C212 Final Exam (130 points)
December 2023

C212 Final Exam Rubric

1. (60 points) In functional programming languages, we often use three operations to act on data structures akin to linked lists: *cons*, *first*, and *rest*. In this question, you will implement a *cons*-like data structure, since Java has no real equivalent.

   We can define a *cons* list as follows:

```
A ICons is one of:
  - EmptyConsList
  - new ConsList(x, ICons)
```

   We need a way of linking these two types together, i.e., `EmptyConsList` and `ConsList`. So, we will design an interface `ICons<T>` to hook the two together.

```
interface ICons<T extends Comparable<T>> extends Comparable<ICons<T>> {

  /**
   * Retrieves the first element of the list.
   */
  T getFirst();

  /**
   * Retrieves the rest of the list, i.e., the list without
   * the first element.
   */
  ICons<T> getRest();

  /**
   * Determines whether this list is the empty list.
   */
  boolean isEmpty();
}
```

(a) *(10 points)* Design the `ConsList<T>` class and the `EmptyConsList<T>` classes. Both classes should implement the `ICons<T>` interface. The former should store two variables: an element of type `T`, and a `ICons<T>` representing the rest of the list. The latter should store no instance variables. **Do not override/implement the methods defined inside `ICons<T>` yet—we will do that in subsequent steps.**

Solution.

Rubric:

- (2 pts) `ConsList` implements `ICons<T>`. −1 if they do not include the generic.
- (2 pts) `ConsList` stores an instance variable of type `T` and is private. −1 for each incorrect.
- (3 pts) `ConsList` stores an instance variable of type `ICons<T>` and is private. −1 if it is not generic, −1 if it is not `ICons`, and −1 if it is not private.
- (2 pts) `EmptyConsList` implements `ICons<T>`. −1 if they do not include the generic.
- (1 pt) `EmptyConsList` has an empty class definition.

```
class ConsList<T extends Comparable<T>> implements ICons<T> {

  private T first;
  private ICons<T> rest;
}

class EmptyConsList<T extends Comparable<T>> implements ICons<T> {}
```

(b) *(5 points)* Design a `private` constructor for `ConsList`. It should receive the `first` and `rest` arguments, and assign them to the respective instance variables.

Solution.

Rubric:

- (1 pt) constructor is private.
- (1 pt) receives an argument of type `T`.
- (2 pt) receives an argument of type `ICons<T>`.
- (1 pt) assigns both parameters to the instance variables correctly.

```
private ConsList(T first, ICons<T> rest) {
  this.first = first;
  this.rest = rest;
}
```

(c) *(3 points)* Design the `public` constructor for `EmptyConsList`.

Solution.

Rubric:

- (1 pt) class is public.
- (2 pts) correctly named and receives no parameters.

```
public EmptyConsList() {}
```

(d) *(5 points)* Inside `ConsList`, override the `getFirst`, `getRest`, and `isEmpty` methods, which retrieve the respective parts of the list, and determine if the list is empty respectively.

Solution.

Rubric:

- (2 pts) correct `getFirst`
- (2 pts) correct `getRest`
- (1 pt) correct `isEmpty`.

```
@Override
public T getFirst() { return this.first; }

@Override
public ICons<T> getRest() { return this.rest; }

@Override
public boolean isEmpty() { return false; }
```

(e) *(5 points)* Inside `EmptyConsList`, override the `getFirst` and `getRest` methods, wherein each method throws an `IllegalOperationException`, since accessing the parts of an empty list is nonsensical. Then, override `isEmpty` as appropriate.

Solution.

Rubric:

- (2 pts) correct `getFirst`
- (2 pts) correct `getRest`
- (1 pt) correct `isEmpty`.

```
@Override
public T getFirst() { throw new IllegalOperationException(); }

@Override
public ICons<T> getRest() { throw new IllegalOperationException(); }

@Override
public boolean isEmpty() { return true; }
```

(f) *(7 points)* Inside `ConsList`, write the `setFirst` and `setRest` methods, which respectively mutate the given list instance variables.

Solution.

Rubric:

- (3 pts) `setFirst` correctly sets the instance variable to the parameter (1 pts). Method is void (1 pt), parameter is `T` (1 pt).
- (4 pts) `setRest` correctly sets the instance variable to the parameter (1 pts). Method is void (1 pt), parameter is `ICons` (1 pt), and is generic (1 pt).

```
public void setFirst(T fst) { this.first = fst; }

public void setRest(ICons<T> rst) { this.rest = rst; }
```

(g) *(7 points)* Inside `ConsList`, write the `static ConsList<T> cons(T first, ICons<T> rest)` method, which invokes the `private` constructor and returns a new *cons* list.

Solution.

Rubric:

- (2 pts) they copied the signature correctly.
- (2 pts) Calls the private constructor.
- (3 pts) correctly populates the private constructor.

```
public static ConsList<T> cons(T first, ICons<T> rest) {
  return new ConsList(first, rest);
}
```

(h) *(8 points)* We see that `ICons` extends `Comparable<ICons<T>>`, meaning that it has to provide a definition of `compareTo`. Override `compareTo` in both `ConsList` and `EmptyConsList` to compare the elements of a *cons* list. If, in `ConsList`, the argument is not a `ConsList`, return 1.

Solution.

Rubric:

- (1 pt) base case is correct in `ConsList` compareTo
- (1 pt) correctly compares the first elements.
- (1 pt) returns 0 if they are the same.
- (1 pt) Recursively calls compareTo on the rest of both lists.
- (4 pts) Returns -1 if the parameter is not an empty list, and 0 otherwise.

```
class ConsList<T> implements Comparable<T> {
  // Other stuff not shown.

  @Override
  public int compareTo(T t) {
    if (t.isEmpty()) { return 1; }
    else {
      int fres = this.first.compareTo(t);
      return fres == 0 ? this.rest.compareTo(t.rest) : fres;
    }
  }
}

class EmptyConsList<T> implements Comparable<T> {
  // Other stuff not shown.

  @Override
  public int compareTo(T t) {
    return !t.isEmpty() ? -1 : 0;
  }
}
```

(i) *(10 points)* Write coherent tests for your `ICons<T>` data structure. In particular, you should test the following methods: `getFirst`, `rest`, `setFirst`, `setRest`, and `isEmpty`. It might make sense to create a couple of lists outside each test method, then test them inside those methods.

Rubric:

- A test for `ConsList` and the `EmptyConsList` inside each method. (1 pt for each up to a max of 10)

2. (30 points) This question has five parts. We need to provide some background for the question first. An *encoded string* S is one of the form:

$$S = n[S']$$
$$S' = SS' \mid [a, ..., z]^* \mid \texttt{""}$$

We imagine this didn't clear up what the definition means. Take the encoded string `"3[a]2[b]"` as an example. The resulting decoded string is `"aaabb"`, because we create three copies of `"a"`, followed by two copies of `"b"`. Another example is `"4[abcd]"`, which returns the string containing `"abcdabcdabcdabcd"`.

(a) *(6 points)* First, write a method `retrieveN` that returns the integer at the start of an encoded string. Take the following examples as motivation. Hint: use `indexOf`, `substring`, and `Integer.parseInt`.

```
retrieveN("3[a]2[b]")     => 3
retrieveN("47[abcd]")     => 47
retrieveN("1[bbbbb]3[a]") => 1
```

Solution.

Rubric:

- (2 pts) Signature is correct: one point for parameter and return type.
- (4 pts) Definition is correct: retrieves and returns the integer. If it does not account for more than single-digit numbers, −2.

```
static int retrieveN(String s) {
  int l = s.indexOf("[");
  String sub = s.substring(0, l);
  return Integer.parseInt(sub);
}
```

(b) *(6 points)* Next, write the `cutN` that returns a string without the integer at the start of an encoded string. Hint: use `indexOf` and `substring`.

```
cutN("3[a]2[b]")     => "[a]2[b]
cutN("47[abcd]")     => "[abcd]"
cutN("1[bbbbb]3[a]") => "[bbbbb]3[a]"
```

Solution.

Rubric:

- (2 pts) Signature is correct: one point for parameter and return type.
- (4 pts) Definition is correct: returns the substring after the first integer.

```
static String cutN(String s) {
  int l = s.indexOf("[");
  return s.substring(l);
}
```

(c) *(6 points)* Design the *standard recursive* `decode` method, which receives an encoded string and performs a decoding operation. Hint: use s`.repeat(`n`)`, which receives an integer n and operates on a string s, returning a new string with n copies of s.

Solution.

Rubric:

- (1 pt) uses standard recursive.
- (2 pts) correct signature.
- (3 pts) correct return values.

```
static String decode(String s) {
  if (s.isEmpty()) { return ""; }
  else {
    int v = retrieveN(s);
    String ss = cutN(s);
    String sss = ss.substring(1, ss.indexOf("]");
    return sss.repeat(v) + decode(s.substring(s.indexOf("]") + 1);
  }
}
```

(d) *(6 points)* Design the `decodeTR` and `decodeTRHelper` methods. The former acts as the driver to the latter; the latter solves the same problem that `decode` does, but it instead uses tail recursion. Remember to include the relevant access modifiers! Hint: use s`.repeat(`n`)`.

Solution.

Rubric:

- (1 pt) correct driver method.
- (1 pt) tail recursive method uses `private` access modifier.
- (2 pts) correct conditionals.
- (3 pts) correctly updates accumulator and s.

```
static String decodeTR(String s) {
  return decodeTRHelper(s, "");
}

private Static String decodeTRHelper(String s, String acc) {
  if (s.isEmpty()) return acc;
  else {
    int n = retrieveN(s);
    String ss = cutN(s);
    String sss = ss.substring(1, ss.indexOf("]")
    return decodeTRHelper(s.substring(s.indexOf("]") + 1, acc + sss.repeat(v));
  }
}
```

(e) *(6 points)* Design the `decodeLoop` method, which solves the problem using either a `while` or `for` loop. Hint: use s`.repeat(`n`)`.

Solution.

Rubric:

- (1 pt) correct signature.
- (1 pt) localized accumulator.
- (2 pts) correct loop condition.
- (2 pts) correctly updates local variables.
- (1 pt) correct return value.

```
static String decodeLoop(String s) {
  String acc = "";
  while (!(s.isEmpty())) {
    int v = retrieveN(s);
    String ss = cutN(s);
    String sss = ss.substring(1, ss.indexOf("]");
    s = s.substring(s.indexOf("]") + 1);
    acc += sss.repeat(v);
  }
  return acc;
}
```

3. (20 points) The *substitution cipher* is a text cipher that encodes an alphabet string A (also called the *plain-text alphabet*) with a key string K (also called the *cipher-text alphabet*). The A string is defined as `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`, and K is any permutation of A. We can encode a string s using K as a mapping from A. For example, if K is the string `"ZEBRASCDFGHIJKLMNOPQTUVWXY"` and s is `"FLEE AT ONCE. WE ARE DISCOVERED!"`, the result of encoding s produces `"SIAA ZQ LKBA. VA ZOA RFPBLUAOAR!"`

   Design the `subtitutionCipher` method, which receives a plain-text alphabet string A, a cipher-text string K, and a string s to encode, `substitutionCipher` should return a string s' using the aforementioned substitution cipher algorithm. You must follow the "design recipe" laid out in class. That is, you must write the method purpose statement comment, tests, and the implementation.

   The skeleton code is on the next page.

   Solution.

   Rubric:

   - (4 pts) at least two coherent examples.
   - (2 pts) sensible purpose statement.
   - (14 pts) definition works as expected.

```
class SubstitutionCipherTester {

  @Test
  void substitutionCipherTest() {
    assertAll(
        Some sensible examples... :D
    );
  }
}

import java.util.*; // Import all necessary collections.

class SubstitutionCipher {

  static String substitutionCipher(String A, String K, String s) {
    String res = "";
    Map<Character, Character> sub = new HashMap<>();
    for (int i = 0; i < A.length(); i++) { sub.put(A.charAt(i), K.charAt(i)); }
    for (char c : s.toCharArray()) { res += sub.get(c); }
    return res;
  }
}
```

4. (20 points) Oh no! Joshua's cat, Butterscotch, has scratched part of this exam away and we need you to fix the missing code. Fill in the missing code for this quick sort implementation. Note that this is a *functional* implementation of the quick sort, which means that we return a new array rather than sorting the one we provide.

Solution.

Rubric:

- -1 point for each incorrect blank up to -20. If they use `AbstractList` or use `T` for the type of value returned by the random method, just accept it.

```
import java.util.List;

interface IQuickSort<T extends Comparable<T>> {

  List<T> quicksort(List<T> ls);
}


class FunctionalQuickSort<T> implements IQuickSort<T> {

  @Override
  public List<T> quicksort(List<T> A) {
    if (A.isEmpty()) { return A; }
    else {
      // Choose a random pivot.
      int pivot = new Random().nextInt(A.size());

      // Sort the left-half.
      List<T> leftHalf = A.stream()
                            .filter(x -> x.compareTo(A.get(pivot)) < 0)
                            .collect(Collectors.toList());
      List<T> leftSorted = quicksort(leftHalf);

      // Sort the right-half.
      List<T>  rightHalf = A.stream()
                              .filter(x -> x.compareTo(A.get(pivot)) > 0)
                              .collect(Collectors.toList());
      List<T>  rightSorted = quicksort(rightHalf);

      // Get all elements equal to the pivot.
      List<T>  equal = A.stream()
                          .filter(x -> x.compareTo(A.get(pivot)) == 0)
                          .collect(Collectors.toList());

      // Merge the three.
      leftSorted.addAll(equal);
      leftSorted.addAll(rightSorted);
      return leftSorted;
    }
  }
```

```
}
```