls$ and returns something that is a member of *Ord*. You must then return the minimum element of the list after $f$ has been applied to each element of $ls$.

For example:

```
argmin length [ "ABC", "EF", "GHIJ", "K" ] ==> "K"
argmin length [ "ABC", "DEF" ]             ==> "ABC"
```

Note if there is a tie pick the first value.

## Problem 4 (30 pts)

A DNA molecule is a sequence of four bases which are represented by 'A', 'G', 'C', 'T'. There are four different mutations that can occur on DNA:

- insertions: A base is inserted between two adjacent bases
- deletions: A base is deleted
- substitutions: A base is replaced with another base
- transpositions: Two adjacent bases are exchanged

You must define the following four functions:

```
insertions :: String -> [String]
deletions :: String -> [String]
substitutions :: String -> [String]
transpositions :: String -> [String]
```

Each function takes in a DNA sequence represented as a String and outputs all possibilities of applying the given transformation to the given DNA sequence. For example:

```
insertions "GC"       ==> [ "AGC", "GAC", "GCA", "GGC", "GGC", "GCG",
                            "CGC", "GCC", "GCC", "TGC", "GTC", "GCT" ]

deletions "AGCT"      ==> [ "GCT", "ACT", "AGT", "AGC" ]

substitutions "ACT"   ==> [ "ACT", "AAT", "ACA", "GCT", "AGT", "ACG",
                            "CCT", "ACT", "ACC", "TCT", "ATT", "ACT" ]

transpositions "GATC" ==> [ "AGTC", "GTAC", "GACT" ]
```

Order of the output does not matter.