CIS 120 Final Exam   May 5-7, 2021

Name:

PennKey (penn login id):

By entering my name, I certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination. This exam is solely my own work.

- There are 120 total points. **The exam is timed and you have 4 hours to complete the exam once you download it from Gradescope.** The per-problem point values are available to help you budget your time.
- This exam is *open Codio* and *open course resources*. You may access the course videos, the course notes and slides, prior exams, OCaml and Java documentation, recitation materials, and your homework assignments during the exam.
- Do not collaborate with anyone else when completing this exam. Do not share this PDF with anyone else.
- This is an editable PDF document. **Do not use a web-browser embedded PDF viewer**. On OSX, Preview is a good choice. On Windows, Adobe Reader or Adobe Acrobat should work. (e.g., in Chrome or Safari) since refreshing the browser tab will lose your work. In all cases, save your work periodically!
- Submit your exam via Gradescope before the four hour deadline. After you submit, check your PDF to make sure that your answers are legible. Gradescope will allow you to resubmit your PDF as long as you have time remaining.
- For coding problems: aim for accurate syntax, but we will not grade your code style for indentation, spacing, etc. **Do not leave any answers blank.** We will give partial credit.
- We have provided you with a Codio project that you are welcome to use when completing the exam. Do not access this project before your time starts or after you have submitted the exam. Although we strongly encourage you to use Codio, we will only grade the answers in your PDF.
- If you have a question or need clarification, please use a private post on Piazza.
- There are 16 pages in the exam, plus an Appendix for your reference.
- Do not spend too much time on any one question. Be sure to recheck all of your answers.
- Good luck!

1. **Lists and Higher-order Functions** (16 points)

This problem asks you to identify when OCaml function definitions are equivalent. In other words, can you determine when two functions will always return the same results when given the same inputs?

Below, you will find the definitions of four mystery functions, named `a`, `b`, `c` and `d`.

Your job is to match up these mystery definitions with the mystery functions defined in Appendix A named `e`, `f`, `g`, `h`, and `k`. For each function below, there will be *at least one* appendix function that matches, but there may be more than one. Write the names of *all* such functions in the corresponding text box. (Note: there is nothing on Codio available for this problem. You may wish to download an additional copy of the exam so that you can see the Appendix at the same time as this page. But if you do, make sure that all of your answers are in a *single* PDF.)

```
let rec a (x : 'a) (xs : 'a list) : 'a list =
  begin match xs with
    | [] -> []
    | (h :: t) -> x :: h :: a x t
  end
```

Matches for `a`

```
let rec b (x : 'a) (xs : 'a list) : 'a list =
  begin match xs with
    | [] -> []
    | (h :: t) -> h :: x :: b x t
  end
```

Matches for `b`

```
let rec c (x : 'a) (xs : 'a list) : 'a list =
  begin match xs with
    | [] -> []
    | (h :: t) -> h :: c x t
  end
```

Matches for `c`

```
let d (x : 'a) (xs : 'a list) : 'a list =
  let rec loop (zs : 'a list) (w : 'a list) =
    begin match zs with
      | [] -> w
      | (h :: t) -> loop t (x :: h :: w)
    end
  in loop xs []
```

Matches for `d`

2. **Tree structured data** (14 points)

   Recall the type of `LeafyTree`s in OCaml from your first midterm

```
type 'a leafy_tree =
  | Leaf1 of 'a                                  (* single leaf *)
  | Leaf2 of 'a * 'a                             (* double leaf *)
  | Node of 'a leafy_tree * 'a * 'a leafy_tree   (* internal node *)
```

   along with the following function that constructs an *in order traversal* of the tree.

```
let rec inorder (t:'a leafy_tree) : 'a list =
  begin match t with
    | Leaf1 x -> [x]
    | Leaf2(x,y) -> [x;y]
    | Node(lt,x,rt) -> inorder lt @ (x :: inorder rt)
  end
```

   For example, given the following tree

```
let t1 : int leafy_tree = Node (Leaf2 (1, 2), 3, Leaf1 4)
```

   the result of `inorder t1` is the list `[1,2,3,4]`.

   In this problem, you will translate this data structure and function to an analogous implementation in Java.

   As a start, we have given you a definition for the `LeafyTree` interface (below) and the `Leaf1` class (next page).

```
public interface LeafyTree<E> {
    /* Return the elements of the tree in order.
       This method should *never* throw a NullPointerException. */
    List<E> inOrder();
}
```

   After you have defined the `Leaf2` and `Node` classes, the following test case should compile and execute successfully.

```
    @Test
    public void testInOrder() {
        List<Integer> list = new LinkedList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        LeafyTree<Integer> tree =
                    new Node<Integer>(new Leaf2<Integer>(1,2),
                    3,
                    new Leaf1<Integer>(4));
        assertEquals(list,tree.inOrder());
    }
```

   (Code for this problem is available on Codio.)

Here is our definition of a `Leaf1` class, add your version of `Leaf2` below. *Hint*: An excerpt of the Java `List<E>` interface is available in Appendix D.2.

```
public class Leaf1<E> implements LeafyTree<E> {

    private final E val;

    public Leaf1(E v) {
        this.val = v;
    }

    @Override
    public List<E> inOrder() {
        List<E> lst = new LinkedList<E>();
        lst.add(val);
        return lst;
    }
}
```

```
public class Leaf2<E> implements LeafyTree<E> {













}
```

Hint: Note the comment above the declaration of the `inOrder` method in the interface. How can you prevent this method from throwing a `NullPointerException`? Is there an invariant that you can establish in the constructor?

```
public class Node<E> implements LeafyTree<E> {







}
```

3. **Exceptions in Java**

Consider the Java code shown in Appendix B and available in Codio. When run, this code produces the following sequence of letters.

```
1  A
2  B
3  C
4  C
5  B
6  B
7  B
8  C
9  C
10 B
11 B
12 C
13 C
14 A
```

Observe that the first `A` of the output (on line 1) is printed by line 6 of `ExceptionDemo.java`. (You can edit the Codio definition of this file if it would help you solve this problem.)

(a) (3 points) Which line of the program prints the `B` on line 7 of the output?

(b) (3 points) Which line of the program prints the `C` on line 9 of the output?

(c) (3 points) Why does `methodA` need a `throws` annotation? Select all reasons that apply.

- The call to `methodB` on line 7 throws a checked exception.
- The call to `methodB` on Line 8 throws a checked exception.
- The call to `methodC` on Line 9 throws a checked exception.
- It calls `methodB` which has a throws clause.
- It calls `methodC` which has a throws clause.
- It doesn't actually need this annotation.

(d) (3 points) Why does `methodC` need a `throws` annotation? Select all reasons that apply.

- Line 30 throws an `IOException`.
- `IOException` is a subclass of `RuntimeException`.
- It is called by `methodA` which throws a checked exception.
- It is called by `methodA` which has a throws clause.
- It doesn't actually need this annotation.

(e) (2 points) All calls to `methodB` (in this code and in other classes that use this one) will reach line 17.

- True
- False, because

(f) (2 points) All calls to `methodB` (in this code and in other classes that use this one) will reach line 22.

- True
- False, because

(g) (2 points) All calls to `methodB` (in this code and in other classes that use this one) will reach line 24.

- True
- False, because

4. **Data structure trade-offs**

Suppose you are implementing a massively multiplayer game and have already designed a class `Player` to represent each individual player in the game. You know the following to be true about your game design:

- There may be many, many players during the game.
- Players can be added or eliminated at anytime the game is running.
- All players in the game must be distinct. Players are not allowed to join the game if they are already playing it.

Note: there is nothing in the Appendix or Codio for this problem. You must answer the questions based only on the information given here.

(a) (4 points) Which data structure would you pick to keep track of all players in the game? Check one of the three options below.

- `List<Player> players = new LinkedList<Player>();`
- `Set<Player> players = new TreeSet<Player>();`
- `Player[] players = new Player[NUM_PLAYERS];`

Write a short justification of this choice of data structure. Why would you prefer it over the other two options?

(b) (4 points) Now suppose at certain situations during the game a small group of players enters combat. During combat, the players take turns attacking according to an order determined at the beginning of the combat session. If a player is killed or incapacitated, they should be removed from the combat session.

Which data structure would you use to keep track of players during combat sessions? Check one of the three options below.

- `List<Player> combatants = new LinkedList<Player>();`
- `Set<Player> combatants = new TreeSet<Player>();`
- `Player[] combatants = new Player[NUM_PLAYERS];`

Write a short justification of this choice of data structure. Why would you prefer it over the other two options?

(c) (3 points) Suppose that you selected `LinkedList`. When using this class, you observe some strange behavior—the following test fails with an `AssertionError`. (Note: we have **not** given you the definition of the `Player` class, but you can assume that this code compiles. You can also assume that this code refers to the `LinkedList` class from the Java Collections library and the `assertTrue` class from JUnit.)

```
@Test
void testList () {
    List<Player> players = new LinkedList<Player>();
    players.add(new Player("John"));
    players.add(new Player("Delenn"));
    assertTrue(players.contains(new Player("John")));  // fails
}
```

Describe the potential source of this bug in a few sentences. Be specific.

(d) (3 points) Suppose that you selected `TreeSet` instead. However, you then encountered a different `AssertionError`, shown below. (Note: you should make the same assumptions as above and can assume that the move method updates some private state of the `Player` object.)

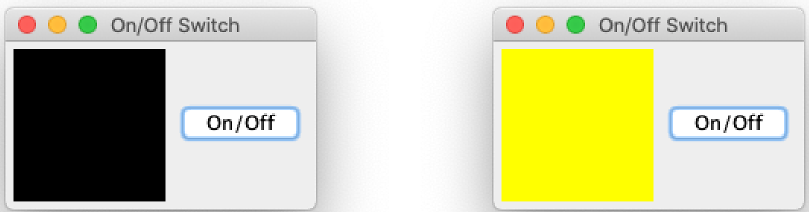
```
@Test
void testSet () {
    Set<Player> players = new TreeSet<Player>();
    Player john = new Player("John");
    Player delenn = new Player("Delenn");
    players.add(john);
    players.add(delenn);
    delenn.move(3);
    assertTrue(players.contains(delenn)); // fails
}
```

Describe the potential source of this bug in a few sentences. Be specific.

5. **Java Swing**

Consider the file `OnOff.java` which was discussed in lecture and is available in Codio and in Appendix C.

When run, this application creates a window showing a black square (the lightbulb) and a button, as shown on the left below. Clicking this button turns the lightbulb yellow (right).



For this question you may wish to refer to Oracle's online documentation for the Java Swing library (i.e. the webpages for package awt and package swing).

(a) (9 points) Put an 'x' next to True or False.

a. True False The class `LightBulb` is a subclass of `JComponent`.

b. True False The `run` method contains five uses of the **`new`** keyword. Exactly three of them create instances of some subclass of `JComponent`.

c. True False The class `ButtonListener` could be replaced by an anonymous inner class.

d. True False Because the method `getPreferredSize` is never called in this file, it can be removed without changing the behavior of the application.

e. True False The variable `JFrame.EXIT_ON_CLOSE` is a static member of the `JFrame` class.

f. True False The `run` method is a static method in the `OnOff` class.

g. True False The class `OnOff` does not have a superclass.

h. True False The type `ActionEvent` is a subtype of `ActionListener`.

i. True False The dynamic class of the variable `frame` defined in the first line of the `run` method is `JFrame`.

(b) (3 points) Note that none of the `OnOff` code directly invokes the `paintComponent` method of the `LightBulb` class. Which of the following explanations best describes why that is unnecessary? (Put an 'x' next to only one option below.)

- The `@Override` directive causes the `Lightbulb` class to overwrite the `JComponent`'s class table entry for `paintComponent` with its own code.
- The `Lightbulb` `paintComponent` method is called as a static proxy for a `JComponent` reference delegate object.
- The `Lightbulb` `paintComponent` method is called via dynamic dispatch from somewhere inside the Swing library, where the bulb object is treated as a `JComponent`.
- The `Lightbulb` `paintComponent` method is invoked by using `instanceOf` and a type cast operation from somewhere inside the Swing library.

(c) (7 points)

What code could you add to the `run` method to make the lightbulb turn on and off when the mouse is clicked within the lightbulb itself? Clicking on the gray area around the button should have no effect. Your answer should be a few lines of Java code.

Code should be inserted after line:

6. **Java Design Problem**

Next, you will use the design process to implement a Java class called `SnoopingIterator`. You will need to first read through Steps 1 and 2 on this page and then answer questions to complete Steps 3 and 4.

Step 1: Understand the problem Recall that an iterator is an object that yields a sequence of elements. However, one issue with the iterator interface is that there is no way to undo a usage of `next`—a single element can only be produced once by an iterator.

A `SnoopingIterator` solves this issue by providing an additional operation, called `snoop`, that allows one to look ahead in the iteration without advancing the iterator. For example, suppose `it` is a `SnoopingIterator` for a list containing the the integers 0 and 1, in that order. Then a sample usage of this iterator might look like this:

```
System.out.println(it.snoop());      // prints "0"
System.out.println(it.next());       // prints "0"
System.out.println(it.snoop());      // prints "1"
System.out.println(it.snoop());      // prints "1"
System.out.println(it.hasNext());    // prints "true"
System.out.println(it.next());       // prints "1"
```

Step 2: Design the interfaces The Javadocs for the `Iterator<E>` interface are given in Appendix D.1. In this problem you will develop a generic `SnoopingIterator` class that implements the `Iterator` interface.

The constructor of this class should take another iterator as an argument and add "snooping", i.e. the ability to spy ahead to the next value without advancing the iterator. That means that the constructor should have the following declaration.

```
/**
 * Constructor.
 *
 * @param i the underlying iterator
 * @throws IllegalArgumentException when i is null.
 */
public SnoopingIterator(Iterator<E> i)
```

(There are no questions on this page.)

The `snoop` operation should return the same element that would be returned by `next`, but should not advance the iterator. If there there are no more elements remaining in the underlying iterator, then `snoop` should throw a `NoSuchElementException`.

```
/**
 * Returns the next element without advancing the iterator.
 * This element may be null if the next element would be null.
 *
 * @return the next element from the iterator
 * @throws NoSuchElementException if there are no more elements
 */
public E snoop()
```

Note that no method in the `SnoopingIterator` class should throw a `NullPointerException`. If the underlying iterator would produce a null as the `next` element, then the `snoop` method should return `null`.

To simplify this problem, the `SnoopingIterator` class does not need to to support the `remove` operation even if that method is available in the underlying iterator.

Step 3: Write test cases

(a) (10 points)

Suppose you are given the following `SnoopingIterator`, called `s` below.

```
List<Integer> list = new LinkedList<Integer>();
list.add(1);
list.add(null);
list.add(2);
SnoopingIterator<Integer> s =
    new SnoopingIterator<Integer>(list.iterator());
```

What unit tests could you write with `s`?

Below, describe **in words** a suite of nonoverlapping tests. You may assume that each test starts with a fresh definition of `s`. Your description of each test must be specific, describing either the outputs of methods from the `SnoopingIterator` class or any exceptions that could be thrown.

We've given you one example to get started. You need to add **five** more. You will be graded on the correctness and comprehensiveness of your tests.

1. *two successive calls of* `s.snoop()` *both return 1*

2.

3.

4.

5.

6.

(b) (4 points) Choose one of your tests above (tell us which one!) and complete the implementation in the box below. (We have provided the test case associated with our example in Codio.)

```
@Test
public void test(){
    List<Integer> list = new LinkedList<Integer>();
    list.add(1);
    list.add(null);
    list.add(2);
    SnoopingIterator<Integer> s =
        new SnoopingIterator<Integer>(list.iterator());









}
```

This code is for test number:

Step 4: Implementation (25 points)

Complete the implementation of the `SnoopingIterator` class. We suggest that you implement this operation in Codio and then cut and paste your answers into the exam.

You may add any private instances variables and helper methods that you need to your implementation.

Note: Here (and throughout the exam) you may assume that appropriate import statements bring `Iterator` and `NoSuchElementException` into scope; we omit them to save space. **You may not use any additional classes or libraries, nor modify any import statements to your solution.**

```
public class SnoopingIterator<E> implements Iterator<E> {
   private final Iterator<E> it;
   // TODO: Additional fields, if needed




   public SnoopingIterator(Iterator<E> i) {
     if (i == null) { throw new IllegalArgumentException();  }
     this.it = i;
     // TODO: Additional code for constructor, if needed.




   }
   // TODO: Helper method, if needed.
```

```
@Override public boolean hasNext() {









}
@Override public E next() {









}
public E snoop() {









}
```

NOTE: All **Java** code in this appendix is also available on Codio.

A Appendix: OCaml list functions

```
(***** Transform and fold *******)

let rec transform (f : 'a -> 'b) (xs : 'a list) : 'b list =
  begin match xs with
    | [] -> []
    | (h :: t) -> f h :: transform f t
  end

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list) : 'b =
  begin match l with
    | [] -> base
    | hd :: tl -> combine hd (fold combine base tl)
  end


(***** Mystery functions *******)

let e (x : 'a) (xs : 'a list) : 'a list =
  fold (fun h acc -> (h :: x :: acc)) [] xs

let f (x : 'a) (xs : 'a list) : 'a list =
  transform (fun h -> h) xs

let g (x : 'a) (xs:'a list) : 'a list =
  fold (fun h acc -> (x :: h :: acc)) [] xs

let h (x : 'a) (xs:'a list) : 'a list = xs

let rec k (x : 'a) (xs : 'a list) : 'a list =
  begin match xs with
    | [] -> []
    | (h :: t) -> k x t @ [x; h]
  end
```

B Appendix: ExceptionDemo.java

```java
import java.io.IOException;

class ExceptionDemo {

    public void methodA() throws IOException {
        System.out.println("A");
        methodB(false);
        methodB(true);
        methodC(false);
        System.out.println("A");
    }

    public void methodB(boolean b) {
        System.out.println("B");
        try {
            methodC(b);
            System.out.println("B");
            return;
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            System.out.println("B");
        }
        System.out.println("B");
    }

    public void methodC(boolean raiseException) throws IOException {
        System.out.println("C");
        if (raiseException) {
            throw new IOException("C");
        }
        System.out.println("C");
    }

    public static void main(String[] args) {
        try {
            new ExceptionDemo().methodA();
        } catch (Exception e) {
            System.out.println("A");
        }
    }

}
```

C Appendix: Java OnOff example

```java
/* A Swing version of the Light switch GUI program */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

@SuppressWarnings("serial")
class LightBulb extends JComponent {

    private boolean isOn = false;

    public void flip() {
        isOn = !isOn;
    }

    @Override
    public void paintComponent(Graphics gc) {
        // display the light bulb here
        if (isOn) {
            gc.setColor(Color.YELLOW);
        } else {
            gc.setColor(Color.BLACK);
        }
        gc.fillRect(0, 0, 100, 100);
    }

   @Override
   public Dimension getPreferredSize() {
        return new Dimension(100, 100);
   }
}

class ButtonListener implements ActionListener {
    private LightBulb bulb;

    public ButtonListener(LightBulb b) {
        bulb = b;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        bulb.flip();
        bulb.repaint();
    }
}

public class OnOff implements Runnable {
    public void run() {
        JFrame frame = new JFrame("On/Off Switch");
```

```java

        // Create a panel to store the two components
        // and make this panel the contentPane of the frame
        JPanel panel = new JPanel();
        frame.getContentPane().add(panel);

        LightBulb bulb = new LightBulb();
        panel.add(bulb);

        JButton button = new JButton("On/Off");
        panel.add(button);

        button.addActionListener(new ButtonListener(bulb));

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new OnOff());
    }
}
```

D Appendix: Java Collections Interfaces and Classes

D.1 Java Iterator interface

```
interface Iterator<E>

boolean hasNext()
         // Returns true if the iteration has more elements. (In other words,
         // returns true if next() would return an element rather than
         // throwing an exception.)


E       next()
         // Returns the next element in the iteration.
         // throws NoSuchElementException if the iteration has no more elements

void    remove()
         // Removes from the underlying collection the last element
         // returned by this iterator (optional operation).
```

D.2 Java List interface

```
interface List<E> extends Collection<E>

boolean add(E e)
         // Appends the specified element to the end of this list.

boolean addAll(Collection<? extends E> c)
         // Appends all of the elements in the specified collection to
         // the end of this list, in the order that they are returned
         // by the specified collection's iterator.

void    clear()
         // Removes all of the elements from this list.

boolean contains(Object o)
         // Returns true if this list contains the specified element.

Iterator<E>     iterator()
         // Returns an iterator over the elements in this list in
         // proper sequence.
```