Lab 6: Lazy

- Deadline: 18 October, 2022, Tuesday, 23:59, SST
- Mark: 3%

Prerequisite

- Caught up to Unit 32 of Lecture Notes
- Completed Lab 5

Files

The following functional interfaces are already provided:

- `cs2030s.fp.Action`
- `cs2030s.fp.Immutator`
- `cs2030s.fp.Constant`
- `cs2030s.fp.Combiner`
  - This is a new functional interface to combine two values into one.
  - If `Immutator` takes in only one value, `Combiner` takes in two values.

Additionally, the following interfaces are already provided as well:

- `cs2030s.fp.Actionable`
- `cs2030s.fp.Immutatorable`

Copy your implementation of `Actually` over before you get started with Lab 6. A skeleton for `Lazy<T>` and `Memo<T>` are provided for your. Additionally, you are given the simple and non-lazy implementation of Boolean expression:

- `Cond.java` : an interface abstracting a boolean condition that can be evaluated
- `Bool.java` : a boolean value
- `And.java` : a conjunction
- `Or.java` : a disjunction

- `Not.java` : a negation

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.

Being Lazy and Smarter

Programming languages such as Scala support lazy values, where the expression that produces a lazy value is not evaluated until the value is needed. Lazy value is useful for cases where producing the value is expensive, but the value might not eventually be used. Java, however, does not provide a similar abstraction. So, you are going to build one.

This task is divided into several stages. You are highly encouraged to read through all the stages to see how the different levels are related.

You are required to design a `Lazy` and `Memo` classes as part of the `cs2030s.fp` package with one field. *You are not allowed to add additional fields* to `Lazy` .

```
public class Lazy<T> /* implements Immutatorable<T> (for later) */ {
  private Constant<? extends T> init;
   :
}
```

```
public class Memo<T> extends Lazy<T> {
  private Actually<T> value;
   :
}
```

The Basics of Being Lazy

The idea of being lazy is that we do not compute unless we really *really* ***really*** need to. When do we need to compute the value? Simple, that is when we try to get the value.

Define a generic `Lazy<T>` class to encapsulate a value with the following operations such that for each `...` derive the most flexible type:

- protected constructor `protected Lazy(Constant<...> c)` that takes in a constant that produces the value when needed.
- static `from(T v)` method that instantiate the `Lazy` object with the given value using the protected constructor above.
- static `from(Constant<...> c)` method that takes in a constant that produces the value when needed and instantiate the `Lazy` object.

- `get()` method that is called when the value is needed. Compute the value and return.
- `toString()`: returns the string representation of the value.

```
jshell> import cs2030s.fp.Constant
jshell> import cs2030s.fp.Lazy

jshell> Lazy<Integer> mod1 = Lazy.from(2030)
jshell> mod1.get()
$.. ==> 2030

jshell> Lazy<String> mod2 = Lazy.from(() -> "CS2030S")
jshell> mod2.get()
$.. ==> "CS2030S"

jshell> Lazy<String> hello = Lazy.from(() -> {
   ...>   System.out.println("world!");
   ...>   return "hello";
   ...> })
jshell> hello.get()
world!
$.. ==> "hello"
jshell> hello.get() // note "world!" is printed again
world!
$.. ==> "hello"
```

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
Lazy.java
$ javadoc -quiet -private -d docs cs2030s/fp/Lazy.java
```

Smarter Lazy

The smarter idea of being lazy is that if we have computed the value before (*i.e., called `get()`*) then we do not compute the value again. This is called memoization (*the word comes from memo and not a typo from memorization*). Since we already have an `Actually<T>`, when we try perform a computation and the current value might still be uninitialised we treat this like a failure. In order to prevent this failure, we first produce the value using a `Constant` to initialise the value. Now, this is no longer a failure but a success!

Take note of the following constraints:

- Avoid using the `Actually::unwrap` method and avoid access the classes `Actually.Success<T>` or `Actually.Failure` directly.
- Since `Actually` has internalized `try-catch` (*which kind of mimics* `if-else`) checks for whether the value is there or not, you must not use any form of conditional statements and/or try-catch to compare if `value` has been initialised or not.
- You are not allowed to use any raw types.
- You don't need any `@SuppressWarnings` for this lab, but if you do, it must be used responsibly.

Define a generic `Memo<T>` class to encapsulate a value with the following operations such that for each `...` derive the most flexible type:

- you should not have `public` constructor.
- static `from(T v)` method that initializes the `Memo` object with the given value. In this case, the `Memo` is already initialised (*i.e., already computed*).
- static `from(Constant<...> c)` method that takes in a constant that produces the value when needed. In this case, the `Memo` is uninitialised.
- `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()`: returns `"?"` if the value is not yet available; returns the string representation of the value otherwise.

Note that for our class to be immutable and to make the memoization of the value transparent, `toString` should call `get()` and should never return `"?"`. We break the rules of immutability and encapsulation here, just so that it is easier to debug and test the laziness of your implementation.

Hint: You may find the method `valueOf` from the class `String` useful.

```
jshell> import cs2030s.fp.Constant
jshell> import cs2030s.fp.Memo

jshell> Memo<Integer> mod1 = Memo.from(2030)
jshell> mod1
mod1 ==> 2030
jshell> mod1.get()
$.. ==> 2030

jshell> Memo<String> mod2 = Memo.from(() -> "CS2030S")
jshell> mod2
mod2 ==> ?
jshell> mod2.get()
```

```
$.. ==> "CS2030S"

jshell> Memo<String> hello = Memo.from(() -> {
   ...>   System.out.println("world!");
   ...>   return "hello";
   ...> })
jshell> hello
hello ==> "hello"  ? instead of "hello"
jshell> hello.get()
world!
$.. ==> "hello"
jshell> hello.get() // note "world!" is NOT printed again
$.. ==> "hello"
```

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
Memo.java
$ javadoc -quiet -private -d docs cs2030s/fp/Memo.java
```

Implementing Immutatorable and Adding Next

Now let's implement `Immutatorable` interface to our `Lazy` and `Memo` as well as adding `next` method. To do this, add the `transform` and `next` method to both `Lazy` and `Memo`. First, for `Lazy`.

Add the `transform` method to `Lazy`. Additionally, add the `next` method (*we did not have an interface for this for simplicity*). Remember that `Lazy` should not evaluate anything until `get()` is called, so the function `f` passed into `Lazy` through `transform` and `next` should not be evaluated until `get()` is called. The result should not be cached (*i.e., memoized*).

Next, add the `transform` and `next` method to `Memo`. These two methods should override the methods from `Lazy`. This should limit the type of the input parameter. But the return type here should be as specific as method overriding allows. Remember that `Memo` should cache the result. In other words, they should only be evaluated once, so that function must not be called again.

```
jshell> import cs2030s.fp.Constant
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.Lazy
jshell> import cs2030s.fp.Memo

jshell> Constant<String> password = () -> "123456"

jshell> Lazy<String> lazy = Lazy.from(password)
jshell> lazy
lazy ==> 123456
jshell> lazy.transform(str -> str.substring(0, 1))
$.. ==> 1

jshell> Memo<String> memo = Memo.from(password)
jshell> memo.transform(str -> str.substring(0, 1))
$.. ==> ?
jshell> memo
memo ==> ?
jshell> memo.transform(str -> str.substring(0, 1)).get()
$.. ==> "1"
jshell> memo
memo ==> 123456
jshell> memo.get()
$.. ==> "123456"

jshell> Immutator<Integer, String> len = str -> {
   ...>   System.out.println("length");
   ...>   return str.length();
   ...> }

jshell> Lazy<Integer> lazyLen = lazy.transform(len)
jshell> lazyLen
length
lazyLen ==> 6
jshell> lazyLen.get()
length
$.. ==> 6
jshell> lazyLen.get()
length
$.. ==> 6

jshell> Memo<Integer> memoLen = memo.transform(len)
jshell> memoLen
memoLen ==> ?
jshell> memoLen.get()
length
$.. ==> 6
jshell> memoLen.get()
$.. ==> 6

jshell> Memo<Integer> step1 = Memo.from(1010)
step1 ==> 1010

jshell> Memo<Integer> step2 = step1.transform(i -> i * 2)
step2 ==> ?
jshell> Memo<Integer> step3 = step2.next(i -> Memo.from(i + 10))
step3 ==> ?
jshell> step3.get()
$.. ==> 2030
jshell> step2 // to get() step3 need to get() step2
step2 ==> 2020
jshell> step1 // to get() step2 need to get() step1
step1 ==> 1010
```

```

jshell> Memo<Integer> noErr = Memo.from(0)
noErr ==> 0
jshell> Memo<Integer> err = noErr.transform(x -> 1/x)
err ==> ?
jshell> // if you run err.get(), you will get an exception
```

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
Lazy.java
$ javadoc -quiet -private -d docs cs2030s/fp/Lazy.java
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
Memo.java
$ javadoc -quiet -private -d docs cs2030s/fp/Memo.java
```

Combine

From here, we are more interested in `Memo` because although `Lazy` is useful, it may perform unnecessary duplicate computation.

We have provided an interface called `Combiner<R, S, T>` in `cs2030s.fp`, with a single `combine` method to combine two values, of type `S` and `T` respectively, into a result of type `R`.

Add a method called `combine` into `Memo`. The `combine` method takes in another `Memo` object and a `Combiner` implementation to lazily combine the two `Memo` objects (which may contain values of different types) and return a new `Memo` object.

```
jshell> import cs2030s.fp.Combiner
jshell> import cs2030s.fp.Memo

jshell> Memo<Integer> twenty, thirty, modInt
twenty ==> null
thirty ==> null
modInt ==> null

jshell> twenty = Memo.from(() -> 20)
twenty ==> ?
jshell> thirty = Memo.from(() -> 30)
thirty ==> ?

jshell> Combiner<String, Integer, Integer> concat = (x, y) -> {
   ...>   System.out.println("combine");
   ...>   return x.toString() + y.toString();
   ...> }

jshell> modInt = twenty.combine(thirty, (x, y) -> x * 100 + y)
modInt ==> ?
jshell> Memo<String> modStr = twenty.combine(thirty, concat)
modStr ==> ?

jshell> modStr.get()
combine
$.. ==> "2030"
jshell> twenty
twenty ==> 20
jshell> thirty
thirty ==> 30

jshell> modInt
modInt ==> ?

jshell> Combiner<String, Integer, Double> comb = (x, y) -> x.toString() +
" + " + y.toString()
jshell> Memo<String> s = modInt.combine(Memo.from(0.1), comb)
s ==> ?
jshell> s.get()
$.. ==> "2030 + 0.1"
jshell> modInt
modInt ==> 2030

jshell> Memo<Integer> x = Memo.from(1)
jshell> for (int i = 0; i < 10; i ++) {
   ...>   final Memo<Integer> y = x; // final just to ensure it is
unchanged
   ...>   final int j = i;
   ...>   x = Memo.from(() -> { System.out.println(j); return y.get() +
y.get(); });
   ...> }
jshell> x.get();
9
8
7
6
5
4
3
2
1
0
$.. ==> 1024
```

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
Lazy.java
$ javadoc -quiet -private -d docs cs2030s/fp/Lazy.java
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
Memo.java
$ javadoc -quiet -private -d docs cs2030s/fp/Memo.java
```

Boolean Algebra

The `Memo` class can be used to build a function with short-circuit operation.

Consider a boolean expression with 4 classes abstracted into an interface `Cond.java`:

- `Bool.java`: stores a `boolean` value which may be `true` or `false`.
- `And.java`: conjunction of two boolean expressions, created from `And(Cond lVal, Cond rVal)` to mean `lVal && rVal`.
- `Or.java`: disjunction of two boolean expressions, created from `Or(Cond lVal, Cond rVal)` to mean `lVal || rVal`.
- `Not.java`: negation of a boolean expression, created from `Not(Cond val)` to mean `!val`.

Creating an instance of boolean expression above requires us to have fully evaluated the arguments. As such, when we call the `eval` method, all the values have actually been evaluated. But remember that the operation `&&` and `||` are actually short-circuit operation. In particular, `false && X` and `true || X` should not evaluate the value `X` since the result will already be `false` and `true` respectively.

But suppose `X` takes very long to evaluate. Short-circuit operation will simply return the result without evaluating the value of `X` and that saves us time. This is exactly what we want to do. We want to avoid evaluating this `X` whenever possible. Study the code below to understand how the classes work. You do not really have to know how negation method `neg` works. But if you are interested in it, you can look up De Morgan's Law.

```
jshell> /open Cond.java
jshell> /open Bool.java
jshell> /open And.java
jshell> /open Or.java
jshell> /open Not.java

jshell> Constant<Boolean> t = new Constant<>() {
   ...>   public Boolean init() {
   ...>     return true;
   ...>   }
   ...> }
jshell> Constant<Boolean> f = new Constant<>() {
   ...>   public Boolean init() {
```

```
   ...>     String res = "";
   ...>     for (int i=0; i<100000; i++) {
   ...>       res += i;
   ...>     }
   ...>     return false;
   ...>   }
   ...> }
jshell> // the following line will take some time to run
jshell> Cond cond = new And(new Or(new Bool(t), new Bool(f)), new Not(new
Not(new Bool(t))))
cond ==> ((t | f) & !(!(t)))
jshell> cond.neg()
$.. ==> ((!(t) & !(f)) | !(t))
jshell> cond.neg().neg()
$.. ==> ((t | f) & t)
jshell> cond.eval()
$.. ==> true
jshell> cond.neg().eval()
$.. ==> false
jshell> cond.neg().neg().eval()
$.. ==> true
```

Change the boolean expression implementation above such that a value is only evaluated when it is truly needed. You should use `Memo` class in your changes.

Hint: you only need to make minimal changes. Neither a new field nor a new function is necessary. If done correctly, the following sample run below should run very quickly.

```
jshell> /open Cond.java
jshell> /open Bool.java
jshell> /open And.java
jshell> /open Or.java
jshell> /open Not.java

jshell> Constant<Boolean> t = new Constant<>() {
   ...>   public Boolean init() {
   ...>     return true;
   ...>   }
   ...> }
jshell> Constant<Boolean> f = new Constant<>() {
   ...>   public Boolean init() {
   ...>     String res = "";
   ...>     for (int i=0; i<100000; i++) {
   ...>       res += i;
   ...>     }
   ...>     return false;
   ...>   }
   ...> }
jshell> // the following line will run very quickly
jshell> Cond cond = new And(new Or(new Bool(t), new Bool(f)), new Not(new
Not(new Bool(t))))
cond ==> ((? | ?) & !(!(?)))
jshell> cond.neg()
$.. ==> ((!(?) & !(?)) | !(?))
jshell> cond.neg().neg()
```

```
$.. ==> ((? | ?) & ?)
jshell> cond.eval()
$.. ==> true
jshell> cond.neg()
$.. ==> ((!(t) & !(?)) | !(t))
jshell> cond.neg().neg()
$.. ==> ((t | ?) & t)
```

You can test your code 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 12 marks and contributes 3% to your final grade. The marking scheme is as follows:

- Documentation: 2 marks
- Everything Else: 10 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 5, 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`
- `Lazy.java`
- `Memo.java`

Additionally, you ***must*** submit the file `Lab6.h`. Otherwise, you CodeCrunch submission will not run.