CMSC 132A, Final exam
SOLUTION

Spring 2018

NAME:_____

UID: _____

Question	Points
1	20
2	15
3	15
4	15
5	15
6	5
Total:	85

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.

You may assume the following interface has been implemented for all `Listof<X>` objects:

```
interface Listof<X> extends Iterable<X> {

    // Compute the length of this list
    Integer length();

    // Reverse the elements of this list
    Listof<X> reverse();

    // Apply f to each element of this list
    <Y> Listof<Y> map(Function<X, Y> f);

    // Fold right over this list with combining function and base case
    // The fundamental abstraction of structural recursion list methods
    <Y> Y foldr(BiFunction<X, Y, Y> f, Y b);

    // Fold left over this list with combining function and base case
    // The fundamental abstraction of accumulator-based list methods
    <Y> Y foldl(BiFunction<X, Y, Y> f, Y b);

    // Append this list and xs
    Listof<X> append(Listof<X> xs);

    // Accept the given visitor and visit this list
    <R> R accept(ListVisitor<X, R> v);

    // Does any element of this list satisfy p?
    Boolean exists(Predicate<X> p);

    // Does every element of this list satisfy p?
    Boolean forAll(Predicate<X> p);

    // Produce the list of elements which satisfy p
    Listof<X> filter(Predicate<X> p);

    // Zip together this list and xs2 into a list of pairs
    // Runs out whenever the shorter list runs out
    <Y> Listof<Pairof<X, Y>> zip(Listof<Y> xs2);
}
```

You may write `[e1, e2, e3]` as shorthand for `new Cons<>(e1, new Cons<>(e2, new Cons<>(e3, new Empty<>()))))` for the purposes of test cases on this exam.

Problem 1 (20 points). Succintly answer each of the following questions:

What's the difference between an abstract class and an interface?

SOLUTION:

An abstract class consists of fields, constructors, and methods, but cannot be directly constructed (and need not implement all of the methods of the interfaces it claims to implement).

An interface has no fields, constructors, or method bodies; only method signatures.

What's the difference between `Iterable<T>` and `Iterator<T>`?

SOLUTION:

An iterator is an object representing some state of an iteration through collection of values; an iterable is an object that can produce an iterator.

When is it appropriate to use the double dispatch pattern?

SOLUTION:

When defining a computation that requires case analysis over multiple union data definitions.

How is a class definition in Java like a structure definition in ISL?

SOLUTION:

A class definition defines (sets of) compound data by defining some fixed number of fields, just like an ISL structure definition.

Suppose you were reading the definition of some class C and you wanted to know how many fields exist in every C object. How would you determine this?

SOLUTION:

Count up the number of fields in C and all of its super classes.

A hash table is represented as an array list of buckets, where each bucket is a list of key value pairs. What do all of the keys in each bucket list have in common with each other?

SOLUTION:

They have the same hasCode() modulo table size.

Problem 2 (15 points). Design a method for `Listof<X>` objects that (optionally) produces the first element in the list whose next element is equal to it (if such an element exists):

```
// In Listof<X>

// Find first element where next element is equal, if one exists
Optional<X> dupNext();
```

You may use any of the methods listed on page 2 or add methods to the interface. Your solution should take time proportional to the length of the list.

SOLUTION:

```
[].dupNext() --> Optional.empty
[1].dupNext() --> Optional.empty
[1,2,3].dupNext() -> Optional.empty
[1,2,2,4,4].dupNext() -> Optional.of(2)

// In Listof<X>

// Produce previous element such that first element is equal (if exsist)
// ACCUM: previous element seen in this list
Optional<X> dupNextAcc(X prev);

// In Empty<X>

Optional<A> dupNext() { return Optional.empty(); }
Optional<A> dupNextAcc(A prev) { return Optional.empty(); }

// In Cons<X>

Optional<A> dupNext() {
    return this.rest.dupNextAcc(this.first);
}

Optional<A> dupNextAcc(A prev) {
    return prev.equals(this.first) ?
            Optional.of(prev) :
            this.rest.dupNextAcc(this.first);
}
```

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;
  }
}
```

Design the following method for binary trees:

```
// Produce (optionally) the first element (in NLR order) satisfying p
Optional<X> findPred(Predicate<X> p);
```

[Space for problem 3.]

SOLUTION:

```
lf = new Leaf<>();
tr1 = new Node<>(4, lf, lf);
tr2 = new Node<>(6, lf, lf);
tr3 = new Node<>(5, tr1, tr2);

lf.findPred(n -> true) --> Optional.empty
t3.findPred(n -> (n % 2) == 0) --> Optioanl.of(4)
t3.findPred(n -> n > 5) --> Optioanl.of(6)
t3.findPred(n -> n > 8) --> Optioanl.empty

// In Leaf

Optional<X> findPred(Predicate<X> p) {
  return Optional.empty;
}

// In Node

Optional<X> findPred(Predicate<X> p) {
  if (p.test(this.val)) {
     return Optional.of(this.first);
  } else {
    Optional<X> r = this.left.findPred(p);
    if (r.isPresent()) {
      return r;
    } else {
      return this.right.findPred(p);
    }
  }
}
```

Problem 4 (15 points). Here is a data representation for (a small subset of) JSON documents, with the visitor pattern implemented.

```
interface Json {
  // Accept visitor v and visit this document
  <R> R accept(JsonVisitor<R> v);
}

class JStr implements Json {
  String s;
  JStr(String s) { this.s = s; }
  <R> R accept(JsonVisitor<R> v) {
    return v.visitJStr(this);
  }
}

class JInt implements Json {
  Integer i;
  JInt(Integer i) { this.i = i; }
  <R> R accept(JsonVisitor<R> v) {
    return v.visitJInt(this);
  }
}

interface LoJson extends Json {}

class JMt implements LoJson {
  <R> R accept(JsonVisitor<R> v) {
    return v.visitJMt(this);
  }
}

class JCons implements LoJson {
  Json first;
  LoJson rest;
  JCons(Json first, LoJson rest) {
    this.first = first;
    this.rest = rest;
  }
  <R> R accept(JsonVisitor<R> v) {
    return v.visitJCons(this);
  }
}
```

```
// 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);
}
```

Design a visitor for `Json` documents that computes the sum of the length of all the strings in the document.

SOLUTION:

```
ji = new JInt(5)
js1 = new JStr("hi")
js2 = new JStr("there")
j1 = new JCons(new JCons(ji, new JCons(js1, new JCons(js2, new JEmpty()))),
               new JEmpty())
sl = new SumLength()

ji.accept(sl) --> 0
js1.accept(sl) --> 2
js2.accept(sl) --> 5
j1.accept(sl) --> 7

class SumLength implements JsonVisitor<Integer> {
  Integer visitJInt(JInt ji) { return 0; }
  Integer visitJStr(JStr js) { return js.s.length(); }
  Integer visitJMt(JMt jm) { return 0; }
  Integer visitJCons(JCons jc) {
    return jc.first.accept(this) + jc.rest.accept(this);
  }
}
```

Problem 5 (15 points). Design the following method which is given an array list and produces an array list of *bigrams*, i.e. pairs of elements that are next to each other. For example, if an array list has elements 7, 8, 3, and 2, then the bigrams are (7,8), (8,3), and (3,2).

```
class Algorithm {
  // Produce all the bigrams of the given array list
  static <T> ArrayList<Pairof<T,T>> bigrams(ArrayList<T> xs) { ... }
}
```

SOLUTION:

```
static <T> ArrayList<Pairof<T,T>> bigrams(ArrayList<T> xs) {
    ArrayList<Pairof<T,T>> bigrams = new ArrayList<>();
    for (Integer i = 0; i < xs.size() - 1; i = i + 1) {
        bigrams.add(new Pairof<>(xs.get(i), xs.get(i+1)));
    }
    return bigrams;
}
```

```
Algorithm.bigrams([]) --> []
Algorithm.bigrams([1]) --> []
Algorithm.bigrams([1,2,3]) -> [(1,2), (2,3)]
```

Problem 6 (5 points). Briefly describe three of the most interesting things you learned over the course of CMSC 131/132A.

SOLUTION:

Open ended.