

Infinite List

We will consider the simplified `InfiniteList<T>` below.

Simplified InfiniteList

```
import cs2030s.fp.Producer;
import cs2030s.fp.Transformer;
import cs2030s.fp.BooleanCondition;

class InfiniteList<T> {
  private final Producer<T> head;
  private final Producer<InfiniteList<T>> tail;

  private InfiniteList(Producer<T> head, Producer<InfiniteList<T>> tail) {
    this.head = head;
    this.tail = tail;
  }

  public static <T> InfiniteList<T> generate(Producer<T> producer) {
    return new InfiniteList<T>(producer, () -> InfiniteList.generate(producer));
  }

  public static <T> InfiniteList<T> iterate(T init, Transformer<T, T> next) {
    return new InfiniteList<T>(() -> init,
        () -> InfiniteList.iterate(next.transform(init), next));
  }

  private T head() {
    return this.head.produce();
  }

  private InfiniteList<T> tail() {
    return this.tail.produce();
  }

  public T get(int n) {
    if (n == 0) {
      return this.head();
    }
    return this.tail().get(n - 1);
  }
}
```

Preliminary

The following questions are not going to be discussed. Answers is given without explanation. You are encouraged to discuss them on Ed.

1. We want to modify `InfiniteList<T>` into `AltList<T, U>` such that the type is now alternating. In other words, the tail segment is `AltList<U, T>`. A template is as follows.

   Alternating Infinite List

   ```
   class AltList<T, U> {
     private final Producer<T> head;
     private final Producer<AltList<U, T>> tail;

     private AltList(Producer<T> head, Producer<AltList<U, T>> tail) {
       this.head = head;
       this.tail = tail;
     }

     public static <T, U> AltList<T, U> generate(Producer<? extends T> head,
         Producer<? extends U> tail) {
       return new AltList<T, U>(/* (A) */, /* (B) */);
     }
   }
   ```

   We want to implement the function `generate` that takes in two `Producer`s. The first producer is used to produce the `head` element and the second producer is used to produce the `head` of the `tail` element.

   (a) What is the return type of `generate`?
   (b) What is the type of the first producer argument? Use the most flexible type.
   (c) What is the type of the second producer argument? Use the most flexible type.
   (d) What is the method descriptor of `generate`? Use the most flexible type.
   (e) Implement `generate`.

2. Consider the class `InfiniteList<T>` above, which implements an infinite list using `Producer<T>` which is our version of `Supplier<T>` [1].

   (a) Implement the method `InfiniteList::fib` that creates an infinite list of Fibonacci numbers. The output should be of type `InfiniteList<BigInteger>`.

   Template

   ```
   /**
    * Returns an InfiniteList of Fibonacci numbers starting with
    *    the given parameters.
    * @param a the first BigInteger in the sequence
    * @param b the second BigInteger in the sequence
    * @return an InfiniteList of Fibonacci numbers
    */
   public static InfiniteList<BigInteger> fib(BigInteger a, BigInteger b) {
     // : your code here
   }
   ```

   Sample Run

   ```
   jshell> InfiniteList<BigInteger> list = fib(BigInteger.ONE, BigInteger.ONE);
   jshell> list.get(0)
   $.. ==> 1
   ```

```
jshell> list.get(2)
$.. ==> 2
jshell> list.get(3)
$.. ==> 3
jshell> list.get(10)
$.. ==> 89
jshell> list.get(20)
$.. ==> 10946
jshell> list.get(50)
$.. ==> 20365011074
jshell> list.get(100)
$.. ==> 573147844013817084101
jshell> list.get(200)
$.. ==> 453973694165307953197296969697410619233826
jshell>
```

(b) Implement the method `InfiniteList::zipWith`, to combine two lists, element-wise, with a curried lambda expression[2].

Template

```
public <S, R> InfiniteList<R> zipWith(InfiniteList<? extends S> list,
    Transformer<? super T, ? extends Transformer<? super S, ? extends R>> map) {
  // : your code here
}
```

Sample Run

```
jshell> InfiniteList<Integer> l1 = InfiniteList.iterate(0, i -> i + 1);
jshell> InfiniteList<String> l2 = InfiniteList.iterate("", s -> s + "S");
jshell> InfiniteList<Object> l3 = l1.zipWith(l2, i -> s -> i + s.length());
jshell> l3.get(0);
$.. ==> 0
jshell> l3.get(1);
$.. ==> 2
jshell> l3.get(2);
$.. ==> 4
```

Discussion

The following questions are problem sets to be discussed during recitation. You are expected to attempt them before coming to the recitation. You may be randomly called to present your answer during recitation. You are encouraged to discuss them on Ed.

Write your answer in Functional Style.

1. Implement the method `fib` that returns the first n Fibonacci numbers. The output should be of type `Stream<BigInteger>`[3].

Template

```
/**
 * Returns a Stream of n Fibonacci numbers starting with 1 and 1.
 * @param n the number of Fibonacci numbers to return
 * @return a Stream of n Fibonacci numbers
 */
Stream<BigInteger> fib(int n) {
```

```
  // : your code here
}
```

Sample Run

```
jshell> Stream<BigInteger> s = fib(10)
jshell> s.forEach(System.out::println)
1
1
2
3
5
8
13
21
34
55
```

You can use the methods `iterate`, `map`, and `limit` from the class `Stream`, as well as the `Pair` class below to help you.

Pair.java

```
class Pair<T> {
  public T first;
  public T second;

  public Pair(T first, T second) {
    this.first = first;
    this.second = second;
  }
}
```

2. Implement the method `product` that takes in two `List` objects `list1` and `list2`, and produce a `Stream` containing elements combining each element from `list1` with evey element from `list2` using a given lambda expression of type `BiFunction`. This is similar to our own version called `cs2030s.fp.Combiner`.

Template

```
<T, U, R> Stream<R> product(List<? extends T> list1,
    List<? extends U> list2,
    BiFunction<? super T, ? super U, ? extends R> func) {
  // : your code here
}
```

Sample Run

```
jshell> List<Integer> list1 = List.of(1, 2, 3, 4)
jshell> List<String> list2 = List.of("A", "B")
jshell> product(list1, list2, (i, s) -> i + s).forEach(System.out::println)
1A
1B
2A
2B
3A
3B
4A
4B
```

You may find the method `stream()` in the interface `List`, which converts a `List` of objects into a `Stream` of objects, useful.

3. The i-th omega number[4] is the number of **distinct** prime factors for the number i. The first 10 omega numbers are 0, 1, 1, 1, 1, 2, 1, 1, 1, 2. Implement the method `omega` that returns the first n omega numbers. The output should be of type `Stream<Integer>`.

   Template

   ```
   /**
    * Returns a Stream consisting of the first n omega numbers.
    * @param n the number of omega numbers to return
    * @return a Stream of the first n omega numbers
    */
   Stream<Long> omega(int n) {
     // : your code here
   }
   ```

   Sample Run

   ```
   jshell> omega(10).forEach(System.out::println)
   0
   1
   1
   1
   1
   2
   1
   1
   1
   2
   ```

   You can use the methods `iterate`, `map`, `filter`, `noneMatch`, and `count` from the class `Stream` to help you. Write additional methods if necessary.

Exercise

The following questions are to be discussed on Ed as no answer will be given. ***Use only concepts that you have learnt****. You are expected to attempt them before the next lecture as the concepts may be used and/or mentioned in the lecture. The concept may also be tested for assessment.*

1. Given two functions F and G, let us define the composition $F \circ G$ as the function H such that $H(x) \equiv G(F(x))$. Let us implement this in several ways. For simplicity, you are given the following interface and class.

   Func

   ```
   @FunctionalInterface
   interface Func<T, R> {
     R call(T t);
   }
   ```

   Fn

   ```
   public abstract class Fn<T, R> implements Func<T, R> {
     // : your code here
   }
   ```

(a) Implement an instance method `Fn::then(Fn g)` to compose this with `g` (i.e., `this` $\circ g$). Use the following method descriptor:

```
public <U> Fn<T, U> then(Fn<R, U> g)
```

(b) Implement a class method `Fn::compose(Fn f, Fn g)` to compose `f` with `g` (i.e., $f \circ g$). Use the following method descriptor:

```
public static <T, R, U> Fn<T, U> then(Fn<T, R> f, Fn<R, U> g)
```

Bibliography

[1] Oracle, "Java 25 API: Supplier." [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/function/Supplier.html

[2] H. B. Curry, "Some Philosophical Aspects of Combinatory Logic," *The Kleene Symposium*, vol. 101. in Studies in Logic and the Foundations of Mathematics, vol. 101. Elsevier, pp. 85–101, 1980.

[3] Oracle, "Java 25 API: Stream." [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/stream/Stream.html

[4] G. H. Hardy and E. M. Wright, "An Introduction To The Theory Of Numbers," Oxford University Press, 2008.