CIS 120 Final Exam May 9, 2022

Name (printed): ______________________________

Pennkey (letters, not numbers): ______________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: ______________________________ Date: ____________________

- There are 120 total points. The exam period is 120 minutes long.
- Please skim the entire exam first—some of the questions will take significantly longer than others.
- There are 14 pages in this exam.
- Do not collaborate with anyone else when completing this exam.
- Please write your name and Pennkey (e.g., `sweirich`) on the bottom of *every other* page where indicated.
- There is a separate appendix for reference. Answers written in the appendix will not be graded.
- Good luck!

1. **OCaml Higher-Order functions** (12 points)

Recall the higher-order list processing functions transform and fold (reproduced in Appendix A). Each part of this problem below begins with a sample function written using simple recursion over lists, followed by several alternative versions written using fold. In each part, first indicate what the function returns for the sample input shown. Then mark all of the alternatives that implement the same behavior as the recursive sample. **There may be zero, one, or more than one such function**. Some of the alternatives do not typecheck—do not mark these.

(a)
```
let rec func1 (x:'a) (lst: 'a list) : int =
  begin match lst with
  | [] -> 0
  | hd :: tl -> if x = hd then 0 else 1 + func1 x tl
  end
let ans1 = func1 3 [0; 1; 2; 3]
```

What is the result? `ans1 =` ____________________________

Which of the following functions behave the same as `func1` on all inputs? (Check all that apply.)

☐
```
let func1 (x: 'a) (lst: 'a list) : int =
  fold (fun hd acc -> if (x = hd) then 0 else 1 + acc) 0 lst
```

☐
```
let func1 (x: 'a) (lst: 'a list) : int =
  fold (fun hd acc -> 1 + acc) 0 lst
```

☐
```
let func1 (x: 'a) (lst: 'a list) : int =
  fold (fun y acc -> if y then 0 else 1 + acc)
  0
  (transform (fun hd -> x = hd) lst)
```

(b)
```
let func2 (lst: 'a list): 'a list =
  let rec loop (res: 'a list) (acc: 'a list) =
      begin match res with
      | [] -> acc
      | hd :: tl -> loop  tl (hd :: acc)
      end in
    loop lst []

let ans2 = func2 [0; 1; 2; 3]
```

What is the result? `ans2 =` ____________________________

Which of the following functions behave the same as `func2` on all inputs? Recall that `@` appends two lists in OCaml. (Check all that apply.)

☐
```
let func2 (lst: 'a list): 'a list =
  fold (fun x acc -> x :: acc) [] lst
```

☐
```
let func2 (lst: 'a list) : 'a list =
  fold (fun x acc -> acc @ [x]) [] lst
```

☐
```
let func2 (lst: 'a list): 'a list =
  fold (fun x xs -> x @ xs) [] lst
```

2. **OCaml queues and Java Linked Lists** (13 points)

Consider the following OCaml functions that work with the queue data structure shown in Appendix B. The Java `LinkedList` class also implements a mutable, linked data structure. For each OCaml function below, use the documentation in Appendix C to determine which method of the `LinkedList` class provides the most similar functionality, or write none if there is no corresponding method in this class. This question tests your understanding of OCaml, so each function is called `f`.

For some of these operations, you may also be asked to indicate whether the function `f` is tail recursive.

(a)
```
let f (q: 'a queue) : bool =
   q.head = None
```

Most similar `LinkedList` method: ___________________________

(b)
```
let f (q: 'a queue) : unit =
   q.head <- None; q.tail <- None
```

Most similar `LinkedList` method: ___________________________

(c)
```
let f (q: 'a queue) : int =
    let rec loop (no: 'a qnode option) : int =
      begin match no with
      | None -> 0
      | Some n -> 1 + loop n.next
      end
    in loop q.head
```

Most similar `LinkedList` method: ___________________________

Is this function tail recursive? ☐ Yes ☐ No

(d)
```
let f (q: 'a queue) (elt:'a) : int =
    let rec loop (no: 'a qnode option) (i:int) : int =
      begin match no with
      | None -> -1
      | Some n -> if n.v = elt then i else loop n.next (i+1)
      end
    in loop q.head 0
```

Most similar `LinkedList` method: ___________________________

Is this function tail recursive? ☐ Yes ☐ No

(e)
```
let f (q: 'a queue) : 'a queue =
    let rec loop (no: 'a qnode option) (q2 : 'a queue) : 'a queue =
      begin match no with
      | None -> q2
      | Some n -> enq q2 n.v ; loop n.next q2
      end
    in let q2 = { head = None; tail = None}
    in loop q.head q2
```

Most similar `LinkedList` method: ___________________________

Is this function tail recursive? ☐ Yes ☐ No

3. **TreeSets in Java** (10 points)

The Java `TreeSet` class is implemented using a Binary Search Tree (BST). This class maintains the Binary Search Tree Invariant by storing the entries in the tree in order. Based on your understanding of BSTs in OCaml, which of the following methods of this class make use of this invariant?

(a) `int size()`
Returns the number of elements in this set (its cardinality).

Uses BST invariant: ☐ Yes ☐ No

(b) `boolean contains(Object value)`
Returns true if this set contains the specified element.

Uses BST invariant: ☐ Yes ☐ No

(c) `K floor(E e)`
Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.

Uses BST invariant: ☐ Yes ☐ No

(d) `boolean isEmpty()`
Returns true if this set contains no elements.

Uses BST invariant: ☐ Yes ☐ No

(e) `Iterator<E> iterator()`
Returns an iterator over the elements in this set in ascending order.

Uses BST invariant: ☐ Yes ☐ No

Connect Four

The remaining exam problems refer to a partial implementation of the game shown in Appendices D, E, F, and G and summarized below.

The *Connect Four* game features two players (`WHITE` and `BLACK`) taking turns adding pieces to a game board. In the physical version of the game, the game board is held upright, so players can only add their piece to the lowest empty spot in a given column. A player wins if they line up four of their pieces in a straight line, either vertically or horizontally. (For simplicity, this version does not look at diagonals.) The `WHITE` player goes first.

A sample game in progress is shown in the figure below on the left. By selecting the third column, the `BLACK` player can win by stacking four of their pieces in this column, as shown in the figure on the right.

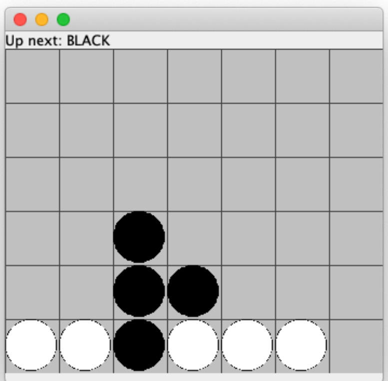

Figure 1: *Connect Four* game in progress. It is `BLACK`'s turn.

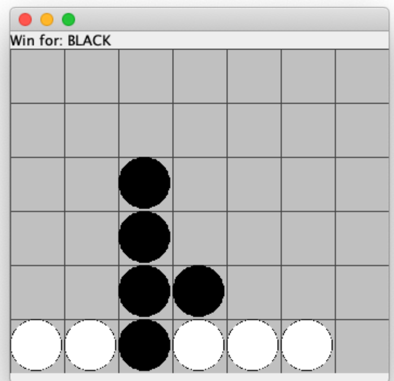

Figure 2: `BLACK` plays in the third column to win.

Take a moment now to familiarize yourself with the code in the appendices. Appendix D defines classes that represent the two players. The code shown in Appendices E-G is all part of the same class, called `ConnectFour`, with the structure shown below.

```
class ConnectFour {
   public static final int ROWS = 6;
   public static final int COLUMNS = 7;

   // Appendix E: instance variables and methods for game logic
   // Appendix F: inner class View
   // Appendix G: instance variables for GUI, inner class Mouse,
   //             and ConnectFour constructor

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

(There is nothing to answer on this page.)

4. **Understanding OO Class Definitions** (15 points)

The following questions refer to the classes defined in Appendix D that represent the two players in the *Connect Four* game and to the `LinkedList` class from the Collections Framework (Appendix C).

Which of the following code blocks is legal Java code that will not cause any compile-time (i.e. type checking) or run-time errors? If it is legal code, check the "Legal Code" box and answer the questions that follow it. If it is not legal, check one of the "Not Legal" options and explain why. You can assume each block below is independent and written in some static method defined in the `ConnectFour` class and that the appropriate imports have been made at the top of the file.

(a) (3 points)

```
Player p = Player.WHITE;
Player n = p.getNext();
boolean ans = (n == p);
```

☐ Legal Code

A. The static type of `p` is __________________________ .

B. The dynamic class of `p` is __________________________ .

C. The value of `ans` is __________________________ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

____

(b) (3 points)

```
Player p = new Player(Color.WHITE);
boolean ans = p.equals(Player.WHITE);
```

☐ Legal Code

A. The static type of `p` is __________________________ .

B. The dynamic class of `p` is __________________________ .

C. The value of `ans` is __________________________ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

____

(c) (3 points)

```
Player p = new WhitePlayer();
boolean ans = p.equals(Player.WHITE);
```

☐ Legal Code

A. The static type of `p` is ______________________________ .

B. The dynamic class of `p` is ______________________________ .

C. The value of `ans` is ______________________________ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

____

(d) (3 points)

```
List<Player> lst = new LinkedList<Player>();
lst.add(Player.WHITE);
boolean ans = lst.contains(Player.WHITE);
```

☐ Legal Code

A. The static type of `lst` is ______________________________ .

B. The dynamic class of `lst` is ______________________________ .

C. The value of `ans` is ______________________________ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

____

(e) (3 points)

```
List<WhitePlayer> lst = new LinkedList<WhitePlayer>();
lst.add(Player.BLACK);
boolean ans = lst.contains(Player.BLACK);
```

☐ Legal Code

A. The static type of `lst` is ______________________________ .

B. The dynamic class of `lst` is ______________________________ .

C. The value of `ans` is ______________________________ .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

____

5. **Invariants** (10 points)

The state of the connect four game is stored by the following instance variables in the class `ConnectFour`.

```
private Player[][] board;  // the game board
private Player player;     // current player
private Player winner;     // null if no player has yet won
```

These instance variables are initialized by the `start` method, shown below.

```
public void start() {
    board = new Player[ROWS][COLUMNS];
    player = Player.WHITE;
    winner = null;
}
```

The `board` is represented by a two-dimensional array that stores the locations of the players' pieces, indexed by their row number and column number. The current `player` is either `Player.WHITE` or `Player.BLACK`. We also keep track of whether either player is the `winner` of the game.

For example, in the state shown in Figure 1 on page 5, `player` is equal to `Player.BLACK`, the game is still in progress so `winner` is **`null`**, and `board` is equal to the 2D array shown below. (For conciseness we write `Player.WHITE` as `WHITE` and `Player.BLACK` as `BLACK`).

```
new Player[][] {
        {null,  null,  null,  null,  null,  null,  null},
        {null,  null,  null,  null,  null,  null,  null},
        {null,  null,  null,  null,  null,  null,  null},
        {null,  null,  BLACK, null,  null,  null,  null},
        {null,  null,  BLACK, BLACK, null,  null,  null},
        {WHITE, WHITE, BLACK, WHITE, WHITE, WHITE, null}};
```

Locations on the board are referenced by `board[row][column]`. For example, position `board[5][0]` contains the `WHITE` player's piece in the last (lowest) row and first (leftmost) column.

Which of the properties below would make good invariants for the `ConnectFour` class? (*i.e.,* which properties should be true after the instance variables have been initialized and throughout the execution of the application?)

(a) True ☐ False ☐ `board` is never **`null`**

(b) True ☐ False ☐ `winner` is never **`null`**

(c) True ☐ False ☐ `board[row]` is never **`null`** for any `0 <= row < ROWS`.

(d) True ☐ False ☐ `board[row][col]` is never **`null`** for any `0 <= col < COLUMNS` and `0 <= row < ROWS`.

(e) True ☐ False ☐ If `board[row][col]` is not **`null`**, then for any `k` such that `row < k < ROWS`, `board[k][col]` is not **`null`**.

6. **2D array Programming** (15 points)

Complete the `addPiece` method of the `ConnectFour` class. This method should update the `board` with a new piece for the current `player` in the specified column. For example, if called when the `board` is equal to the sample board shown on page 8, and when the current player is `Player.BLACK`, this method should update `board[2][2]` to `Player.BLACK`.

The method should return whether the new piece was successfully added. In the case that the specified column is full, the method should return **`false`**. This method should only modify `board`—it should not update `player` or `winner`.

```
public boolean addPiece (int col) {
```

for (int

```
}
```

Now think about the invariants that you checked "True" in the previous problem. Select **one** of these invariants and describe how, if it did not hold, your implementation of `addPiece` could behave incorrectly.

Selected invariant: (a) ☐ (b) ☐ (c) ☐ (d) ☐ (e) ☐

What could go wrong in `addPiece` if this invariant does not hold:

7. **Iterators** (15 points)

To check whether a player has won, we use the following methods to search for four pieces in a row either horizontally (rows) or vertically (columns).

```
// check all arrays produced by the iterator for win by player p
public static boolean checkAll(Iterator<Player[]> it, Player p) {
    while (it.hasNext()) {
        Player[] arr = it.next();
        if (checkFourInARow(arr, p)) {
            return true;
        }
    }
    return false;
}

// Check whether the current player has won the game
public boolean checkWin() {
    if (checkAll(new RowIterator(), player)) { return true; }
    return checkAll(new ColumnIterator(), player);
}
```

In the `checkWin` method, the classes `RowIterator` and `ColumnIterator` each implement the interface `Iterator<Player[]>`, providing access to a sequence of arrays representing the individual rows and columns respectively. For example, if the board is equal to the sample board on page 8, and the variable `cit` is a newly-created instance of the `ColumnIterator` class, then the following JUnit test will pass:

```
Player[] column1and2 = {null, null, null, null, null, WHITE};
Player[] column3 = {null, null, null, BLACK, BLACK, BLACK};

assertArrayEquals(cit.next(), column1and2);
assertArrayEquals(cit.next(), column1and2);
assertArrayEquals(cit.next(), column3);
```

For this problem you will complete the `ColumnIterator` class, an inner class of `ConnectFour`, on the next page. We have already declared and initialized the instance variable `currColumn`. You must complete the `hasNext` and `next` methods and their behavior must match the description given by the documentation. You may not define any additional class members (instance variables, constructors, or methods). HINT: because this class is an inner class it has access to the `board` instance variable of `ConnectFour`.

(Complete the code on the next page. There is nothing to answer on this page.)

```
public class ColumnIterator implements Iterator<Player[]> {

   private int currColumn = 0;

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

   }
   /** Returns the next element in the iteration.
    *   @throws NoSuchElementException if the iteration has no more elements
    */
   @Override public Player[] next() {

   }
```

8. **Java True/False** (15 points)

The following questions refer to the `ConnectFour` code shown in Appendix E.

(a) True ☐ False ☐ Line 58 is an example of the use of parametric polymorphism (i.e. generics).

(b) True ☐ False ☐ Line 85 is an example of the use of dynamic dispatch.

(c) True ☐ False ☐ Line 87 is an example of the use of dynamic dispatch.

(d) True ☐ False ☐ Line 96 is an example of the use of subtype polymorphism.

(e) True ☐ False ☐ Line 96 is an example of the use of parametric polymorphism (i.e. generics).

The following questions refer to the `ConnectFour` code shown in Appendix F.

(f) True ☐ False ☐ The type `View` is a subtype of `Object`.

(g) True ☐ False ☐ The class `View` is a subclass of `ConnectFour`.

(h) True ☐ False ☐ The methods and constructors in class `View` may refer to the private instance variable `color` of class `Player` (Appendix D).

(i) True ☐ False ☐ The call to `super.paintComponent` on line 117 refers to a member of class `JPanel`.

The following questions refer to the `ConnectFour` code shown in Appendix G and to the classes of the Java Swing library. For reference, documentation for the Swing library appears in Appendix H.

(j) True ☐ False ☐ The class `Mouse` is a subclass of `JPanel`.

(k) True ☐ False ☐ The type `Mouse` is a subtype of `MouseListener`.

(l) True ☐ False ☐ The type `MouseListener` is a subtype of `Object`.

(m) True ☐ False ☐ The class `Mouse` inherits a method called `mousePressed`.

(n) True ☐ False ☐ The methods and constructors in class `Mouse` may refer to the private instance variables of class `ConnectFour`.

(o) True ☐ False ☐ We can remove line 149 in Appendix G and the application would still work.

9. **Exceptions and ActionListeners** (15 points)

Suppose we would like to add a "save" button to the *Connect Four* game. Pressing this button should record the current state of the game to a save file using the following method to be added in class `ConnectFour` (the implementation of this method is not shown). If the game cannot be saved, this method throws an `IOException`. In this case, the application should use the `info` label to display an error message to the user.

```
// write the game state to a file
// throws IOException if the file cannot be written to
public void saveGame() throws IOException {
  // not shown
}
```

To implement the save button, we'll add the following lines to Appendix G at at line 178.

```
JButton save = new JButton("save");
panel.add(save, BorderLayout.PAGE_END);
```

What should come next? Which of the following code blocks correctly add the action listener for the save button. Check **Yes** if the code would work, or **No** if it either would not compile or would not implement the desired behavior. There may be zero, one or more correct code blocks.

(a) ☐ Yes ☐ No

```
save.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            saveGame();
        } catch (IOException io) {
            info.setText("Cannot save game");
        }
    }
});
```

(b) ☐ Yes ☐ No

```
save.addActionListener(
  try {
      new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) throws IOException {
          saveGame();
        }
      }
  } catch (IOException io) {
      info.setText("Cannot save game.");
  });
```

(c) ☐ Yes ☐ No

```
save.addActionListener((ActionEvent e) -> {
    try {
        saveGame();
    } catch (IOException io) {
        info.setText("Cannot save game");
    }
});
```

(d) ☐ Yes ☐ No

```
try {
    save.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            saveGame();
        }
    });
} catch (IOException io) {
     info.setText("Cannot save game.");
}
```

(e) ☐ Yes ☐ No

```
save.addActionListener((ActionEvent e) -> {
    saveGame();
    throw new IOException("Cannot save game.");
});
```