CS2030 Programming Methodology
Semester 1 2019/2020

18 October 2019
Problem Set #7 Suggested Answers
Lazy Evaluation

1. Study the following implementation of an infinite list.

```
public interface IFL<T> {
    public static <T> IFL<T> iterate(T seed, Function<T, T> next) {
        return new IFLImpl<T>() {
            T element = seed;
            Function<T, T> func = x -> {
                func = next;
                return element;
            };

            Optional<T> get() {
                element = func.apply(element);
                return Optional.of(element);
            }
        };
    }

    public <R> IFL<R> map(Function<T, R> mapper);
    public void forEach(Consumer<T> action);
}

abstract class IFLImpl<T> implements IFL<T> {
    public <R> IFL<R> map(Function<T, R> mapper) {
        return new IFLImpl<R>() {
            Optional<R> get() {
                return IFLImpl.this.get().map(mapper);
            }
        };
    }

    public void forEach(Consumer<T> action) {
        Optional<T> curr = get();
        while (true) {
            curr.ifPresent(action);
            curr = get();
        }
    }

    abstract Optional<T> get();
}
```

(a) Modify the `iterate` method such that it now supports a condition to stop iterating.

```
IFL<Integer> if = IFL.iterate(0, i -> i < 2, i -> i + 1);
```

```
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.Optional;

public interface IFL<T> {
    public static <T> IFL<T> iterate(T seed, Predicate<T> pred, Function<T, T> next) {
        return new IFLImpl<T>() {
            private T element = seed;
            private boolean empty;

            Function<T, T> func = x -> {
                func = next;
                return element;
            };

            Optional<T> get() {
                element = func.apply(element);
                if (pred.test(element)) {
                    empty = false;
                    return Optional.of(element);
                } else {
                    empty = true;
                    return Optional.empty();
                }
            }

            boolean isEmpty() {
                return this.empty;
            }
        };
    }

    public <R> IFL<R> map(Function<T, R> mapper);
    public void forEach(Consumer<T> action);
}

abstract class IFLImpl<T> implements IFL<T> {
    boolean isEmpty = false;

    public <R> IFL<R> map(Function<T, R> mapper) {
        return new IFLImpl<R>() {
            Optional<R> get() {
                return IFLImpl.this.get().map(mapper);
            }

            boolean isEmpty() {
                return IFLImpl.this.isEmpty();
            }
        };
    }

    public void forEach(Consumer<T> action) {
        Optional<T> curr = get();
        while (!isEmpty()) {
            curr.ifPresent(action);
            curr = get();
        }
    }

    abstract Optional<T> get();
    abstract boolean isEmpty();
}
```

(b) Suppose we call

```
IFL.iterate(0, i -> i < 2, i -> i + 1).map(f).map(g).forEach(c)
```

where `f` and `g` are lambda expressions of type `Function` and `c` is a lambda expression of type `Consumer`. Let `e` be the lambda expression `i -> i + 1` passed to `iterate`. Write down the sequence of which the lambda expressions `e`, `f`, `g`, and `c` that are evaluated. Verify your answer. `f g c e f g c e`

```
IFL.iterate(0, x -> x < 2, x -> {
        System.out.print("e ");
        return x + 1;
        })
    .map(x -> {
        System.out.print("f ");
        return x + 2;
        })
    .map(x -> {
        System.out.print("g ");
        return x * 2;
        })
    .forEach(x -> System.out.print("c "));
```

(c) Define method `concat` takes in two IFL objects, `ifl1` and `ifl2`, and creates a new `IFL` whose elements are all the elements of the first list `ifl1` followed by all the elements of the second list `ifl2`.

```
public static <T> IFL<T> concat(IFL<T> ifl1, IFL<T> ifl2)
```

The elements in newly concatenated list must be lazily evaluated as well. For example, in

```
IFL<Integer> ifl1 = IFL.iterate(0, i -> i < 2, i -> i + 1);
IFL<Integer> ifl2 = IFL.iterate(5, i -> i < 8, i -> i + 2);
IFL<Integer> ifl3 = IFL.concat(ifl1, ifl2);
ifl3.forEach(x -> System.out.print(x + " "));
```

Being a lazy-evaluated, nothing is evaluated when `ifl3` is created. Thus, `concat` should not result in an infinite loop even if the list `ifl1` infinitely long. The elements are only evaluated when terminal operator `forEach` is called. In the example above, `0 1 5 7` will be printed.

```
    public static <T> IFL<T> concat(IFL<T> list1, IFL<T> list2) {
        return new IFLImpl<T>() {
            boolean emptyStream = false;
            Optional<T> get() {
                IFLImpl<T> l1 = (IFLImpl<T>)list1;
                IFLImpl<T> l2 = (IFLImpl<T>)list2;
                if (!l1.isEmptyStream()) {
                    return l1.get();
```

```
                }
                if (!l2.isEmptyStream()) {
                    return l2.get();
                }
                emptyStream = true;
                return Optional.empty();
            }

            boolean isEmptyStream() {
                return emptyStream;
            }
        };

    }
```

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 above `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.

```
public interface Compute<T> {
    public boolean isRecursive();

    public Compute<T> recurse();

    public T evaluate();
}
```

```java
import java.util.function.Supplier;

public class Base<T> implements Compute<T> {
    private Supplier<T> supplier;

    public Base(Supplier<T> supplier) {
        this.supplier = supplier;
    }

    public boolean isRecursive() {
        return false;
    }

    public T evaluate() {
        return supplier.get();
    }

    public Compute<T> recurse() {
        throw new IllegalStateException("Invalid recursive call in base case");
    }

}

import java.util.function.Supplier;

public class Recursive<T> implements Compute<T> {
    private Supplier<Compute<T>> supplier;

    public Recursive(Supplier<Compute<T>> supplier) {
        this.supplier = supplier;
    }

    public boolean isRecursive() {
        return true;
    }

    public Compute<T> recurse() {
        return supplier.get();
    }

    public T evaluate() {
        throw new IllegalStateException("Invalid evaluation in recursive case")
    }
}
```

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

```
import java.util.Scanner;

class Main {
    static long summer(long n) {
        Compute<Long> result = sum(n, 0);
        while (result.isRecursive()) {
            result = result.recurse();
        }
        return result.evaluate();
    }

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

    public static void main(String[] args) {
        System.out.println(factorial(new Scanner(System.in).nextLong()));
    }
}
```

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

```
import java.util.Scanner;

class Main {
    static Compute<Long> fact(long n, long s) {
        if (n == 0) {
            return new Base<>(() -> s);
        } else {
            return new Recursive<>(() -> fact(n - 1, n * s));
        }
    }

    public static void main(String[] args) {
        Compute<Long> result = fact(new Scanner(System.in).nextLong(), 1);
        while (result.isRecursive()) {
            result = result.recurse();
        }
        System.out.println(result.evaluate());
    }
}
```