Exercise Set 3 — 19 September

Fragments

3.1 Evaluation

For each of the following well-typed expressions, provide a correct type and then determine whether the expression has a value, and, if not, whether it loops forever, or raises an error. For each expression, either give its value, or write "diverges" if it loops forever, or else write "fails". (If the value is a function, just write "a function".)

-
  ```
  \n -> n ‘div‘ 0
  ```

-
  ```
  (\n -> n ‘div‘ 0) 5
  ```

-
  ```
  j j
  where
    j x  =  j x
  ```

-
  ```
  f (2, [1,2,3])
  where
    f (n, x:xs)  =  f (n-1, xs)
    f (0, xs)    =  xs
  ```

-
  ```
  g 5
  where
    g 0  =  1
    g n  =  2 * g (n-2)
  ```
- `d a`, in the scope of the following declarations:

  ```
  data T a  =  A [T a]
  a         =  A [A [A [A [], A [A [A []]]], A [A [], A []]]]
  m         =  foldr max 0
  d (A [])  =  1
  d (A c)   =  2 * m (map d c)
  ```

3.2 Fill in the blanks

The following function `nondec` determines if the elements of a list appear in nondecreasing order. Complete the definition, i.e., provide the missing right-hand side of the definition of `f`. Do not change anything else.

```
nondec :: Ord alpha => [alpha] -> Bool
nondec xs = f (zip xs (tail xs))
            where f =
```

Small programs

3.3 Simple functions on numbers

The Goldbach conjecture states that any even number greater than two can be written as the sum of two prime numbers. Using list comprehensions, write a function

```
goldbach :: Int -> [(Int,Int)]
```

which, when given an even number n, returns a list of all pairs of primes which sum to n. Note: You will have to write a function which tests an integer for primality and this should be written as a list comprehension also. For example, `goldbach 6` should evaluate to `[(3,3)]`. When the two primes in the pair are unequal, report them only once, smaller prime first. Report the pairs in lexicographically sorted order. Thus, `goldbach 20` should evaluate to `[(3,17),(7,13)]`.

3.4 Graphs

Many representations for graphs are possible, but here we use a very simple one. We represent a directed graph as a set of edges, and an edge as a pair of numbers which are the vertex labels:

```
type Graph = [(Int, Int)]
```

- Are all directed graphs representable in this fashion?
- (Consider the graph as undirected for this question.) Write a function `isTree :: Graph -> Bool` with the obvious meaning.
- Write a function `isDAG :: Graph -> Bool` with the obvious meaning.
- A rooted dag is a dag with a unique distinguished vertex (the root) from which all vertices are reachable. Write a function `isRootedDAG :: Graph -> Bool` with the obvious meaning.
- Assuming that the given graph is a rooted dag, write a function `calcDepth :: Graph -> [(Int, Int)]` to produce a list of pairs of the form (vertex, depth), with one pair for each vertex in the graph, where the depth is the length of the shortest path from the root to the vertex.

3.5 Paths in trees

We can define binary trees without any interesting content as follows:

```
data T = Leaf | Node T T
```

A path from the root to any subtree consists of a series of instructions to go left or right, which can be represented using another datatype:

```
data P = GoLeft P | GoRight P | This
```

where the path `This` denotes the whole tree. Given some tree, we would like to find all paths, i.e., the list of all paths from the root of the given tree to each of its subtrees. Write a function `allpaths :: T -> [P]` to do so.

For instance, `allpaths (Node Leaf (Node Leaf Leaf))` should evaluate to
`[This,GoLeft This,GoRight This,GoRight (GoLeft This),GoRight (GoRight This)]`
(but the ordering of the paths is immaterial).

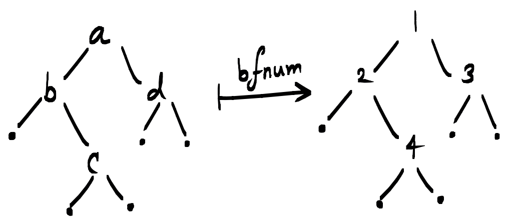
3.6 Renumbering trees

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. Aim for correctness, simplicity, elegance, and, preferably, *algorithmic efficiency*.

Example:
`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)`



3.7 Infinite lists

Recall 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"]
```