CS2030S Recitation: Problem Set #7 — Wk 9, Sem 1, 2022/23

1. Which of the following are pure functions?

   (a) Problem #A

```java
int f(int i) {
  if (i < 0) {
    throw new IllegalArgumentException();
  } else {
    return i + 1;
  }
}
```

   (b) Problem #B

```java
int g(int i) {
  System.out.println(i);
  return i + 1;
}
```

   (c) Problem #C

```java
int h(int i) {
  return new Random().nextInt() + i;
}
```

   (d) Problem #A

```java
int k(int i) {
  return Math.abs(i);
}
```

2. Consider the following lambda expression:

   `x -> y -> z -> f(x, y z)`

   where `x`, `y`, and `z` are of some type T and `f` returns a value of type R.

   (a) What kind of lambda expression is this?

   (b) Suppose that:

   - T and R are of type `Integer`
   - `f(x,y,z)` is given by `x + y + z`
   - The above lambda expression implements the `Immutator` functional interface

   Initialize the appropriate lambda expression and assign it to a variable `trisum`. Given three inputs `x`, `y`, and `z`, show how you can evaluate the lambda expression with `x`, `y`, and `z` to obtain `f(x,y,z)`.

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

```java
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`.

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

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

- **Recursive Case:** Represented by a Recursive<T> object, that can be recursed, or
- **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:

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

Then we can evaluate the sum of n terms via the `summer` method below:

```java
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.