OCaml

1. (14 points) Recall the fold and member functions that we saw in the OCaml part of the course:

   ```ocaml
   let rec fold (c:'a -> 'b -> 'b) (b:'b) (l:'a list) : 'b =
   begin match l with
   | []    -> b
   | h::t  -> c h (fold c b t)
   end

   let rec member (x:int) (l:int list) =
   begin match l with
   | []     -> false
   | h::t   -> x = h || (member x t)
   end
   ```

   (a) Consider the following function, cryptically named f:

   ```ocaml
   let f (l: int list) =
   fold (fun (x:int) (m:int) -> if x > m then x else m) 0 l
   ```

   i. What is the value of the expression f [1;3;5;4;2]?
   Answer: 5

   ii. Suggest a better name for f.
   Answer: One good name would be nat_max, since it computes the maximum value of a list of natural numbers.

(b) Now consider the following function, cryptically named g.

   ```ocaml
   let g (l:int list list) =
   fold (fun (l:int list) (s:int list) ->
       fold (fun (x:int) (t:int list) ->
           if member x t then t else x::t) s l)
   ```

   i. What is the value of the expression g [[1;3;5];[];[3;2;3]]?
   Answer: [1;2;3]

   ii. Suggest a better name for g.
   Answer: One good name would be union. These nested folds compute the union of int lists when viewed as int sets — the order the ints appear in the output list is the reverse order in which they first appear in the list of lists when read from right to left (that's a consequence of using fold right instead of fold left).

Grading scheme: 6 points part a; 8 points part b; evenly split between subparts with generous partial credit
2. (12 points) Given the OCaml type definition for `a tree and the helper functions below, complete the definition of the function is_bst so that it returns `true exactly when its input is a well-formed binary search tree.

```ocaml
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree

let rec tree_lt t x =
  begin match t with
  | Empty -> true
  | Node(lt, a, rt) -> a < x && (tree_lt lt x) && (tree_lt rt x)
  end

let rec tree_gt t x =
  begin match t with
  | Empty -> true
  | Node(lt, a, rt) -> a > x && (tree_gt lt x) && (tree_gt rt x)
  end

Answer:

let rec is_bst (t: 'a tree): bool =
  begin match t with
  | Empty -> true
  | Node (lt, a, rt) ->
    is_bst lt && is_bst rt
    && tree_lt lt a
    && tree_gt rt a
  end

Grading scheme: 4 points for recursive structure and base case; 2 points each for the checks in the recursive case
```
Java

3. (10 points) Circle “True” or “False” after each statement.

(a) In Java, the special value null is often used in situations where an OCaml programmer would use an option type.

   Answer: True

(b) The value null can be assigned to a variable of any type in Java.

   Answer: False

(c) Suppose s is a variable of type String. The Java expression s.equals(s) always returns true.

   Answer: False

(d) In Java, all exceptions that might be raised while executing a method must either be caught within the method body or else declared in a raises clause in its header.

   Answer: False

(e) If a try block has a finally clause, then the statements in this clause will always be executed, regardless of whether the statements in the try block terminate normally or raise an exception.

   Answer: True

(f) Representing a set using a hash table is only a good idea if the hashCode method for the type of values being stored produces very few collisions.

   Answer: True

(g) If two Java objects test equal (o1.equals(o2) == true) then they should have the same hash code (o1.hashCode() == o2.hashCode()).

   Answer: True

(h) If two Java objects have the same hash code (o1.hashCode() == o2.hashCode()) then they should test equal (o1.equals(o2) == true).

   Answer: False

(i) A Java class definition may give several “overloaded” variants of the same method, but they must all take different numbers of parameters.

   Answer: False

(j) In most situations, code that runs as fast as possible is better than code that is easy to read.

   Answer: False

Grading scheme: One point per subproblem, no partial credit.
4. (22 points) Suppose we are using two-dimensional Java arrays to represent matrices of numbers. A flip is an operation that flips the matrix top-to-bottom. As in the image processing assignment, matrices are stored in left-to-right, top-to-bottom order. Here is a sample JUnit test case that shows the desired behavior. (The `assertArrayEquals` method checks element-by-element equality of arrays.)

```java
int[][] input = new int[] {
    new int[] {1, 2, 3, 4},
    new int[] {5, 6, 7, 8},
    new int[] {9,10,11,12}};

int[][] expectedOutput = new int[] {
    new int[] { 4, 3, 2, 1},
    new int[] { 8, 7, 6, 5},
    new int[] {12,11,10, 9}};

flip(input);

assertArrayEquals(input, expectedOutput);
```

Complete the implementation of the method `flip` below. Note that `flip` should do its work in place: rather than returning a new array with flipped contents, it should rearrange the values in the array it is given. In particular, it should not call `new`. You may assume that the input array is non-null and isn’t “ragged”: that is, the sub-arrays are also non-null and have the same length.

**Answer:**

```java
void flip(int[][] input) {
    for (int row = 0; row < input.length; row++) {
        for (int col = 0; col < (input[row].length / 2); col++) {
            int ocol = input[row].length - col - 1;
            int tmp = input[row][col];
            input[row][col] = input[row][ocol];
            input[row][ocol] = tmp;
        }
    }
}
```

**Grading scheme:** 5 points for correct double loop procedure; 6 points for only swapping halfway in inner loop; 5 points for correct swapping of values; 6 points for using a temp variable during swapping; small deductions for minor syntax errors; partial credit awarded.
5. (8 points) If we run the following program, what sequence of numbers gets printed?

```java
interface Foo {
    public void bar(int x);
}

class A {
    public Foo m (final int x) {
        System.out.println(x);
        return new Foo() {
            public void bar (int y) {
                System.out.println(x);
                System.out.println(y);
            }
        };
    }
}

class B extends A {
    public Foo n (final int x) {
        System.out.println(x);
        return new Foo() {
            public void bar (int y) {
                System.out.println(y);
                System.out.println(x);
            }
        };
    }
}

public class Inner {
    public static void main (String args[]) {
        A a = new A();
        B b = new B();
        a.m(1).bar(2);
        b.m(3).bar(4);
        b.n(5).bar(6);
    }
}
```

**Answer:** 1 1 2 3 3 4 5 6 5

*Grading scheme: 3 points for first triplet, 2 points for second triplet, 3 points for third triplet. No partial credit.*
6. (12 points) Consider the following class definitions:

```java
class S {
    
}
class T extends S {
    
}
class U extends S {
    
}
class V extends U {
    
}
```

For each of the questions below, circle all the correct answers—there may be zero, one, or more.

(a) What is the static type of x in the code below?
```
S x = new V();
x = new U();
```

a. Object  
  b. Answer: S  
  c. T  
  d. U  
  e. V

(b) What is the dynamic type of the value stored in x after running the code below?
```
S x = new V();
x = new U();
```

a. Object  
  b. S  
  c. T  
  d. Answer: U  
  e. V

(c) Which types can we place in the hole marked `__?__` below so that no `ClassCastException` is thrown at when this program is run?
```
Object o = new U();
Object x = (__?__)o;
```

a. Answer: Object  
  b. Answer: S  
  c. T  
  d. Answer: U  
  e. V

(d) Which types, when placed in the hole marked `__?__` below, cause the compiler to generate an “inconvertible types” error message?
```
T t = new T();
boolean b = t instanceof __?__;
```

a. Object  
  b. S  
  c. T  
  d. Answer: U  
  e. Answer: V

**Grading scheme:** Each subproblem is graded separately out of 3 points. For (a) and (b), +3 for correct answer circled. For (c), +1 for each correct answer circled. For (d), +2 if one correct answer is circled and +3 if both correct answers are circled. Penalty of -1 for each incorrectly circled answer (applied per subproblem so that students cannot lose more points than they earn in the same subproblem).
7. (18 points) The definitions of SimpleCollection and related classes are reproduced, for your reference, on pages 10 to 12. On page 13, you can find the code for the IterDouble class (which we also saw on the review sheet for the second midterm). Suppose we execute the following program:

```java
public class Iter {
    public static void main (String args[]) {
        SimpleCollection<String> c = new LinkedSimpleCollection<String>();
        c.add(null); c.add("a");

        SimpleIterator<String> j = new IterDouble(c.iterator());
        SimpleIterator<String> i = new IterDouble(j);
        String x = i.next(); String y = i.next(); String z = i.next();
        // HERE
    }
}
```

In the space below, draw a diagram of the memory configuration—both stack and heap—when execution reaches the line marked HERE. Omit the parameter args to the main method, and use the same conventions as the sample stack/heap drawing on page 14. Answer:

![Memory Configuration Diagram](image)

Grading scheme: We saw a very diverse range of partially correct solutions to this. Points were assigned based on how closely the solution approximated the correct one.
8. (24 points) This problem again refers to the definitions of `SimpleCollection` and related classes on pages 10 to 12. Your job is to complete the definition of a method `interleave` that combines the contents of two `LinkedSimpleCollections` “in-place,” in alternating order. For example, if `c` contains "a", "c", and "e" (in that order) and `c2` contains "b", and "d" (in that order), then after `c.interleave(c2)`, `c` should contain "a", "b", "c", "d", and "e", and `c2` should be empty. If the two initial collections are not the same size, the extra elements should appear at the end of the interleaved collection. Your solution should not create any new objects (i.e., don’t use `new` anywhere).

Answer:

```java
class LinkedSimpleCollection<E> implements SimpleCollection<E> {
    // ...

    public void interleave(LinkedSimpleCollection<E> other) {
        // ...
        if (other == null) {
            return;
        }

        if (this.first == null) {
            this.first = other.first;
            other.first = null;
            return;
        }

        if (other.first == null) return;

        Node<E> last = this.first;
        Node<E> left = this.first.next;
        Node<E> right = other.first;
        other.first = null;
        boolean takeRight = true;

        while (true) {
            if (left == null) {
                last.next = right;
                return;
            } else if (right == null) {
                last.next = left;
                return;
            }

            if (takeRight) {
                last.next = right;
                right = right.next;
            } else {
                last.next = left;
                left = left.next;
                takeRight = !takeRight;
            }
        }
    }
}
```


Grading scheme: The top level grading was broken into three 6-point sections: the basic pointer setup, the basic loop structure, and the corner cases necessary for correctness.

Two point deductions were made for, e.g., not considering `other == null`. Larger deductions were made for bad logic in the loop.

Solutions with iterators and some logic were typically awarded 4 points.
For Reference:
SimpleCollection, SimpleIterator, and Node

interface SimpleCollection<E> {
    boolean removeAll(E element);
    boolean add(E element);
    boolean contains(Object o);
    SimpleIterator<E> iterator();
}

interface SimpleIterator<E> {
    boolean hasNext();
    E next();
}

class Node<E> {
    public E element;
    public Node<E> next;

    public Node (E element, Node<E> next) {
        this.element = element;
        this.next = next;
    }
}
For Reference: LinkedSimpleCollection class

class LinkedSimpleCollection<E> implements SimpleCollection<E> {
    private Node<E> first = null;

    public boolean removeAll(E element) {
        boolean changed = false;
        while (first != null && first.element.equals(element)) {
            first = first.next;
            changed = true;
        }
        if (first == null) return changed;
        Node<E> prev = first;
        for (Node<E> current = first.next;
             current != null;
             current = current.next) {
            if (current.element.equals(element)) {
                prev.next = current.next;
                changed = true;
            }
            prev = prev.next;
        }
        return changed;
    }

    public boolean add(E element) {
        Node<E> newnode = new Node<E>(element, first);
        first = newnode;
        return true;
    }

    public boolean contains(Object o) {
        for (Node<E> current = first; current != null; current = current.next) {
            if ( (current.element == null && o == null)
                || (current.element != null && current.element.equals(o))) {
                return true;
            }
            return false;
        }

    public SimpleIterator<E> iterator() {
        return new LinkedSimpleIterator<E>(first);
    }
}
For Reference: LinkedSimpleIterator class

class LinkedSimpleIterator<E> implements SimpleIterator<E> {
    private Node<E> current;

    public LinkedSimpleIterator (Node<E> first) {
        current = first;
    }

    public boolean hasNext() {
        return (current != null);
    }

    public E next() {
        if (current == null) {
            throw new java.util.NoSuchElementException();
        } else {
            E n = current.element;
            current = current.next;
            return n;
        }
    }
}

For Reference: IterDouble Class

class IterDouble implements SimpleIterator<String> {
    SimpleIterator<String> it;
    boolean hasCached = false;
    String cached = null;

    IterDouble (SimpleIterator<String> wrapped) { it = wrapped; }

    public boolean hasNext () { return (hasCached || it.hasNext()); }

    public String next () {
        if (hasCached) {
            String s = cached;
            hasCached = false;
            cached = null;
            return s;
        } else {
            cached = it.next();
            hasCached = true;
            return cached;
        }
    }
}

For Reference: Sample Stack/Heap Drawing

Stack
- coll1
- coll2
- c
- e

Heap
- LinkedSimpleCollection
  - first
    - Node
      - element
      - next null
    - String
      - "foo"
  - Node
    - element
    - next
    - String
      - "bar"
- LinkedSimpleCollection
  - first
    - Node
      - element
      - next null