CMSC 132A, Final exam (Practice)
SOLUTION

Spring 2018

NAME:_____

UID: _____

Question	Points
1	15
2	15
3	15
4	15
5	15
6	15
Total:	90

This test is open-book, open-notes, but you may not use any computing device other than your brain and may not communicate with anyone. You have 120 minutes to complete the test.

The phrase "design a program" means follow the steps of the design recipe. Unless specifically asked for, you do not need to provide intermediate products like templates or stubs, though they may be useful to help you construct correct solutions.

You may use any of the data definitions given to you within this exam and do not need to repeat their definitions.

Unless specifically instructed otherwise, you may use any Java language features we have seen in class.

You do not need to write visibility modifiers such as `public` and `private`.

When writing tests, you may use a shorthand for writing check-expects by drawing an arrow between two expressions to mean you expect the first to evaluate to same result as the second. For example, you may write `"foo".length()` $\rightarrow$ `3` instead of `t.checkExpect("foo".length(), 3)`. You do not have to place tests inside a test class.

Problem 1 (15 points). Succintly answer each of the following questions:

If a (concrete, non-abstract) class `C` implements interface `I` but doesn't actually define all of the methods listed in `I` within the definition of `C`, what can you conclude about `C` (assuming the Java type-checker accepts this program)?

SOLUTION:

C must extend some class that implements I and defines those methods.

If you have a method `Integer m(String s)`, what do you know about possible inputs to the method when the program is run?

SOLUTION:

s is either a string or null.

Suppose you have a program that contains the following interface and class definitions (and potentially others):

```
interface I {
  Boolean m1();
  Boolean m2();
}
abstract class A implements I {
  public Boolean m1() { return this.m2(); }
}
class B extends A {
  public Boolean m2() { return true; }
}
```

If you have an object `o` of type `A`, what does `o.m1()` produce?

SOLUTION:

Either true, false, or null.

Besides when multiple classes have identical method definitions, when (if ever) does it make sense to define methods in an abstract class?

SOLUTION:

```
When several classes have mostly identical methods, but some don't.
The common code can be lifted and the uncommon code can override
in the exceptional classes.
```

List three *conceptual* differences between Java and ISL. (Examples of non-conceptual differences: the syntax, the IDE, the number of programmers, etc.)

SOLUTION:

```
- Java has a type system which rules out a certain class of errors
- Java uses object dispatch to perform case analysis, ISL uses cond
- Java has mutable fields in objects, in ISL everything is immutable
```

Problem 2 (15 points). Here's an idea for a sorting method that, given a comparator, sorts a list into ascending order according the comparator. This method is implemented in the `AListof<X>` abstract class:

```
// Sort this list in ascending order according to c
public Listof<X> sort(Comparator<X> c) {
    // Convert this list into a list of singleton lists
    // For example [1, 2, 3] --> [[1], [2], [3]]
    // Note: each singleton list is a sorted list.
    Listof<Listof<X>> singletons =
      this.map(x -> new Cons<>(x, new Empty<>()));

    // Combining function to merge two sorted lists into one.
    BiFunction<Listof<X>, Listof<X>, Listof<X>> combine =
           (s1, s2) -> s1.merge(s2, c);

    // Fold over singleton lists, merging into a sorted list
    return singletons.foldr(combine, new Empty<X>());
}
```

This method works (although it's not a very efficient algorithm!), assuming the following `merge` method in the `Listof<X>` interface:

```
// Merge all the elements of this list and that list into a sorted list
// ASSUME: this and that are already sorted according to c.
Listof<X> merge(Listof<X> that, Comparator<X> c);
```

Some examples (written in exam shorthand notation):

```
Comparison<Integer> lt = (i, j) -> i - j;
[1,2,3].merge([4,5,6], lt) --> [1,2,3,4,5,6]
[1,3,5].merge([2,3,6], lt) --> [1,2,3,4,5,6]
[4,5,6].merge([1,2,3], lt) --> [1,2,3,4,5,6]
```

Design the `merge` method so that `sort` works. You may design and add any needed helper methods for the `Listof<X>` interface, but you cannot change the original `sort` method (nor use it!). For full credit, your `merge` method should take time linear in the length of the resulting list, i.e. the sum of the length of this and that.

[Space for problem 2.]

SOLUTION:

```
// In Listof<X>

// Merge all the elements of this and that into a sorted list
// ASSUME: this and that are already sorted according to c.
Listof<X> merge(Listof<X> that, Comparator<X> c);

// Merge this list with the given cons into a sorted list
// ASSUME: this and cons are each sorted according to c.
Listof<X> mergeCons(Cons<X> cons, Comparator<X> c);

// In Empty<X>

Listof<X> merge(Listof<X> that, Comparator<X> c) { return that; }
Listof<X> mergeCons(Cons<X> cons, Comparator<X> c) { return cons; }

// In Cons<X>

Listof<X> merge(Listof<X> that, Comparator<X> c) {
  return that.mergeCons(this, c);
}

Listof<X> mergeCons(Cons<X> cons, Comparator<X> c) {
  return c.compare(cons.first, this.first) < 0 ?
          new Cons<>(cons.first, this.merge(cons.rest, c)) :
          new Cons<>(this.first, this.rest.mergeCons(cons, c));
}
```

Problem 3 (15 points). Here is the usual definition for binary trees:

```
interface BT<X> {}

class Leaf<X> implements BT<X> {}
class Node<X> implements BT<X> {
  X val;
  BT<X> left;
  BT<X> right;
  Node(X val, BT<X> left, BT<X> right) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}
```

Define the following methods to implement structural equality and respect the law of hash codes (your `hashCode` method should distinguish at least some nodes):

```
// Is this tree structurally equal to that tree?
boolean equals(BT<X> that);

// Hash code for this tree
int hashCode();
```

[Space for problem 3.]

SOLUTION:

```
lf1 = new Leaf<>();
lf2 = new Leaf<>();
tr1 = new Node<>(4, lf1, lf2);
tr2 = new Node<>(4, lf2, lf1);
tr3 = new Node<>(4, tr2, lf1);

lf1.equals(lf2) --> true
lf1.equals(tr1) --> false
tr1.equals(tr2) --> true
tr1.equals(tr3) --> false

lf1.hashCode() --> 9
tr1.hashCode() --> 4

// In Leaf

boolean equals(BT<T> that) { return that.equalsLeaf(this); }
Boolean equalsLeaf(Leaf<T> l) { return true; }
Boolean equalsNode(Node<T> n) { return false; }
int hashCode() { return 9; }

// In Node

boolean equals(BT<T> that) { return that.equalsNode(this); }
Boolean equalsLeaf(Leaf<T> l) { return false; }
Boolean equalsNode(Node<T> n) {
  return this.value.equals(n.value) &&
          this.left.equals(n.left) &&
          this.right.equals(n.right);
}

int hashCode() {
  return this.value.hashCode();
}
```

Problem 4 (15 points). Here is a data representation for (a small subset of) JSON documents:

```
interface Json {}

class JStr implements Json {
  String s;
  JStr(String s) { this.s = s; }
}

class JInt implements Json {
  Integer i;
  JInt(Integer i) { this.i = i; }
}

interface LoJson extends Json {}

class JMt implements LoJson {}

class JCons implements LoJson {
  Json first;
  LoJson rest;
  JCons(Json first, LoJson rest) {
    this.first = first;
    this.rest = rest;
  }
}
```

Implement the visitor pattern for `Json` documents. (You do not need to write tests for this problem.)

[Space for problem 4.]

SOLUTION:

```
// Computations over Json documents producing Rs
interface JsonVisitor<R> {
  R visitJStr(JStr js);
  R visitJInt(JInt ji);
  R visitJMt(JMt jmt);
  R visitJCons(JCons jcons);
}

// In Json

// Visit this Json document
<R> R accept(JsonVisitor<R> v);

<R> R accept(JsonVisitor<R> v)
// In JInt  ....              { return v.visitJInt(this); }
// In JStr  ....              { return v.visitJStr(this); }
// In JMt   ....              { return v.visitJMT(this); }
// In JCons ....              { return v.visitCons(this); }
```

Problem 5 (15 points). Here is a representation for directed graphs:

```
// A node in a directed graph
class GNode {
    String name;
    Listof<GNode> neighbors;

    GNode(String name) {
        this.name = name;
        this.neighbors = new Empty<>();
    }

    // EFFECT: add the given node to this node's neighbor
    public void addNeighbor(GNode n) {
        this.neighbors = new Cons<>(n, this.neighbors);
    }

    // Are there any cycles in this graph reachable from this node?
    public Boolean hasCycle() { ... }
}
```

Design the `hasCycle` method (examples on the next page).

```
void testGraph(Tester t) {
    // +----------+
    // v          |
    // A--->B--->C-->D
    //      +--->E-->F
    //
    GNode a = new GNode("A");
    GNode b = new GNode("B");
    GNode c = new GNode("C");
    GNode d = new GNode("D");
    GNode e = new GNode("E");
    GNode f = new GNode("F");

    a.addNeighbor(b);
    b.addNeighbor(c);
    c.addNeighbor(d);
    b.addNeighbor(e);
    e.addNeighbor(f);
    c.addNeighbor(a);

    t.checkExpect(a.hasCycle(), true);
    t.checkExpect(b.hasCycle(), true);
    t.checkExpect(c.hasCycle(), true);
    t.checkExpect(d.hasCycle(), false);
    t.checkExpect(e.hasCycle(), false);
    t.checkExpect(f.hasCycle(), false);
}

SOLUTION:

// Is there a cycle reachable from this node?
Boolean hasCycle() {
  return this.hasCycleAcc(new Empty<>());
}

// Is there a node reachable that's already been seen?
// ACCUM: list of nodes seen so far
Boolean hasCycleAcc(Listof<GNode> seen) {
    return seen.exists(n -> n.equals(this)) ||
            this.neighbors.exists(n ->
                    n.hasCycle(new Cons<>(this, seen));
}
```

Problem 6 (15 points). Design the follow method which is given a function produces doubles and a (non-empty) array list of elements. It produces an element of the list such that applying `f` to that element produces a number less than or equal to applying `f` to any other element in the list.

```
class Algorithm {
  // Find an element of xs that minimizes f.
  // ASSUME: xs has at least one element
  static <T> T argmin(Function<T,Double> f, ArrayList<T> xs) { ... }
}
```

SOLUTION:

```
static <T> T argmin(Function<T,Double> f, ArrayList<T> xs) {
  T minx = xs.get(0);
  Double minfx = f.apply(minx);
  for (T x : xs) {
    Double fx = f.apply(x);
    if (fx < min) {
      minx = x;
      minfx = fx;
    }
  }
  return minx;
}
```

```
as = new ArrayList<>(["hi", "there", "!", "friendo"]);
Algorithm.argmin(s -> s.length(), as) --> "!"
```