NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL ASSESSMENT 1 FOR
Semester 2 AY2023/2024

CS2030S Programming Methodology II

March 2021 Time Allowed 120 minutes

INSTRUCTIONS TO CANDIDATES

1. The total mark is 30.
2. This is an OPEN-BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed. except those accessible from the PE nodes (`peXXX.comp.nus.edu.sg`) (e.g., `man` pages are allowed).
3. Solve the programming tasks by creating any necessary files and editing them. You can leave the files in your home directory and log off after the assessment is over. There is no need to submit your code.
4. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
5. Write your student number on top of EVERY FILE you created or edited as part of the `@author` tag. Do not write your name.
6. To compile your code, run `javac -Xlint:unchecked -Xlint:rawtypes *.java`. You can also compile the files individually if you wish.

IMPORTANT: Any code you wrote that cannot be compiled will result in 0 marks will be given for the question or questions that depend on it. Make sure your program compiles by running

```
javac -Xlint:unchecked -Xlint:rawtypes *.java
```

before submission.

1. (15 points) **Expression**

Marking Criteria

- Correctness (3 marks)
- Design (10 marks)
- Style (2 marks)

An expression is an entity that can be evaluated into a value. We consider two types of expression in this question:

- An operand, which itself is a value.
- A binary operation, which is a mathematical function that takes in two expressions and produces an output value.

For instance,

- `3` is an expression that evaluates to 3.
- `3 + 2` is an expression that evaluates to 5.
- `(3 + 2) + 3` is also an expression that evaluates to 8.

An operand is not necessarily an integer. It can be of any type. An expression can be evaluated to any type.

Three skeleton files are provided for you: `Operand.java`, `Operation.java`, and `InvalidOperandException.java`. If you need extra classes or interfaces, create the necessary additional Java files yourself.

(a) **Operand**

Create a class called `Operand` that encapsulates the operands of an operation. The `Operand` class can contain references to a value of any reference type.

You may create additional parent classes or interfaces if you think it is appropriate.

The `Operand` has an `eval` method that returns its value.

```
jshell> new Operand(5).eval()
$.. ==> 5
jshell> new Operand("string").eval()
$.. ==> "string"
jshell> new Operand(true).eval()
$.. ==> true
```

(b) **InvalidOperandException**

Create an unchecked exception named `InvalidOperandException` that behaves as follows:

```
jshell> InvalidOperandException e = new InvalidOperandException('!')
jshell> e.getMessage();
$.. ==> "ERROR: Invalid operand for operator !"
```

The constructor for `InvalidOperandException` takes in a char which is the corresponding symbol for the operation that is invalid. Recall that all unchecked exceptions are a subclass of the runtime exception `java.lang.RuntimeException`. The class RuntimeException has the following constructor:

`RuntimeException(String message)`

that constructs a new runtime exception with the specified detail message message. The message can be retrieved by the `getMessage()` method. You can test your code by running the `Test1.java` provided. Make sure your code follows the CS2030S Java style.

(c) **Operation**

Create an abstract class called `Operation` with the following fields and methods:

two private fields that correspond to two expressions (an expression is as defined at the beginning of this question).

a class factory method `of`, which returns the appropriate subclass that implements a specific operation. The first parameter `of` the of methods is a `char` to indicate the operation to be performed. You need to support three operations: if the first parameter is `*`, return an operation that performs multiplication

on integers if the first parameter is +, return an operation that performs concatenation on strings if the first parameter is^, return an operation that performs XOR on booleans if the first parameter is none of the above, return `null`.

Note that the operator to perform XOR on two boolean variables is^. For instance,

```
jshell> Operation o = Operation.of('*', new Operand(2), new Operand(3));
jshell> o.eval()
$.. ==> 6

jshell> Operation o = Operation.of('+', new Operand("hello"), new Operand("world"));
jshell> o.eval()
$.. ==> "helloworld"

jshell> Operation o = Operation.of('^', new Operand(true), new Operand(false));
jshell> o.eval()
$.. ==> true

jshell> Operation.of('!', new Operand(2), new Operand(3));
$.. ==> null

jshell> Operation o1 = Operation.of('*', new Operand(2), new Operand(3));
jshell> Operation o = Operation.of('*', o1, new Operand(4));
jshell> o.eval()
$.. ==> 24

jshell> Operation o2 = Operation.of('*', new Operand(2), new Operand(4));
jshell> Operation o = Operation.of('*', o1, o2);
jshell> o.eval()
$.. ==> 48
```

If the operands are not of the correct type, eval must throw an unchecked InvalidOperandException exception. For instance,

```
jshell> Operation o = Operation.of('*', new Operand("1"), new Operand(3));
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator *

jshell> Operation o = Operation.of('+', new Operand(1), new Operand(4));
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator +

jshell> Operation o = Operation.of('^', new Operand(false), new Operand(3));
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator ^

jshell> Operation o1 = Operation.of('*', new Operand(1), new Operand(3));
jshell> Operation o2 = Operation.of('^', new Operand(false), new Operand(false));
```

```
jshell> Operation o = Operation.of('+', o1, o2);
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator +

jshell> Operation o1 = Operation.of('*', new Operand(1), new Operand("3"));
jshell> Operation o2 = Operation.of('^', new Operand(false), new Operand(false));
jshell> Operation o = Operation.of('+', o1, o2);
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator *
```

You can test your code by running the `Test2.java` provided. Make sure your code follows the CS2030S Java style.

2. (15 points) **Source List**

Marking Criteria

- Correctness (3 marks)
- Design (10 marks)
- Style (2 marks)

In this question, you will create a generic list using a `Pair` class which is similar to what you saw in lectures. *Note:* This generic list is different and unrelated to the `java.util.List` interface. We will not use that here. We will call this generic list `SourceList` (after the Source language used in the module CS1101S).

This question also uses the `Transformer` and `BooleanCondition` interfaces from Lab 4.

Remember the `Pair` class from lectures with a `first` and a `second` value. The implementation of `Pair` used in this question is different from that in the lectures, in that it has only one type parameter `T`. Using this implementation it is possible to create a list using pairs: The generic list `SourceList` is just a `Pair` object, whose second value is either itself a `Pair` object, or an `EmptyList` object.

This chain of pairs constitutes a `SourceList`. Each chain of pairs is terminated with an `EmptyList` object.

Consider some examples:

```
jshell> Operation o = Operation.of('*', new Operand("1"), new Operand(3));
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator *

jshell> Operation o = Operation.of('+', new Operand(1), new Operand(4));
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator +

jshell> Operation o = Operation.of('^', new Operand(false), new Operand(3));
jshell> try {
```

```
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator ^

jshell> Operation o1 = Operation.of('*', new Operand(1), new Operand(3));
jshell> Operation o2 = Operation.of('^', new Operand(false), new Operand(false));
jshell> Operation o = Operation.of('+', o1, o2);
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator +

jshell> Operation o1 = Operation.of('*', new Operand(1), new Operand("3"));
jshell> Operation o2 = Operation.of('^', new Operand(false), new Operand(false));
jshell> Operation o = Operation.of('+', o1, o2);
jshell> try {
   ...>   o.eval();
   ...> } catch (InvalidOperandException e) {
   ...>   System.out.println(e.getMessage());
   ...> }
ERROR: Invalid operand for operator *
```

You have been provided with the following skeletons:

- The `SourceList` interface: `SourceList.java`
- The `EmptyList` class: `EmptyList.java`
- The `Pair` class: `Pair.java`

Familiarise yourself with these files.

The `Pair::toString` method has already been implemented for you. Read and understand how this method works by recursively calling the `toString` method of the next pair in the pair chain.

You will now implement more methods for your `SourceList` interface in the `Pair` class.

(a) **Implement the `length` method**

An important `SourceList` operation is to calculate the length of the list.

Implement the `length` method which takes in no arguments and returns an int which is the length of the list.

Consider some examples:

```
jshell> SourceList<String> strList = new Pair<>("AAA",
   ...>     new Pair<>("AA", new Pair<>("A", new EmptyList<>())))
jshell> strList.length()
$.. ==> 3

jshell> EmptyList<Integer> intEmpty = new EmptyList<>()
jshell> intEmpty.length()
$.. ==> 0
```

(b) **Implement the `equals` method**

Implement the `equals` method which takes in an argument of type `Object` and returns a `boolean`. This `equals` method should return `true` when the two `SourceList` are equal, and `false` otherwise. A `SourceList` is equal if, in all `Pairs` of the `SourceList`, the `first` values are equal. All `EmptyLists` are equal.

Consider the following example:

```
jshell> Object intList1 = new Pair<>(1, new Pair<>(2,
   ...>     new Pair<>(3, new EmptyList<>())));
jshell> Object intList2 = new Pair<>(1, new Pair<>(2,
   ...>     new Pair<>(3, new EmptyList<>())));
jshell> intList1.equals(intList1)
$.. ==> true
jshell> intList1.equals(intList2)
$.. ==> true
jshell> intList1.equals(new Pair<>("1", new Pair<>("2",
   ...>     new Pair<>("3", new EmptyList<>()))))
$.. ==> false
jshell> intList1.equals(new EmptyList<>())
$.. ==> false
jshell> new EmptyList<Integer>().equals(new EmptyList<String>())
$.. ==> true
jshell>
```

Note: you may only use one `@SuppressWarnings` for this method. Nowhere else in your source code may you use any `@SuppressWarnings`.

You can test your code by running the `Test3.java` provided. Make sure your code follows the CS2030S Java style.

(c) **Implement the `filter` method**

Implement the `filter` method which takes in a parameter of type `BooleanCondition` and returns a new `SourceList` containing only the elements which match the `BooleanCondition`. *Note*: that this should create new pairs and not change the current `SourceList`.

Consider the following example:

```
jshell> SourceList<Integer> intList = new Pair<>(1,
   ...>     new Pair<>(2, new Pair<>(3, new Pair<>(4, new EmptyList<>()))));
jshell> intList.filter(new GreaterThanTwo())
$.. ==> 3, 4, EmptyList

jshell> intList
intList ==> 1, 2, 3, 4, EmptyList
jshell> intList.filter(new GreaterThanTwo()) == intList
$.. ==> false

jshell> SourceList<Integer> l = intList.filter(new GreaterThanTwo())
jshell> SourceList<String> l = intList.filter(new GreaterThanTwo())
|  Error:
|  incompatible types: SourceList<java.lang.Integer> cannot be converted to SourceList<java
|  SourceList<String> l = intList.filter(new GreaterThanTwo());
|                         ^---------------------------------^

jshell> new EmptyList<Integer>().filter(new GreaterThanTwo())
$.. ==> EmptyList

jshell> SourceList<Integer> l = new EmptyList<Integer>().filter(new GreaterThanTwo())
jshell> SourceList<String> l = new EmptyList<Integer>().filter(new GreaterThanTwo())
|  Error:
|  incompatible types: SourceList<java.lang.Integer> cannot be converted to
|  SourceList<java.lang.String>
|  SourceList<String> l = new EmptyList<Integer>().filter(new GreaterThanTwo());
|                         ^-------------------------------------------------^

jshell> intList.filter(new GreaterThanTwo())
   ...>        .filter(new GreaterThanTwo());
$.. ==> 3, 4, EmptyList
```

```
jshell> new EmptyList<Integer>().filter(new GreaterThanTwo())
   ...>                         .filter(new GreaterThanTwo())
$.. ==> EmptyList
```

Make sure your `filter` method is as flexible as you can make it.

```
jshell> class A implements BooleanCondition<Object> {
   ...>   public boolean test(Object o) {
   ...>     return false;
   ...>   }
   ...> }
jshell> intList.filter(new A());
jshell>
```

You can test your code by running the `Test4.java` provided. Make sure your code follows the CS2030S Java style.

(d) **Implement the `map` method**

Implement the `map` method which takes in a parameter of type `Transformer` and returns a new `SourceList` where all elements of the new `SourceList` are the result of applying the `Transformer` to all the elements of the original `SourceList`. Note: that this should create new pairs and not change the current `SourceList`.

Consider the following example:

```
jshell> SourceList<Integer> intList = new Pair<>(1,
   ...>     new Pair<>(2, new Pair<>(3, new Pair<>(4,
   ...>         new EmptyList<>()))))
jshell> intList.map(new IntegerToString())
$.. ==> "1", "2", "3", "4", EmptyList
jshell> intList
intList ==> 1, 2, 3, 4, EmptyList

jshell> SourceList<Integer> intList = new Pair<>(1,
   ...>     new Pair<>(2, new Pair<>(3,
   ...>         new Pair<>(4, new EmptyList<>()))))
jshell> SourceList<String> l = intList.map(new IntegerToString())

jshell> SourceList<Integer> l = intList.map(new IntegerToString())
|  Error:
|  incompatible types: inference variable U has incompatible bounds
|      equality constraints: java.lang.Integer
|      lower bounds: java.lang.String
|  SourceList<Integer> l = intList.map(new IntegerToString());
|                          ^--------------------------------^

jshell> new EmptyList<Integer>().map(new IntegerToString())
$.. ==> EmptyList

jshell> SourceList<String> l = new EmptyList<Integer>().map(new IntegerToString())

jshell> SourceList<Integer> l = new EmptyList<Integer>().map(new IntegerToString())
|  Error:
|  incompatible types: inference variable U has incompatible bounds
|      equality constraints: java.lang.Integer
|      lower bounds: java.lang.String
|  SourceList<Integer> l = new EmptyList<Integer>().map(new IntegerToString());
|                          ^-------------------------------------------------^
```

Make sure your `map` method is as flexible as you can make it.

```
jshell> class A implements BooleanCondition<Object> {
   ...>   public boolean test(Object o) {
```

```
   ...>     return false;
   ...>   }
   ...> }
jshell> intList.filter(new A());
jshell>
```

(e) **Implement the `StringToLength` class**

This class should implement the `Transformer` interface. The method of the class should take in a `String` and return an `Integer` which is the length of the `String`.

Hint: The `java.lang.String::length()` method may come in handy.

Consider the following example:

```
jshell> SourceList<String> strList = new Pair<>("AA",
   ...>     new Pair<>("A", new Pair<>("", new EmptyList<>())));
jshell> strList.map(new StringToLength())
$.. ==> 2, 1, 0, EmptyList

jshell> SourceList<Integer> l = strList.map(new StringToLength())
jshell> SourceList<String> l = strList.map(new StringToLength())
|  Error:
|  incompatible types: inference variable U has incompatible bounds
|      equality constraints: java.lang.String
|      lower bounds: java.lang.Integer
|  SourceList<String> l = strList.map(new StringToLength());
|                         ^-------------------------------^

jshell> SourceList<Integer> intList = new Pair<>(1,
   ...>     new Pair<>(2, new Pair<>(3, new Pair<>(4, new EmptyList<>()))))
jshell> intList.map(new IntegerToString()).map(new StringToLength())
$.. ==> 3, 3, 3, 3, EmptyList
jshell> new EmptyList<Integer>().map(new IntegerToString()).map(new StringToLength())
$.. ==> EmptyList
jshell> strList.map(new StringToLength()).filter(new GreaterThanTwo())
$.. ==> EmptyList
jshell> intList.filter(new GreaterThanTwo()).map(new IntegerToString())
$.. ==> "3", "4", EmptyList
```

You can test your code by running the `Test5.java` provided. Make sure your code follows the CS2030S Java style.

END OF PAPER