CSE 216 – Homework III

Instructor: Dr. Ritwik Banerjee

This homework document consists of 4 pages. Carefully read the entire document before you start coding. Unlike the previous assignments, this assignment requires you to write code in both Python (Python 3.x) and Java (JDK 1.8).

1 Functional Programming in Java [55 points]

You are required to write a few `public static` methods, where the signature and documentation of each method is provided below. Additionally, there are implementation and code-style requirements specified for each method.

```
return sequence.stream()
            .intermediate_operation_1(...)
            .intermediate_operation_2(...)
            .intermediate_operation_3(...).terminal_operation();
```

Example 1: A Java function implemented as a single method chain.

1.1 Using streams with Java [25 points]

Each function implementation must be done using a single method chain (as shown in example 1 above), and all the functions must be implemented in a file named `FunctionalUtils.java`:

1. **Capitalized strings.** [5 points]

```
/**
 * @param  strings: the input collection of <code>String</code>s.
 * @return          a collection of <code>String</code>s that start with a
 *                  capital letter (i.e., 'A' through 'Z').
 */
public static Collection<String> capitalized(Collection<String> strings);
```

2. **The longest string.** [5 points]

```
/**
 * Find and return the longest <code>String</code> in a given collection of <code>String</code>s.
 *
 * @param strings:    the given collection of <code>String</code>s.
 * @param from_start: a <code>boolean</code> flag that decides how ties are broken.
                      If <code>true</code>, then the element encountered earlier in
 *                    the iteration is returned, otherwise the element encountered
 *                    later is returned.
 * @return            the longest <code>String</code> in the given collection,
 *                    where ties are broken based on <code>from_start</code>.
 */
public static String longest(Collection<String> strings, boolean from_start);
```

3. **The least element.** [10 points]
   In this function, the single method chain can return a `java.util.Optional<T>`. Therefore, you must write additional code to convert that object to an object of type `T` (handling any potential exceptions in the process).

```
/**
 * Find and return the least element from a collection of given elements that are comparable.
 *
```

```
 * @param items:      the given collection of elements
 * @param from_start: a <code>boolean</code> flag that decides how ties are broken.
 *                    If <code>true</code>, then the element encountered earlier in
 *                    the iteration is returned, otherwise the element encountered
 *                    later is returned.
 * @param <T>:        the type parameter of the collection (i.e., the items are all of type T).
 * @return            the least element in <code>items</code>, where ties are
 *                    broken based on <code>from_start</code>.
 */
public static <T extends Comparable<T>> T least(Collection<T> items, boolean from_start);
```

4. **Flatten a map.** [5 points]

```
/**
 * Flattens a map to a stream of <code>String</code>s, where each element in the list
 * is formatted as "key -> value".
 *
 * @param aMap the specified input map.
 * @param <K>  the type parameter of keys in <code>aMap</code>.
 * @param <V>  the type parameter of values in <code>aMap</code>.
 * @return the flattened list representation of <code>aMap</code>.
 */
public static <K, V> List<String> flatten(Map<K, V> aMap)
```

1.2 Higher-order functions in Java [30 points]

To complete the code for this section, you may have to read some of the official Java documentation on the classes that implement higher-order functions in Java. All the code must be written in a file named `FunctionalOperations.java`.

1. First, write a nested interface in `FunctionalOperations` called `NamedBiFunction` that extends the interface `java.util.Function.BiFunction`. Your interface should just have one method declaration: `String name();`, any class that implements this interface should provide a "name" for every instance of that class. [5 points]
2. Next, create `public static NamedBiFunction` instances as follows: [5 points]
   (a) `add`, with the name "add", to perform addition of two `Double`s.
   (b) `subtract`, with the name "diff", to perform subtraction of one `Double` from another.
   (c) `multiply`, with the name "mult", to perform multiplication of two `Double`s.
   (d) `divide`, with the name "div", to divide one `Double` by another. This operation should throw a `java.lang.ArithmeticException` if there is a division by zero being attempted.
3. Write a function called `zip` as follows: [10 points]

```
/**
 * Applies a given list of bifunctions -- functions that take two arguments of a
 * certain type, and produce a single instance of that type -- to a list of
 * arguments of that type. The functions are applied in an iterative manner, and
 * the result of each function is stored in the list in an iterative manner as
 * well, to be used by the next bifunction in the next iteration.
 * For example, given
 *   List<Integer> args = [1,1,3,0,4], and
 *   List<BiFunction<Double, Double, Double>> bfs = [add, multiply, add, divide],
 * <code>zip(args, bfs)</code> will proceed iteratively as follows:
 *   - index 0: the result of add(1,1) is stored in args[1] to yield args = [1,2,3,0,4]
 *   - index 1: the result of multiply(2,3) is stored in args[2] to yield args = [1,2,6,0,4]
 *   - index 2: the result of add(6,0) is stored in args[3] to yield args = [1,2,6,6,4]
 *   - index 3: the result of divide(6,4) is stored in args[4] to yield args = [1,2,6,6,1]
 *
 * @param args:        the arguments over which <code>bifunctions</code>
 *                     will be iteratively applied.
 * @param bifunctions: the given list of bifunctions that will iteratively be
 *                     applied on the <code>args</code>.
 * @param <T>:         the type parameter of the arguments (e.g., Integer, Double)
 * @return             the last element in <code>args</code>, the final result of
 *                     all the bifunctions being applied in sequence.
 */
public static <T> T zip(List<T> args, List<NamedBiFunction<T, T, T>> bifunctions);
```

4. Based on the above `zip` function, think about what a function composition would look like. Write a static inner class called `FunctionComposition` that is parameterized by three type parameters. This class should have no methods, and no constructor. It should only have a single `BiFunction` called `composition`, which takes in two functions and provides their composition as the output function. Function composition should be consistent with the types – if there is a function `f: char -> String`, and another function `g: String -> int`, the output of composition should be a function `h: char -> int`. For example, if `f` concatenates a `char` some number of times (say, 'b' yields "bb", 'c' yields "ccc", 'd' yields "dddd", etc.), and `g` converts a string to its length, then `composition(f, g)` should output a function that maps 'z' to 26. [10 points]

2 Functional Programming in Python [45 points]

For this section, you are required to write functions in a file named `functional.py`.

2.1 Recursion in Python [5 + 5 + 5 = 15 points]

Use recursion for the following functions:

1. Write a function `flatten` to flatten a list where some items in the list could, themselves, be lists. We will call these *nested lists*.

```
>>> flatten([ [1, 2, [3, 4] ], [5, 6], 7])
[1, 2, 3, 4, 5, 6, 7]
```

2. Write a function `reverse` to reverse a nested list, while still maintaining the nested structures.

```
>>> reverse([[1, 2], [3, [4, 5]], 6])
[6, [[5, 4], 3], [2, 1]]
```

3. Write a function `compress` to remove consecutive duplicates from a list, and return the results as a new list. The original list must remain unmodified.

```
>>> compress([1, 1, 4])
[1, 4]
```

2.2 Higher-order functions [5 + 10 + 5 + 10 = 30 points]

Lambda expressions in Python have the syntax `lambda <vars>: <body>`. For example, `double = lambda x: 2*x` and `add = lambda x,y: x+y` create the two functions using simple lambda expressions. Functions can be used with standard higher-order functions like `map()`, `filter(), and reduce()`. Some simple illustrative examples are:

```
>>> v = [ [2, 3, 5, 7], [11,13,17,19], [23, 29], [31,37] ]
>>> list(map(len, v))
[4, 4, 2, 2]
>>> items = [1, 2, 3, 4, 5]
>>> list(map(lambda x: x**2, items))
[1, 4, 9, 16, 25]
>>> all = range(-4,5)
>>> list(filter(lambda x: x > 0, all))
[1, 2, 3, 4]
>>> from functools import reduce
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

Use lambda expressions and the above higher-order functions to implement the following. Each function implementation should effectively have one line of code in the function body.

1. The Python equivalent of the `capitalized()` function in Sec. 1.1. The method signature must be

```
def capitalized(items: list) -> list
```

2. The Python equivalent of the `longest()` function in Sec. 1.1. The method signature must be

```
def longest(strings: list, from_start=True) -> object
```

3. A higher-order function called `composition(f,g)`, which takes as input two input functions `f` and `g`, and returns a function that is their composition. For example, if `f` and `g` are the increment and square-root functions (respectively), then `composition(f,g)` would map 15 to 4 (`15 -> 15+1 -> 4`).

4. Generalize the above to a composition of a list n functions, using the signature

```
def n_composition(*functions)
```

NOTES:

- As always, **late submissions** or **uncompilable code** will not be graded.
- Please remember to verify what you are submitting. Make sure you are, indeed, submitting what you think you are submitting!
- What to submit? A single `.zip` file containing the two `.java` files and the one `.py` file. Since the method/function definitions are precise, you do NOT have to write your own test cases for grading (or course, you should always use test cases yourself to make sure your code is free of bugs). **This assignment may be graded by a script, so be absolutely sure that the submission follows this structure**.

Submission Deadline: Apr 21, 2019, 11:59 pm