Lab 7: Memo List

- Deadline: 25 October, 2022, Tuesday, 23:59, SST
- Mark: 4%

Prerequisite

- Caught up to Unit 32 of Lecture Notes

Important Concepts Tested

- ***Memo*:** Compute only when needed & do not repeat yourself
- ***PECS*:** Make your method signature as flexible as possible
- ***JavaDoc*:** Documenting your code and generating the documentation

Files

You are given the following implementation for your Lab 7:

- `cs2030s.fp.Action`
- `cs2030s.fp.Immutator`
- `cs2030s.fp.Constant`
- `cs2030s.fp.Combiner`
- `cs2030s.fp.Actionable`
- `cs2030s.fp.Immutatorable`
- `cs2030s.fp.Actually`
- `cs2030s.fp.Memo`

The files `Test1.java`, `Test2.java`, etc., as well as `CS2030STest.java`, are provided for testing. You can edit them to add your test cases, but they will not be submitted.

You are given a badly implemented `MemoList` *(which is just `EagerList` copied into a new class*). Further note that the `EagerList` is different from our lecture note `EagerList`. This current `EagerList` is simply a wrapper for Java `List`. The `EagerList` and `InfiniteList` in lecture note will form the basis for Lab 8.

Furthermore, as a checkpoint in case you could not complete `Actually` and `Memo` on time or you are still facing many errors in its implementation, we provide an implementation of `Actually` and `Memo`. These implementations will not satisfy many of the requirements for Lab 5 and 6.

You may familiarise yourself with this implementation of you may choose to use your own implementation. If you choose to use your own implementation of `Memo`, please make sure that you include the implementation of the `Lazy` class inside the file `Memo.java`. Otherwise, your submission will not work on CodeCrunch and you will get 0 mark for this lab.

Additionally, from here onwards, you will get 0 mark for your lab if you are using the `unwrap` method inside `Actually`. Our implementation of `Actually` no longer has `unwrap`.

IMPORTANT❗

1. If you are using your own implementation of `Memo`, you **MUST** include the implementation of the `Lazy` class inside the file `Memo.java`. In other words, your `Memo.java` must include two classes: `Lazy` and `Memo`.
2. You are **NOT** allowed to use `unwrap` method from `Actually`.

Preliminary

`Actually`

The class `Actually<T>` is simplified by not using inner/nested class. This is achieved by simply using a convention:

- If `err` is not `null`, then it is a *Failure* and we should not use the value of `val`.
- If `err` is `null`, then it is a *Success* and we can use the value of `val` *including when `val` is `null`*.

We no longer use a monospace font `Failure` and `Success` as they are no longer a class.

Another simplification is that we just have added a notion of a "common error" that can be created using the static method `err()` that takes in no argument. This common error can be used to indicate the most general kind of error you can think of. We also modified the `equals` method such that all *Failure* (*i.e., the case when `err` is not `null`*) are considered to be the equal to each other.

Furthermore, note that we added a method `check` that takes in an `Immutator<Boolean, ? super T>`. This can be used to convert a *Success* to a *Failure* when the immutator predicate is not satisfied. Otherwise, this method returns itself.

Lastly, we simplify the `toString()` method to simply print `"<>"` when we are having a *Failure*. This is just a simplification as we no longer need to know the error inside.

Please familiarise yourself with this implementation. The behaviour should be similar to what you expect. You may also replace `Actually` with your implementation.

`Memo`

The class `Memo` is simplified by not inheriting from `Lazy`. In fact, we remove `Lazy` completely and merge the field `Constant<? extends T>` into `Memo`. We will be using the following convention:

- If `com` is `null`, then we have evaluated the value and we can use the value of `val` *including when `val` actually stores `null`*.
- If `com` is not `null`, then we have not evaluated the value and we cannot use the value of `val` unless we evaluate it first using the private method `eval`.

The private method `eval` is used to force an evaluation of `com` into the value `val`. This will then set the value of `com` to `null` to prevent another evaluation as we have now memoized the value produced into `val`.

Please familiarise yourself with this implementation. The behaviour should be similar to what you expect. You may also replace `Memo` with your implementation. If you do so, please include the implementation of `Lazy` within your `Memo.java`.

Memo List

The `Memo` class can be used to build a lazy-evaluated-and-memoized list.

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

We can then use the method `get(i)` to find the i-th element in this list, or `indexOf(obj)` to find the index of `obj` in the list. (*Hint: for `indexOf` to work properly, you need to provide `equals` method in `Memo` where two `Memo` are equal if the elements are of the same type and equal based on their respective `equals` method.*)

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

  public static <T> EagerList<T> generate(int n, T seed, Immutator<T, T>
f) {
    EagerList<T> eagerList = new EagerList<>(new ArrayList<>());
    T curr = seed;
    for (int i = 0; i < n; i++ ) {
      eagerList.list.add(curr);
      curr = f.invoke(curr);
    }
    return eagerList;
  }

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

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

  @Override
  public String toString() {
    return this.list.toString();
  }
}
```

But suppose `f()` is an expensive computation, and we ended up just needing to `get(k)` where `k` is much smaller than `N`, then, we would have wasted our time computing all the remaining elements in the list! Similarly, if the `obj` that we want to find using `indexOf` is near the beginning of the list, there is no need to compute the remaining elements of the list.

We want to change this `EagerList` into a `MemoList` that make use of the `Memo` class such that `get()` and `indexOf()` causes evaluation of `f()` only as many times as necessary. The first step is already done for you, we have made a copy of `EagerList` into `MemoList` but it has not used `Memo` yet. Change this implementation to use `Memo`. (*Hint: you only need to make minimal changes. Neither a new field nor a new loop is necessary.*)

Important: You need to change the method signature of `generate` to be as flexible as possible following PECS.

```
jshell> /open MemoList.java
jshell> Immutator<Integer, Integer> incr = x -> {
   ...>   System.out.println(x + " + 1");
   ...>   return x + 1;
   ...> }
```

```
jshell> MemoList<Integer> l = MemoList.generate(1000000, 0, incr)
l ==> [0, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> l.indexOf(4)
0 + 1
1 + 1
2 + 1
3 + 1
$.. ==> 4
jshell> l
l ==> [0, 1, 2, 3, 4, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...
jshell> l.get(8)
4 + 1
5 + 1
6 + 1
7 + 1
$.. ==> 8
jshell> l
l ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...
jshell> l.get(2)
$.. ==> 2
jshell> l
l ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...
jshell> l.indexOf(6);
$.. ==> 6
jshell> l
l ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...
```

You can test your code by running the `Test1.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac cs2030s/fp/*java
$ javac -Xlint:rawtypes Test1.java
$ java Test1
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
MemoList.java
$ javadoc -quiet -private -d docs MemoList.java
```

FiboLazy

The way we generate the `MemoList` using an `Immutator` does not allow us to easily generate Fibonacci sequence:

> `[0, 1, 1, 2, 3, 5, 8, ...]`

where each element is the sum of the previous two elements. To accommodate this, we need an overloaded `generate` method as follows:

- takes in 4 parameters:
  - `int n`: the number of elements in the list

  - `T fst`: the first element in the list
  - `T snd`: the second element in the list
  - `Combiner<_, _, _> f`: a combiner
    - Make sure the type is as flexible as possible
- returns a new `MemoList<T>` generated as:
  - let `fst = x` and `snd = y`
  - the result is `[x, y, f(x, y), f(y, f(x, y)), f(f(x, y), f(y, f(x, y))), ...]`

Add the overloaded `generate` method to the `MemoList` class.

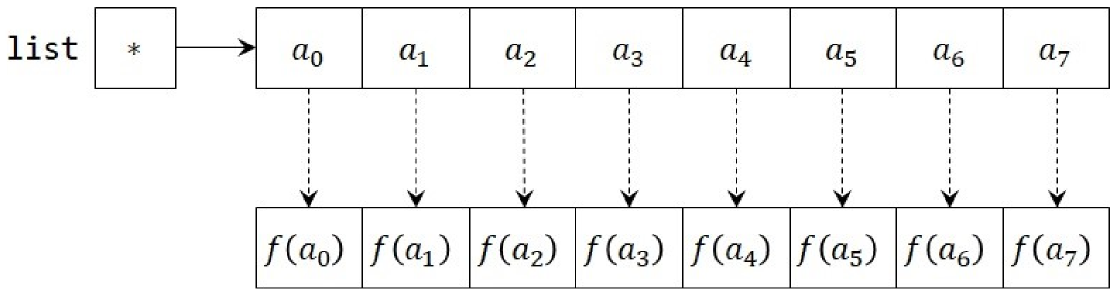
```
jshell> /open MemoList.java
jshell> Combiner<Integer, Integer, Integer> fib = (x, y) -> {
   ...>   System.out.println(x + " + " + y);
   ...>   return x + y;
   ...> }
jshell> MemoList<Integer> fibL = MemoList.generate(1000000, 0, 1, fib)
fibL ==> [0, 1, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> fibL.indexOf(8)
0 + 1
1 + 1
1 + 2
2 + 3
3 + 5
$.. ==> 6
jshell> fibL
fibL ==> [0, 1, 1, 2, 3, 5, 8, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> fibL.get(8)
5 + 8
8 + 13
$.. ==> 21
jshell> fibL
fibL ==> [0, 1, 1, 2, 3, 5, 8, 13, 21, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> MemoList<Integer> l = MemoList.generate(1000000, 0, 1, (x, y) ->
y + 1)
l ==> [0, 1, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...
jshell> l.indexOf(4)
$.. ==> 4
jshell> l
l ==> [0, 1, 2, 3, 4, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...
jshell> l.get(8)
$.. ==> 8
jshell> l
l ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...
```

You can test your code by running the `Test2.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac cs2030s/fp/*java
$ javac -Xlint:rawtypes Test2.java
$ java Test2
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
MemoList.java
$ javadoc -quiet -private -d docs cs2030s/fp/MemoList.java
```

`map`

Now let's add a `map` method. The `map` method (*lazily*) applies the given `Immutator` on each element in the list and returns the resulting `MemoList`. If the element is not yet evaluated, the `map` method should not evaluate the element. This can be visualised as follows where `f` is the `Immutator`:



This is, in fact, something similar to `transform` from `Immutatorable` but operated on each element of the list. The common name for this is `map`.

Note: The `Immutator` for `map` takes in an element of type `T` and returns an element of type `R`. This may need to be changed to make the method signature as flexible as possible following PECS.

```
jshell> /open MemoList.java
jshell> import cs2030s.fp.Immutator
jshell> Immutator<Integer, Integer> incr = x -> {
   ...>   System.out.println(x + " + 1");
   ...>   return x + 1;
   ...> }
jshell> Immutator<Integer, Integer> dbl = x -> {
   ...>   System.out.println(x + " + " + x);
   ...>   return x + x;
   ...> }

jshell> MemoList<Integer> nat = MemoList.generate(1000000, 0, incr)
nat ==> [0, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> MemoList<Integer> even = nat.map(dbl)
even ==> [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> nat.indexOf(3)
0 + 1
1 + 1
2 + 1
```

```
$.. ==> 3
jshell> nat
nat ==> [0, 1, 2, 3, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> even
even ==> [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> // Note the 3 + 1 and 4 + 1 comes from
jshell> // evaluation of nat
jshell> even.indexOf(10)
0 + 0
1 + 1
2 + 2
3 + 3
3 + 1
4 + 4
4 + 1
5 + 5
$.. ==> 5
jshell> nat
nat ==> [0, 1, 2, 3, 4, 5, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> even
even ==> [0, 2, 4, 6, 8, 10, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> MemoList<Integer> odd = even.map(incr)
odd ==> [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,  ...
jshell> // Note only forces evaluation of
jshell> // nat and even that are needed
jshell> odd.get(10)
5 + 1
6 + 1
7 + 1
8 + 1
9 + 1
10 + 10
20 + 1
$.. ==> 21
jshell> nat
nat ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ?, ?, ?, ?, ?,  ...
jshell> even
even ==> [0, 2, 4, 6, 8, 10, ?, ?, ?, ?, 20, ?, ?, ?, ?, ?,  ...
jshell> odd
odd ==> [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 21, ?, ?, ?, ?, ?,  ...
```

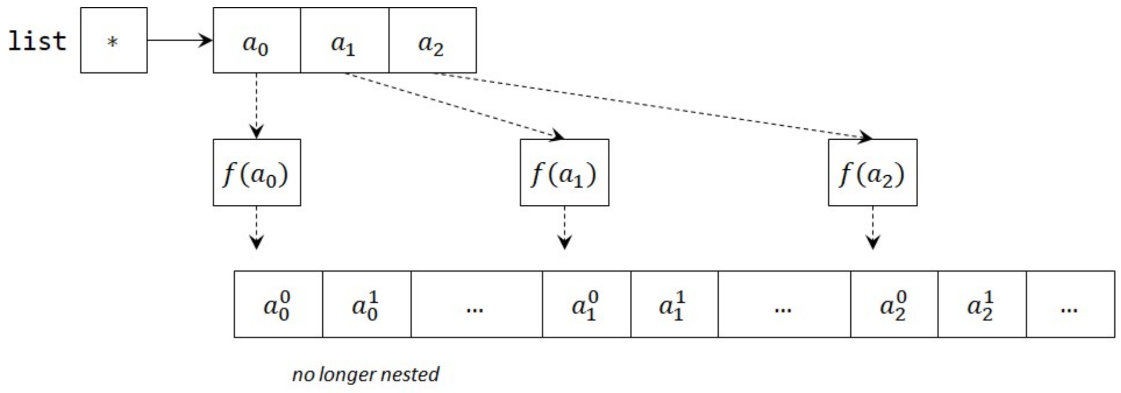
You can test your code by running the `Test3.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac cs2030s/fp/*java
$ javac -Xlint:rawtypes Test3.java
$ java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
MemoList.java
$ javadoc -quiet -private -d docs cs2030s/fp/MemoList.java
```

`flatMap`

Now let's add a `flatMap` method. While the `Immutator` in the `map` method returns an element of type `R`, the `Immutator` in the `flatMap` method returns an element of type `MemoList<R>`. So first, recap that `map` applies the `Immutator` on each element. But if the immutator is returning a `MemoList`, `map` will create a nested list. On the other hand, `flapMap` will *flatten* this nested list.

Recap the image below from recitation illustrating the behaviour of `flatMap`.



You still need to ensure that the method signature for `flatMap` is as flexible as possible. However, to simplify this, you are guaranteed that the `MemoList` generated by the `Immutator` passed to `flatMap` will always be of type `MemoList<R>`. In other words, you do not need to take care of cases where we may want to return `MemoList<? extends R>`.

```
jshell> /open MemoList.java
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.Memo
jshell> Immutator<MemoList<Integer>, Integer> dupl = x -> {
   ...>   System.out.println("Duplicating " + x + " for " + x +
"-times");
   ...>   return MemoList.generate(x, x, n -> x);
   ...> }
jshell> MemoList<Integer> nat = MemoList.generate(5, 1, x -> x + 1);
nat ==> [1, ?, ?, ?, ?]
jshell> MemoList<Integer> superNat = nat.flatMap(dupl)
Duplicating 1 for 1-times
Duplicating 2 for 2-times
Duplicating 3 for 3-times
Duplicating 4 for 4-times
Duplicating 5 for 5-times
superNat ==> [1, 2, ?, 3, ?, ?, 4, ?, ?, ?, 5, ?, ?, ?, ?]

jshell> MemoList<Integer> superEven = superNat.map(x -> {
   ...>   System.out.println("   2*" + x + " = " + (2 * x));
   ...>   return x * 2;
   ...> })
superEven ==> [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]
jshell> superEven.get(12)
   2*5 = 10
$.. ==> 10
```

```
jshell> superEven
superEven ==> [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 10, ?, ?]
jshell> superNat
superNat ==> [1, 2, ?, 3, ?, ?, 4, ?, ?, ?, 5, 5, 5, ?, ?]
jshell> nat
nat ==> [1, 2, 3, 4, 5]

jshell> // we will show the difference with map
jshell> MemoList<Integer> nat2 = MemoList.generate(5, 1, x -> x + 1)
nat2 ==> [1, ?, ?, ?, ?]
jshell> MemoList<MemoList<Integer>> nestNat2 = nat2.map(dupl)
nestNat2 ==> [?, ?, ?, ?, ?]
jshell> nestNat2.get(2)
Duplicating 3 for 3-times
$.. ==> [3, ?, ?]
jshell> nestNat2
nestNat2 ==> [?, ?, [3, ?, ?], ?, ?]

jshell> for (int i=0; i<5; i++) {
   ...>   nestNat2.get(i);
   ...> }
Duplicating 1 for 1-times
Duplicating 2 for 2-times
Duplicating 4 for 4-times
Duplicating 5 for 5-times
jshell> nestNat2
nestNat2 ==> [[1], [2, ?], [3, ?, ?], [4, ?, ?, ?], [5, ?, ?, ?, ?]]

jshell> for (int i=0; i<5; i++) {
   ...>   for (int j=0; j<=i; j++) {
   ...>     nestNat2.get(i).get(j);
   ...>   }
   ...> }
jshell> nestNat2
nestNat2 ==> [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4], [5, 5, 5, 5, 5]]
```

You can test your code by running the `Test4.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac cs2030s/fp/*java
$ javac -Xlint:rawtypes Test4.java
$ java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
MemoList.java
$ javadoc -quiet -private -d docs cs2030s/fp/MemoList.java
```

PECS

If you have not done so, you can do a simple test on PECS by running the `Test5.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style but there is no need to generate javadoc.

```
$ javac cs2030s/fp/*java
$ javac -Xlint:rawtypes Test5.java
$ java Test5
```

Following CS2030S Style Guide

You should make sure that your code follows the given Java style guide.

Grading

This lab is worth 16 marks and contributes 4% to your final grade. The marking scheme is as follows:

- Documentation: 2 marks
- Everything Else: 14 marks

We will deduct 1 mark for each unnecessary use of `@SuppressWarnings` and each raw type. `@SuppressWarnings` should be used appropriately and not abused to remove compilation warnings.

Note that general style marks are no longer awarded will only be awarded for documentation. You should know how to follow the prescribed Java style by now. We will still deduct up to 2 marks if there are serious violations of styles. In other words, if you have no documentation and serious violation of styles, you will get deducted 4 marks.

Submission

Similar to Lab 6, submit the files inside the directory cs2030s/fp along with the other file without the need for folder. Your cs2030s/fp should only contain the following files:

- `Action.java`
- `Actionable.java`
- `Actually.java`
- `Combiner.java`
- `Constant.java`
- `Immutator.java`
- `Immutatorable.java`
- `Memo.java`

Additionally, you ***must*** submit the file `Lab7.h` and `Lab7.java`. Otherwise, you CodeCrunch submission will not run.