Homework 5 — assigned Sunday April 22— due Monday May 7

General instructions

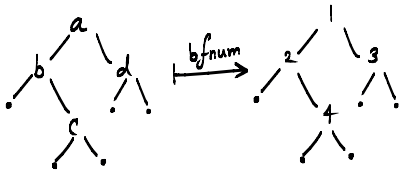
A skeleton literate Haskell source file homework5.lhs will be provided with the type signatures of the functions you are to write. Please edit that file and submit. The skeleton file will start with a few `import` declarations for the Haskell libraries we expect you to find useful. You may add other `import` declarations as you see fit.

5.1 Trees (40pts)

Given a type of labelled binary trees

```
data Tree a = E
            | T a (Tree a) (Tree a)
```

write a function `bfnum :: Tree a -> Tree Int` to create a tree of the same shape as the input tree, but with the values of the nodes replaced with the numbers $1 \ldots N$ in breadth-first order, where N is the number of internal `T` nodes in the input tree. For instance,
`bfnum (T 'a' (T 'b' E (T 'c' E E)) (T 'd' E E))`
evaluates to
`T 1 (T 2 E (T 4 E E)) (T 3 E E)`



In your solution, aim for correctness, simplicity, elegance, and, preferably, *algorithmic efficiency*. It is essential that you explain all the code you write in the comments (in the literate Haskell style; Latex preferred but not required). You should also explain how you designed your code. Comment on the the algorithmic efficiency of your code. (For instance, how does it compare to the obvious solution in an imperative setting?)

5.2 Expression trees (30pts)

We use the following data type declaration to introduce a language of simple arithmetic expressions, with variables and binding:

```
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 = Identifier -> 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'
```

Declare data type `Expr` as an instance of the type class `Show` such that `Expr` can be printed in human-friendly form. For instance,

```
show (Let "x" (Num 3) (Add (Var "x") (Num 5)))
```

might evaluate to `"let x = 3 in x + 5 end"`.

Write a function `evalInEnv`, with type `Env -> Expr -> Integer`, which computes the arithmetic value of an expression (which may have free variables) in a given environment (a mapping from variables to `Integer` values).

Then define:

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

so that `eval` evaluates closed expressions (expressions without free variables).

Example usage:

```
evalInEnv emptyEnv (Let "x" (Num 3) (Add (Var "x") (Num 5)))
```

evaluates to 8.

5.3 Infinite lists (30pts)

Please recall (e.g., from CS261) the trick for putting the positive rational numbers in one-to-one correspondence with the natural numbers. Given an infinite 2D table of fractions

```
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 3/5 4/5 5/5 ...
.         .    .    .
.         .     .     .
.         .      .      .
```

you run through it in diagonal slices to create a 1D sequence that eventually gets to every entry in the 2D table, as follows

```
1/1   2/1 1/2   3/1 2/2 1/3   4/1 3/2 2/3 1/4   5/1 4/2 3/3 2/4 1/5   ...
```

Your task is to write a function that carries out this diagonalization:

```
-- Take a list of lists, all potentially infinite.  Return a single
-- list which hits each element after some time.
diag :: [[a]] -> [a]
```

which embodies this transformation. You pass `diag` an infinite list of infinite lists, and it returns an infinite list, formed in the fashion above, of all the elements in the structure it was passed.

To test the function `diag`, we write the following test code:

```
-- The standard table of all positive rationals, in three forms:
-- (1) as floats
rlist = [ [i/j | i<-[1..]]  |  j <- [1..] ]
-- (2) as strings, not reducd
qlist1 = [ [show i ++ "/" ++ show j | i<-[1..]]  |  j <- [1..] ]
-- (3) as strings, in reduced form
qlist2 = [ [fracString i j | i <- [1..]]  |  j <- [1..] ]

-- take a numerator and denominator, reduce, and return as string
fracString num den = if denominator == 1
                then show numerator
                else 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 big list of lists
block n x = map (take n) (take n x)
```

Now `block 5 qlist2` evaluates to

```
[["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"]]
```

Meanwhile, `take 20 (diag qlist2)` should evaluate to

```
["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"]
```

How to turn in

Use the UNM Learn facility as follows: Navigate to `https://learn.unm.edu/` and log in. Then click on CS-357L-000 (Spring 2018). Now click on Assignments in the left side navigation menu. After that, click on the appropriate homework assignment link. Now attach your `.hs` file(s) and click submit. You are allowed to submit as many times as you like but only the latest submission will be graded.