Programming Abstractions – Homework III

Dr. Ritwik Banerjee
Computer Science, Stony Brook University

This homework document consists of 5 pages, and focuses on functional programming using Java. While doing this assignment, you will notice that to use the functional programming paradigm with Java, you will also need to be very careful about the use of parameterized types.

Development Environment: It is highly recommended that you use IntelliJ IDEA. You can, if you really want, use a different IDE, but if things go wrong there, you may be on your own.

Programming Language: Just like the previous assignment, all Java code *must* be JDK 1.8 compliant.[1]

1. Simple functional programming in a single method chain

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

```
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. The least element: In this function, the single method chain can return a `java.util.Optional<T>`. So you must write something extra (your final code should still be a single method chain) to convert it to an object of type `T` (handling any potential exceptions). (10)

```
/**
 * 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>, the element encountered earlier in the
 *                    iteration is returned, otherwise the later element 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);
```

2. Flatten a map: (10)

```
/**
 * Flattens a map to a list of <code>String</code>s, where each element in the list is formatted
 * as "key -> value" (i.e., each key-value pair is converted to a string in this specific format).
 *
 * @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] You may have a higher version of Java installed, but the "language level" must be set to Java 8. This can be easily done in IntelliJ IDEA by going to "Project Structure" and selecting the appropriate "Project language level". This is a very important requirement, since Java 9 (and beyond) has additional language features that will not compile with a Java 8 compiler.

2. Higher order functions

Code for the following questions must be written in a file named `HigherOrderUtils.java`. You may also have to consult some of the official Java documentation and/or the reference text on Functional Programming in Java. The remaining answers need not be written as a single method chain. *Be very careful with the parameterized types in this section, and pay attention to the warnings issued by your IDE*.

3. First, write a static nested interface in `HigherOrderUtils` called `NamedBiFunction`. This interface must extend the interface `java.util.Function.BiFunction`. The interface should just have one additional method declaration: `String name();`, i.e., a class implementing this interface must provide a "name" for every instance of that class. Then, create `public static NamedBiFunction` instances as follows: (15)
   (a) `add`, with the name "plus", to perform addition of two `Double`s.
   (b) `subtract`, with the name "minus", to subtract the second `Double` *from* the first.
   (c) `multiply`, with the name "mult", to perform multiplication of two `Double`s.
   (d) `divide`, with the name "div", to divide the first `Double` by the second. This operation should throw a `java.lang.ArithmeticException` if there is a division by zero being attempted.

4. Write a method called `zip`, defined as follows: (15)

```
/**
 * 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<Double> args = Arrays.asList(-0.5, 2d, 3d, 0d, 4d), and
 * List<NamedBiFunction<Double, Double, Double>> bfs = Arrays.asList(add, multiply, add, divide),
 * <code>zip(args, bfs)</code> will proceed as follows:
 * - the result of add(-0.5, 2.0) is stored in index 1 to yield args = [-0.5, 1.5, 3.0, 0.0, 4.0]
 * - the result of multiply(1.5, 3.0) is stored in index 2 to yield args = [-0.5, 1.5, 4.5, 0.0, 4.0]
 * - the result of add(4.5, 0.0) is stored in index 3 to yield args = [-0.5, 1.5, 4.5, 4.5, 4.0]
 * - the result of divide(4.5, 4.0) is stored in index 4 to yield args = [-0.5, 1.5, 4.5, 4.5, 1.125]
 *
 * @param args          the arguments over which <code>bifunctions</code> will be applied.
 * @param bifunctions   the list of bifunctions that will be applied on <code>args</code>.
 * @param <T>           the type parameter of the arguments (e.g., Integer, Double)
 * @return              the item in the last index of <code>args</code>, which has the final result
 *                      of all the bifunctions being applied in sequence.
 *
 * @throws IllegalArgumentException if the number of bifunction elements and the number of argument
 *                                  elements do not match up as required.
 */
public static <T> T zip(List<T> args, List<BiFunction<T, T, T>> bifunctions);
```

Now, is this method following the principles of functional programming? If yes, your implementation should also be true to the principles of the functional programming paradigm. Otherwise, you need not abide by them when implementing `zip`. Next, notice that the documentation mentioned bifunctions, and the method signature uses `BiFunction`s, but the example provided in the documentation uses `NamedBiFunction`s. Your implementation should be able to handle both! Specifically, the following code should work with your implementation:

```
public static void main(String... args) {
    List<Double> numbers = Arrays.asList(-0.5, 2d, 3d, 0d, 4d); // documentation example
    List<NamedBiFunction<Double, Double, Double>> operations = Arrays.asList(add,multiply,add,divide);
    Double d = zip(numbers, operations); // expected correct value: 1.125
    // different use case, not with NamedBiFunction objects
    List<String> strings = Arrays.asList("a", "n", "t");
    // note the syntax of this lambda expression
    BiFunction<String, String, String> concat = (s, t) -> s + t;
    String s = zip(strings, Arrays.asList(concat, concat)); // expected correct value: "ant"
}
```

In order to have the above code compile and run correctly, you will need to make a very specific change to the signature of the `zip` method.

3. Function groups

In the previous assignment, you were introduced to the definition of groups. And now, we are working with functions as first-class objects. It's time to combine the two! The definition is repeated below:

> A nonempty set of elements G forms a **group** if in G there is a defined binary operation (which we will denote by $\cdot$ in this document), such that
>
> 1. $x, y \in G$ implies that $x \cdot y \in G$. This property is called *closure*, and the set of elements is said to be *closed under the operation*.
> 2. $a, b, c \in G$ implies that $a \cdot (b \cdot c) = (a \cdot b) \cdot c$. In other words, the binary operation is associative.
> 3. There exists an element $e \in G$ such that $a \cdot e = e \cdot a = a$ for all elements $a \in G$. This special element e is called the *identity* element of the group.
> 4. For every $a \in G$, there exists an element b such that $a \cdot b = b \cdot a = e$. That is, every element has an *inverse*. Often, we simply denote it by a^{-1}.

From this definition, we can see that it requires a set of elements to be defined. Only then can we talk about the binary operation and other group properties (because those properties depend on the set). Given a set, we can define functions from the set to itself. For example, if $S = \{1, 2\}$, we can define functions $f : S \longrightarrow S$. For this particular set $\{1, 2\}$, there are only two one-to-one and onto functions:

1. The identity function, which maps each element to itself, i.e., $f(x) = x \; \forall \, x \in S$.
2. The other is the function $f(1) = 2; f(2) = 1$.

It turns out that given a set S, the set of all one-to-one and onto functions (more commonly called "bijections") defined on it, form a group under function composition as the binary operation.

Note: One-to-one functions are usually known as "injections", and onto functions are known as "surjections". A bijection is a function that is both.

5. To see how this works, your first task in this section is to implement a method called `bijectionsOf`. This method must be written in a class called `BijectionGroup`. (25)

   A part of its signature is given below (and the missing blank, which may be more than one item of the method signature, is something you have to figure out):

   ```
   public static ________________________ bijectionsOf(Set<T> domain);
   ```

   The job of this `bijectionsOf(Set<T>)` method is to take a set as its input argument, and return the set of all the bijections of the input set. Specifically, the following code must compile and run with your implementation:

   ```
   public class BijectionGroup {

       // your methods go here

       public static void main(String... args) {
           Set<Integer> a_few = Stream.of(1, 2, 3).collect(Collectors.toSet());
           // you have to figure out the data type in the line below
           ______________________________ bijections = bijectionsOf(a_few);
           bijections.forEach(aBijection -> {
               a_few.forEach(n -> System.out.printf("%d --> %d; ", n, aBijection.apply(n)));
               System.out.println();
           });
       }
   }
   ```

   If your implementation is correct, this should print the following lines:

   ```
   1 --> 3; 2 --> 1; 3 --> 2;
   1 --> 2; 2 --> 1; 3 --> 3;
   1 --> 2; 2 --> 3; 3 --> 1;
   1 --> 3; 2 --> 2; 3 --> 1;
   1 --> 1; 2 --> 2; 3 --> 3;
   1 --> 1; 2 --> 3; 3 --> 2;
   ```

The order of the lines can be different, but if any other line in addition to the above exist in your output, or if some of the above lines are missing, then it indicates a mistake in your implementation.

6. Your next task is to provide the mathematical group structure (as defined at the beginning of this section) to the set of all bijections of a given set. (25)

   The group interface is the same as the previous assignment, reproduced below:

```
public interface Group<T> {

    /**
     * Performs the binary operation, as defined by the group, of one object with the other specified
     * object. The implementer must take care to ensure that the binary operation is
     * <ul>
     *     <li><b>closed</b> for the parameter type <code>T</code>. That is, the result of the binary
     *         operation is a valid member of the set that defines the type <code>T</code> (taking the
     *         denotational semantics view of data types). For example, addition is a binary operation
     *         that is closed for integers, but division is not.</li>
     *     <li><b>associative</b>. That is, for any elements <code>x</code>, <code>y</code>, and
     *        <code>z</code> in this group, <code>binaryOperation(binaryOperation(x, y), z)</code> is
     *        equal to <code>binaryOperation(x, binaryOperation(y, z))</code>. For example, addition is
     *        an associative binary operation for integers.</li>
     *     <li>respectful of the identity element. That is, for any element <code>x</code> in this
     *        group and the identity element <code>e</code> of this group,
     *       <code>binaryOperation(x, e)</code> is equal to <code>x</code>, and
     *       <code>binaryOperation(e, x)</code> is also equal to <code>x</code>. For example,
     *       <code>0</code> is the identity element of the group of integers under addition.</li>
     * </ul>
     *
     * @param one   the object that is the first argument of the binary operation.
     * @param other the other object (the second argument of the binary operation) to be combined with
     *              this object as per the group's binary operation.
     * @return the result of the binary operation on this object with the other specified object.
     */
    T binaryOperation(T one, T other);

    /**
     * @return the identity element of this group.
     */
    T identity();

    /**
     * In a group, every element <code>x</code> must have its inverse, which is an element <code>y</code>
     * in the group such that <code>binaryOperation(x, y)</code> is equal to
     * <code>binaryOperation(y, x)</code>, and both yield the identity element of the group. For example,
     * for the group of integers under addition, the negative of any integer is its inverse.
     *
     * @return the inverse of this object.
     */
    T inverseOf(T t);

    /**
     * This is a utility function, serving as the definition of exponentiation for this group.
     * Exponentiation is defined as <code>exponent(t, 0)</code> being the <code>identity()</code>
     * element, and <code>exponent(t, n)</code> being <code>binaryOperation(t, exponent(t, n-1))</code>.
     *
     * @param t the group element serving as the base.
     * @param k the integer exponent, indicating the number of times the binary operation is applied
     *          on <code>t</code>. The exponent must be a non-negative integer value.
     * @return the result of the binary operation applied <code>k</code> times on <code>t</code>.
     */
    default T exponent(T t, int k) {
        if (k < 0)
            throw new IllegalArgumentException("The exponent must be a non-negative integer value.");
        return k == 0 ? identity() : binaryOperation(t, exponent(t, k - 1));
    }
}
```

   In the same class, `BijectionGroup`, write a static method called `bijectionGroup`. This method should

take a finite set of items as its only argument, and return the group of its bijections. Specifically, the following code must compile and run with your implementation:

```
public class BijectionGroup {

    // your code goes here

    public static void main(String... args) {
        Set<Integer>              a_few = Stream.of(1, 2, 3).collect(Collectors.toSet());

        // you have to figure out the data types in the lines below
        // some of these data types are functional objects, so look into java.util.function.Function
        ___________________________ g     = bijectionGroup(a_few);
        ___________________________ f1    = bijectionsOf(a_few).stream().findFirst().get();
        ___________________________ f2    = g.inverseOf(f1);
        ___________________________ id    = g.identity();
}
```

For the sake of explanation, let us continue to follow the same format of printing as in the previous question. If, say, `f1` is the element

```
1 --> 2; 2 --> 1; 3 --> 3;
```

then its inverse element, obtained from `g.inverseOf(f1)` in the above code, must be (and you can do a quick manual verification of this)

```
1 --> 2; 2 --> 1; 3 --> 3;
```

That is, `f1` is its own inverse!

Similarly, the identity element of this group must be

```
1 --> 1; 2 --> 2; 3 --> 3;
```

- Please keep in mind these homework-related points mentioned in the syllabus.
- **What to submit?** The complete codebase (including classes and interfaces that were already given to you, including the `Group` interface) as a single `.zip` file. Your zip file, once extracted, must not contain any further packages or subfolders.
  Deviations from the expected submission format carries varying degrees of score penalty (depending on the amount of deviation).

Submission Deadline: Apr 27 (Thursday), 11:59 pm