Mid-term examination #1 — given Tuesday 5th October

General instructions

Closed-book, closed-notes, closed-computer, in-class exam.

Time allowed: 75 minutes.

Total points available: 150 pts.

Answer in the spaces provided.

Your name (print):

I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.

Please sign and date:

1.1 Evaluation in Haskell (30 pts)

For each of the following Haskell expressions (5 pts each), indicate whether evaluation of the expression either:

- reduces to a value (in which case, you must **identify that value**),
- fails to terminate, or
- raises a runtime error.

In each case, you must **provide an explanation for your answer.**

1. `foldl (-) 1 [2, 4..]`

2. `foldl (-) 1 (take 3 [2, 4..])`

3. `foldr (-) 1 (take 3 [2, 4..])`

4. `(\x -> \y -> tail (tail (y:x))) [3, 9] 5`

5. `(\x -> \y -> tail (tail (y:x))) []`

6. `(\x -> \y -> tail (tail (y:x))) [] 5`

1.2 List operations in Haskell (40 pts)

1. (20 pts) Write an implementation of the Haskell function:

```
myZip :: [a] -> [b] -> [(a,b)]
```

   Your function should take two lists as inputs and returns a list of pairs, where the first pair in the returned list consists of the first elements of the two input lists (with the first and second elements of the pair coming from the first and second input lists, respectively), the second pair in the returned list consists of the second elements of the two input lists, and so on until one (or both) input lists runs out of elements.

   For example, `myZip [1,2,3,4] [5,6,7]` should evaluate to `[(1,5),(2,6),(3,7)]`.

2. (20 pts) Name and briefly explain the feature of Haskell evaluation means that Haskell programs can manipulate infinite lists without necessarily looping forever.

   Illustrate your answer by reference to the example expression: `myZip [1,2,3] [2, 4..]`

1.3 Tree operations in Haskell (40 pts)

Here is the definition of a Haskell type of binary trees with exactly two sub-trees per `Node`, with data values stored in the `Leafs` but not in the `Nodes`:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a) deriving Show
```

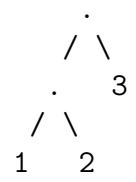
where the first and second arguments to `Node` represent the "left" and "right" sub-trees of that node, respectively.

1. (10 pts) Write a Haskell function

   ```
   treeDepth :: (Tree a) -> Int
   ```

   that returns the *depth* of a tree, defined as the longest path from the root of the tree to any of the `Leafs`. (The depth of a tree consisting of just a single `Leaf` node is zero by definition.)

   For example, let `t` be the Haskell value `Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)`, which represents the following tree:

   Then, the expression `treeDepth t` should evaluate to `2`.

2. (15 pts) Write a Haskell function

```
leafValuesAtDepth :: Int -> (Tree a) -> [a]
```

such that `leafValuesAtDepth n t` returns the list of values stored in the `Leaf`s of the tree `t` at depth `n`. The values should be ordered from left to right within the layer at depth `n`.

For example, with the Haskell value `t` defined as above, the expression

```
leafValuesAtDepth 2 t
```

should evaluate to `[1,2]`.

3. (15 pts) Hence, or otherwise, write a Haskell function

```
treeBreadthFirst :: (Tree a) -> [a]
```

such that `treeBreadthFirst t` returns all values stored in `Leafs` within the input tree `t`, arranged as in a breadth-first tree traversal. That is, the values should be ordered by increasing depth in the tree and then from left to right for values at the same depth.

For example, with the Haskell value `t` defined as above, the expression

```
treeBreadthFirst t
```

should evaluate to `[3,1,2]`.

1.4 Proving properties of Haskell programs (40 pts)

Prove, by induction on lists, that

```
foldr (\x -> \acc -> (f x):acc) [] xs = map f xs
```

for all lists `xs :: [a]` and functions `f :: a -> b`.