Homework 12

Due **Friday**, 5/3/2019

Please turn in your homework solutions as usual at `https://submit.cs.pomona.edu/2019sp/cs131`. You will be writing Grace programs for this assignment, so you will need to zip up the `.grace` files with your pdf before turning them both in. As usual be sure you turn in copies of all the code in your pdf file as well as the .grace files. We will have a very short window to grade this homework, so messing up the files turned in may result in not getting credit for your work.

Because this is your last homework and we have not yet covered all of the Java constructs to do the last three problems, you will have an extra day to work on this assignment. I urge you to work on the Grace programs early on – finishing them before Tuesday, leaving your self the end of the week to work on the Java-related questions.

1. `Grace program` (20 points)

   Look over the Grace papers on the CS131 "Links to useful information" web page before attempting the following programs. Use the Grace Draft Specification as a reference. Note that some of the syntax in the papers is a bit out of date as the language has evolved over time. In particular, class names no longer have "." in them. Thus use the language spec as the definitive source. You may also find the introductory textbook, Programming with Grace, useful as a source of sample programs.

   You can write and run Grace programs by going to the web page at `http://web.cecs.pdx.edu/~grace/ide/`. I recommend that you use Chrome for this as different browsers render the pages differently and the software has been tested most extensively in Chrome. See the instructions in lecture 24 on how to write and run Grace programs in the browser.

   Please use the names suggested in the problem for the file so that we can easily test them. While you should write test code for your programs, they should go in a separate file that you may name as you like. However, we will be using our own test code for your programs and it will expect your files and classes, methods, etc. to have exactly the names and types specified in the problem. You will lose points if your code does not follow those conventions.

   Warning: Minigrace files may not contain tabs. The only white spaces allowed are spaces and returns (generated by the "enter" key). You will receive error messages if your file contains any other white space.

   Write a Grace program that defines a class representing mutable (updatable) sets of Numbers. Your sets should implement the following type:

```
type NumberSet = {
    contains(n:Number) -> Boolean
    add(n:Number) -> Done
    union(s:NumberSet) -> Done
    intersection(s:NumberSet) -> Done
    isEmpty -> Boolean
    asString -> String
    do(action) -> Done
}
```

Notice that the operations `add`*,* `union`*, and* `intersection` *all modify the existing set rather than create a new one.* Please name the file `NumberSet.grace`. Your class should be named `aNumberSet`, should take no parameters, and should construct an empty set.

Grace actually has a library to represent sets, but I don't want you to use it. Your implementation should use an immutable list (use our implementation from class – available on the class web page) as an instance variable to hold the elements of the list. (Do NOT inherit from it.) Remember that sets do not have duplicates of elements, so your list implementation should not include duplicates.

The file with your copy of my list implementation should be named `list.grace`, and your file `NumberSet.grace` should import it. Thus the first line of `NumberSet.grace` should be

```
import "list" as myList
```

where you can replace myList by any legal identifier name. This is important as I will test your code by loading my copy of `list.grace` with your `NumberSet.grace` and my test code. My test code will import your code to test it, so all the pieces must fit together properly.

Show that your program works by creating a new file, where the first line of the new file is

```
import "NumberSet" as mySet
```

In that new file, build a couple of sets, and exercise the operations (showing corner cases work) and printing the results. Feel free to include the file with your test code. While I won't normally run it, if there are issues I can run your code to see if there is anything at all that works.

Because this program represents mutable sets, there is no need for separate representations for empty and non-empty sets. Those variations will be taken care of by the list instance variable.

2. (15 points) **More Grace**

Write a Grace program that represents a tree holding a number at each vertex. I suggest that you emulate the implementation of lists from class (remember that Grace does not have a null element!). Here is the type of the tree (and a type for arguments of the methods):

```
type Tree = {
    // return the number of nodes in the tree.
    size -> Number

    // return the height of the tree
    height -> Number

    // return whether the tree is empty
    isEmpty -> Boolean

    // Create a new tree by applying the unary operation blk on each of the elements
    // in the tree to form a new tree of the same shape.
    map (blk: Function1[[Number,Number]]) -> Tree
    // Perform an inorder traversal of the tree, applying unary operation blk
    // to each of the elements of the tree.
    doInorder (blk: Procedure1[[Number]]) -> Done
}
```

In the above, Function1 and Procedure1 are built-in types for Grace. They are defined in the standard library as follows:

```
type Function1[[S,T]] = {
   apply (s: S) -> T
}

type Procedure1[[S]] = {
   apply (s: S) -> Done
}
```

The first represents an anonymous function from S to T, while the second is a procedure that takes an argument of type S. As an example s = {`x: Number -> x * x`} has type `Function1[[Number,Number]]` and `s.apply(12)` returns 144. Do not copy the definitions of these types in your code as they are already defined!

As you can see from the above definitions, type parameters in Grace are written inside double square brackets.

Your program should define an empty tree object named `emptyTree` and contain a class `aTreeWithRoot (n: Number) left (l: Tree) right (r: Tree)` that constructs a new class with root holding `n` and left and right subtrees given by `l` and `r`. Place your code in a file named `Trees.grace`. My test code will be importing `Trees`, so make sure everything is named properly

As you can see from the operations in the type, the tree is immutable.

As usual, write and include code to test it in a separate file. In particular, build an interesting tree `myTree` and then execute

```
myTree.doInorder {n -> print (n)}
var count:Number := 0
myTree.doInorder {n -> count := count +1}
print(count)
```

3. (10 points) **Atomicity & Race Conditions**

   The `DoubleCounter` class defined below has methods `incrementBoth` and `getDifference`. Assume that `DoubleCounter` will be used in multi-threaded applications.

```
class DoubleCounter {
   protected int x = 0, y = 0;

   public int getDifference() {
      return x - y;
```

```
        }

        public void incrementBoth() {
           x++;
           y++;
        }
     }
```

There is a potential data race between `incrementBoth` and `getDifference` if `getDifference` is called between the increment of x and the increment of y. For the following questions, assume that the "++" operators are atomic (in reality they are not) and that each thread calls incrementBoth exactly once.

(a) What are the possible return values of `getDifference` if there are 2 threads?

(b) What are the possible return values of `getDifference` if there are n threads?

(c) Data races can be prevented by inserting synchronization primitives. One option is to declare

```
public synchronized int getDifference() {...}
public void incrementBoth() {...}
```

This will prevent two threads from executing method `getDifference` at the same time. Is this enough to ensure that `getDifference` always returns 0? Explain briefly.

(d) Is the following declaration

```
public int getDifference() {...}
public synchronized int incrementBoth() {...}
```

sufficient to ensure that `getDifference` always returns 0? Explain briefly.

(e) What are the possible values of `getDifference` if the following declarations are used?

```
public synchronized int getDifference() {...}
public synchronized int incrementBoth() {...}
```

4. (10 points) **Concurrent Access to Objects**

   Please do problem 14.6 from Mitchell, page 471.

5. (10 points) **Java synchronized objects**

   Please do problem 14.7, parts a,b,and c from Mitchell, page 472.