CS2030 Programming Methodology
Semester 2 2019/2020

26 March 2020
Problem Set #8

1. **Currying** is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument, $g(x, y) = h(x)(y)$. Using the context of lambdas in Java, the lambda expression `(x, y) -> x + y` can be translated to `x -> y -> x + y`.

   Show how the use of appropriate functional interfaces can achieve the curried function evaluation of two arguments.

   Hint: If the lambda above looks intriguing, try replacing the lambda with anonymous inner classes instead to make sense of the scope of the variables `x` *and* `y`.

2. The following depicts a classic tail-recursive implementation for finding the sum of values of n (given by $\sum_{i=0}^{n} i$) for $n \geq 0$.

```
static long sum(long n, long result) {
    if (n == 0) {
        return result;
    } else {
        return sum(n - 1, n + result);
    }
}
```

   In particular, the implementation above is considered **tail-recursive** because the recursive function is at the tail end of the method, i.e. no computation is done after the recursive call returns. As an example, `sum(100, 0)` gives `5050`.

   However, this recursive implementation causes a `java.lang.StackOverflowError` error for large values such as `sum(100000, 0)`.

   Although the tail-recursive implementation can be simply re-written in an iterative form using loops, we desire to capture the original intent of the tail-recursive implementation using delayed evaluation via the `Supplier` functional interface.

   We represent each recursive computation as a `Compute<T>` object. A `Compute<T>` object can be either:

   - a recursive case, represented by a `Recursive<T>` object, that can be recursed, or
   - a base case, represented by a `Base<T>` object, that can be evaluated to a value of type `T`.

As such, we can rewrite the sum method as:

```
static Compute<Long> sum(long n, long s) {
    if (n == 0) {
        return new Base<>(() -> s);
    } else {
        return new Recursive<>(() -> sum(n - 1, n + s));
    }
}
```

and evaluate the sum of n terms via the summer method below:

```
    static long summer(long n) {
        Compute<Long> result = sum(n, 0);

        while (result.isRecursive()) {
            result = result.recurse();
        }

        return result.evaluate();
}
```

(a) Complete the program by writing the `Compute`, `Base` and `Recursive` classes.

(b) By making use of a suitable client class `Main`, show how the "tail-recursive" implementation is invoked

(c) Redefine the `Main` class so that it now computes the factorial of n recursively.

3. **Lazy Values** are useful for cases where computing the value is expensive, but the value itself might not eventually be used. Why compute what you might not even use? Unlike other languages such as Scala, Java does not provide this abstraction. Therefore in this recitation you will implement your own.

   We will not use `null`, any looping constructs in Java, or the `isPresent()`, `isEmpty()` or `get()` methods of the `Optional` class.

   (a) Define a generic `Lazy` class to encapsulate a value and include the following methods

   - `static of(T v)` method that initializes the `Lazy` object with the given value.
   - `static of(Supplier s)` method that takes in a supplier that supplies the value when needed.
   - `get()` method that is called when the value is needed. If the value is already available, return that value; otherwise, compute the value and return it. The computation should only be done once for the same value.
   - `toString()` method: returns "?" if the value is not yet available; returns the string representation of the value otherwise.

(b) Make `Lazy` a functor and a monad by adding the `map` and `flatMap` methods. Remember that `Lazy` should not evaluate anything until `get()` is called, so the function `f` passed into `Lazy` through `map` and `flatMap` should not be evaluated until `get()` is called. Furthermore, they should be evaluated once. That result from `map` and `flatMap`, once evaluated, should be cached (also called memoized), so that function must not be called again.

(c) Adding a method `combine`, which takes in another `Lazy` object and a `BiFunction` to lazily combine the two `Lazy` objects and return a new `Lazy` object.

(d) Consider the class `EagerList` below.

```
import java.util.List;
import java.util.function.UnaryOperator;
import java.util.stream.Collectors;
import java.util.stream.Stream;

class EagerList<T> {
  List<T> list;
  private EagerList(List<T> list) {
    this.list = list;
  }

  static <T> EagerList<T> generate(int n, T seed, UnaryOperator<T> f) {
    return new EagerList<T>(
        Stream.iterate(seed, x -> f.apply(x))
          .limit(n)
          .collect(Collectors.toList())
        );
  }

  public T get(int i) {
    return this.list.get(i);
  }

  public int indexOf(T v) {
    return this.list.indexOf(v);
  }
}
```

Given `n`, the size of the list, `seed`, the initial value, and `f`, an operation, we can generate an `EagerList` up to `n` elements as:
`[seed, f(seed), f(f(seed)), f(f(f(seed))), ... ]`

We can then use the method `get(i)` to find the i-th element in this list, or `indexOf(obj)` to find the `obj` in the list.

But what if `f()` is an expensive computation? When we call `get(k)` where `k << n`, we would have wasted our time computing all the remaining elements in the list. Or if the `obj` that we want to find using `indexOf` is near the beginning of the list, why should we need to compute the remaining elements of the list. Enter `LazyList`.

Rewrite the `EagerList` class as a new class called `LazyList`, making use of the `Lazy` class, so that `get()` and `indexOf()` causes the evaluation of `f()` only as many times as necessary. You may assume that list access is guaranteed to be within bounds.