Name: ______________________________

Computer Science 330

Spring 2018 Final

Doodle here.

This is an individual-effort exam. You may read notes, textbooks, or the internet, but you may not post questions to online forums or share answers with classmates. Meaningful attempts must be made when answering questions. Write things that are both true and relevant. Exams with any blank or perfunctory answers will not be awarded partial credit.

1. *Tightly Coupled*

   (a) Using generics, write a method `zip` in Java that zips together two `List`s into a single list of pairs. It behaves exactly like Haskell's `zip` function, which has this type signature: `[a] -> [b] -> [(a, b)]`. Use the `Tuple2` class from the Wasd homework to encapsulate a pair of values. (Do not write `Tuple2`; assume it already exists.) If the lists are of different lengths, the resulting list of pairs has the same length as the shorter of the two.

   (b) Write a `main` method that creates two lists of different types and zips them together. To keep things short, create a list by first creating an array and then calling `Arrays.asList` to convert it.

2. *Arborithms*

Consider the following generic tree class in Java:

```
public class Tree<T> {
  private T value;
  private Tree<T> leftChild;
  private Tree<T> rightChild;

  public Tree(T value, Tree<T> leftChild, Tree<T> rightChild) {
    this.value = value;
    this.leftChild = leftChild;
    this.rightChild = rightChild;
  }
}
```

(a) Write an assignment statement such that variable `root` refers to a `Tree` of `Integers` with at least four nodes.

(b) Write a functional interface named `Mixer<T, U>` that abstracts away the fold algorithm's mixing operation over trees. It imposes a `mix` method on its implementers. The method accepts three parameters in the following order: a generic `T` for the current node's value, a generic `U` for the left child's folded value, and a generic `U` for the right child's folded value. It returns a `U`.

(c) Write method `Tree.fold` that accepts a `Mixer` and a generic `U` representing the zero value. It mixes together the tree node's value and the values from folding the left and right children. If a child node doesn't exist (is null), the zero value is mixed in instead.

(d) Calculate the following values using *only* calls to `Tree.fold` and a lambda. The code should be short and sweet.

i. Assign to variable `sum` the sum of the nodes in the tree starting at `root`.

ii. Assign to variable `count` the number of nodes in the tree starting at `root`.

iii. Assign to variable `inorder` a `String` of bracketed node values enumerated in an in-order-traversal manner. If the root has 21, left child 17, and right child 3, then `inorder` has value `"[17][21][3]"`.

3. *Hall of Fame*

Define the following functions in Haskell using composition, partial evaluation of parameters, and higher-order functions like `filter`, `fold`, and `map`.

(a) `initials`, which accepts a list of list of `Strings` and returns a capitalized `String` comprised of the first characters. For example, `initials ["today", "i", "learned"]` → `"TIL"`. Use point-free style.

(b) `proportionalize`, which accepts a list of `Int` weights and returns the list in which each weight has been turned into its proportion out of the total weights. For example, `proportionalize [1, 2, 3]` → $[\frac{1}{6}, \frac{2}{6}, \frac{3}{6}] = [0.1\overline{6}, 0.\overline{3}, 0.5]$.

(c) `prodduct`, which accepts a list of `Ints` and returns the product of all the odd numbers in the list. For example, `prodduct [1..5]` → 15. Use point-free style.

(d) `fracture`, which accepts an `Int` and returns a list of all non-zero fractions stepping up to that number and having that number as their denominator. For example, `fracture 5` → $[\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, \frac{5}{5}] = [0.2, 0.4, 0.6, 0.8, 1.0]$.

4. *Languages You Don't Know*

   (a) Read *Tour of Scala: By-name Parameters*[1]. Then write a method `meanTime` that accepts an integer `n` and an operation to execute. The operation to execute is passed by name and returns nothing, which is called `Unit` in Scala. It executes the operation `n` times and returns the average time in nanoseconds that it takes to execute the operation. Use `System.nanoTime` to get the current time.

   (b) Read *Wondrous Oddities: R's Function-call Semantics*[2]. The combination of default and lazy parameters allows for a certain kind of computational magic. Write a function `box` that accepts six parameters: `x1`, `y1`, `x2`, `y2`, `width`, and `height`. Return a vector of the box's six properties. (Vectors in R are created with the `c` function, as shown in the reading.) Assign reasonable defaults to each using the other parameters. If you call this function with any two of the three horizontal parameters and any two of the three vertical parameters, the other two will get properly assigned. Assume that `x1` ≤ `x2` and `y1` ≤ `y2`.

[1] `https://docs.scala-lang.org/tour/by-name-parameters.html`

[2] `http://blog.moertel.com/posts/2006-01-20-wondrous-oddities-rs-function-call-semantics.html`