CIS 120 Final Exam 20 December 2018

Name (printed): ______________________________

Pennkey (login id): ______________________________

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

Signature: ____ Date: ___________________________

- Do not begin the exam until you are asked to do so.
- Make sure your name and Pennkey (a.k.a. username) is on the top of this page, and that your PennKey is on the front side of every page.
- There are 120 total points. You have 120 minutes to complete the exam.
- There are 16 pages in the exam.
- There is a separate Appendix for your reference. Answers written in the Appendix will not be graded.

1. **OCaml: Binary Search Trees and Higher Order Functions (19 points)**

Recall the definitions of generic `transform` and `fold` functions for lists and the type of the generic binary search trees, which are all given in Appendix **A**.

a. (5 points) Circle all trees below that **satisfy** the binary search tree invariant.

```
   (a)         (b)         (c)         (d)         (e)

    3           2           3           3           1
   /           / \         / \         / \
  1           1   2       2   5       2   4
 /                         \         /     \
0                           6       1       8
```

We'll use the tree shown below for the remaining questions.

```
t =      5
        / \
       3   8
          / \
         7   10
```

b. (3 points) Consider the following modification to the `transform` function that now takes in a tree as input.

```
let rec transform_tree (f: 'a -> 'b) (t: 'a tree) : 'b tree =
  begin match t with
  | Empty -> Empty
  | Node(lt, x, rt) -> Node(transform_tree f lt, f x, transform_tree f rt)
  end
```

If the tree `t` is provided as input to the code shown below, what will be the resulting output tree `t1`? Draw it below.

```
transform_tree (fun x -> x * x) t

  t1 =
```

c. (3 points) Does the `transform_tree` function preserve BST invariants? That is, if the input to the function is a BST and *any* valid function `f`, will it *always* return a valid BST as output? (Choose one.)

☐ Yes. If you chose yes, explain why.

☐ No. If you chose no, provide an example with a specific valid BST and a function `f` as input.

Consider the following modification to the `fold` function that now takes in a tree as input.

```
let rec fold_tree (combine: 'a -> 'b -> 'b -> 'b) (base: 'b) (t: 'a tree) : 'b =
  begin match t with
  | Empty -> base
  | Node(lt, x, rt) ->
        combine x (fold_tree combine base lt) (fold_tree combine base rt)
  end
```

d. (2 points) What does the following code do? (Choose one.)

```
fold_tree (fun x lacc racc -> 1 + lacc + racc) 0 t
```

- ☐ Sum up the values of all the elements in the tree
- ☐ Count the number of elements in the tree
- ☐ Calculate the height of the tree
- ☐ The code is well-typed, but will produce some other answer than the ones shown above
- ☐ The code will compile, but exhaust stack space when run since it's not tail recursive
- ☐ It is ill-typed

e. (3 points) What does the following code do? (Choose one.)

```
fold_tree (fun x lacc racc -> 1 + (max lacc racc)) 0 t
```

- ☐ Sum up the values of all the elements in the tree
- ☐ Count the number of elements in the tree
- ☐ Calculate the height of the tree
- ☐ The code is well-typed, but will produce some other answer than the ones shown above
- ☐ The code will compile, but exhaust stack space when run since it's not tail recursive
- ☐ It is ill-typed

f. (3 points) What does the following code return? (Choose one.)

```
fold_tree (fun x lt rt -> lt @ [x] @ rt) [] t
```

- ☐ `[3; 5; 7; 8; 10]`
- ☐ `[5; 3; 8; 7; 10]`
- ☐ `[3; 7; 10; 8; 5]`
- ☐ The code is well-typed, but will produce some other answer than the ones shown above
- ☐ The code will compile, but exhaust stack space when run since it's not tail recursive
- ☐ It is ill-typed

2. **Java Typing and Dynamic Dispatch (27 points)**

This problem refers to an interface and several classes that might be part of a program for managing fantastic beasts. You can find them in Appendix **B**.

Which of the following lines 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. (3 points each)

a. `Animal newt = new Animal();`

- ☐ Legal Code
  - **A.** The static type of `newt` is ____________________________.
  - **B.** The dynamic class of `newt` 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.
```
Animal harryPotterEater = new Dragon();
harryPotterEater.eatSomething();
```

- ☐ Legal Code
  - **A.** The static type of `harryPotterEater` is ____________________________.
  - **B.** The dynamic class of `harryPotterEater` 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.
```
MythicalAnimal t = new Thestral();
t.eatSomething();
```

- ☐ Legal Code

  The code above will print (Choose all that apply.)
  - ☐ "A mythical animal can eat some imaginary food"
  - ☐ "Thestral just took a bite out of Hagrid"
  - ☐ This method is abstract and not implemented yet
- ☐ 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. `RealAnimal wolf = new Direwolf();`
`wolf.printSiblingNames();`

☐ Legal Code
  A. The static type of `wolf` is ____________________________.
  B. The dynamic class of `wolf` 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. `Direwolf direwolf1 = (Direwolf) t;` *// where t is as defined above*

☐ Legal Code
  A. The static type of `direwolf1` is ____________________________.
  B. The dynamic class of `direwolf1` 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):

____

f. `Direwolf direwolf2 = (Direwolf) wolf;` *// where wolf is as defined above*
`direwolf2.printSiblingNames();`

☐ Legal Code
  A. The static type of `direwolf2` is ____________________________.
  B. The dynamic class of `direwolf2` 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):

____

g. `Dragon drogon = (Dragon) t;` *// where t is as defined above*
`drogon.eatSomething();`

☐ Legal Code
  The code above will print (Choose all that apply.)
  ☐ "A mythical animal can eat some imaginary food"
  ☐ "Thestral just took a bite out of Hagrid"
  ☐ This method is abstract and not implemented yet
☐ 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):

____

h. _______ `direwolf3 = new Direwolf();`

Which types (there may be one or more) can be correctly used for the declaration of `direwolf3` above?

☐ `Animal` ☐ `Thestral` ☐ `MythicalAnimal`

☐ `Dragon` ☐ `Direwolf` ☐ `RealAnimal`

i.
```
public void printNames(List<Animal> animals) {
   for (Animal a : animals) {
      System.out.println(a.getName());
   }
}

List<MythicalAnimal> animals = new LinkedList<MythicalAnimal>();
animals.add(new Thestral());
animals.add(new Dragon());
printNames(animals);
```

☐ Legal Code

The code above will print (Choose all that apply.)

☐ “HagridsBestFriend”

☐ “HungarianHorntail”

☐ This method is abstract and not implemented yet

☐ 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):

____

3. OCaml & Java Immutability (15 points)

Recall that variables are implicitly *immutable* in OCaml and are implicitly *mutable* in Java. In this portion of the assignment we will implement an *immutable* collection in Java. Consider the following list interface in OCaml:

```
module List: sig
  (** cons x xs is x :: xs **)
  val cons : 'a -> 'a list -> 'a list

  (** Concatenate two lists . Same as the infix operator @. **)
  val append : 'a list -> 'a list -> 'a list

  (** List reversal **)
  val rev : 'a list -> 'a list
end
```

We will translate this OCaml interface into Java. Complete the following implementation of `ImmutableList` that uses an `ArrayList` to store elements. The following `ImmutableList` methods are already implemented for you and you can use them (and no other methods) if you like.

- `add(E e)` (appends the specified element to the end of this list)
- `get(int index)` (returns the element at the specified position in this list)
- `size()` (returns the number of elements in this list)

```
public class ImmutableList<E> {
   private ArrayList<E> list;

   /**
    * create a new ImmutableList
    */
   public ImmutableList() {
      list = new ArrayList<E>();
   }

   /**
    * create a new ImmutableList with the
    * specified initial capacity
    * @param capacity the initial capacity of the list
    */
   public ImmutableList(int capacity) {
      list = new ArrayList<E>(capacity);
   }
```

a. (5 points) Implement `cons` so it has the same behavior as in OCaml.

```
public ImmutableList<E> cons(E e, ImmutableList<E> list) {



}
```

b. (5 points) Implement `append` so it has the same behavior as in OCaml.

```
public ImmutableList<E> append(ImmutableList<E> l1, ImmutableList<E> l2) {








}
```

c. (5 points) Implement `reverse` so it has the same behavior as in OCaml.

```
public ImmutableList<E> reverse(ImmutableList<E> l) {








}
```

4. Java Exceptions (14 points)

Below is a partial implementation of the Set abstract data type. This implementation uses an ArrayList to store items. Note that the focus here is on exceptions in Java and not on the List and Set semantics. Questions in this section are independent from each other.

```
public class ArrayListSet<E> {
   //ArrayList that will hold items
   private ArrayList<E> list;

   //create a new ArrayList set
   public ArrayListSet() {
      list = new ArrayList<E>();
   }

   //Insert key into the set if it was not previously added
   public boolean add(E key) {

      if(!list.contains(key)) {
         return list.add(key);
      }
      return false;
   }

   //add method that will raise better exceptions
   public boolean improvedAdd(E key) {
      return add(key);
   }
}
```

a. (4 points) We will now modify `add()` so that it raises an `IllegalArgumentException` if `key` is already in the `list`. Only provide the additional lines of code.

____

____

____

}

b. (4 points) Because `IllegalArgumentException` is not very expressive, we decided to implement a new Exception type named `DuplicateItemException`. Modify `improvedAdd` so that it handles the `IllegalArgumentException` raised by `add`, and raises a `DuplicateItemException` instead.

____

____

____

}

c. (6 points) Consider the following method. Note that `read` may throw an `IOException`.

```
public static String processReader(Reader reader) {
    String s = "";
    int c = reader.read();
    while (c != -1) {
        s += c;
        c = reader.read();
    }
    return s;
  }
}
```

Which of the following options is correct? (Choose one.)

- ☐ Legal Code
- ☐ 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):

____

If you think the code is not legal, add or modify the necessary lines in the code below to make it legal.

```
public static String processReader(Reader reader)

{

    String s = "";

    int c = reader.read();

    while (c != -1) {

        s += c;

        c = reader.read();

    }

    return s;

  }

}
```

5. Java implementation and Iterator programming (35 points)

We will implement a data structure called a `MaxHeap`. Note that this is not the same as the heap from the ASM. The `MaxHeap` data structure has the following invariants: it is a complete binary tree with the requirement that every node has a value greater than its children (partial order). A complete binary tree is a binary tree where the nodes are filled in row by row, with the bottom row filled in left to right.

Below are examples of valid `MaxHeaps` except (h4):

```
        (h1)              (h2)              (h3)            (h4)

         10                7                10               10
        /  \              /  \             /  \             /  \
      8      6          5     2          9      6         7     5
     /                 / \              / \    / \       / \     \
    3                 4   1            1   2 4   5      8   4     1
```

```
(h4) is not valid because 8 is greater that its parent 7,
and the bottom row is not filled left to right
(the left child of 5 is empty).
```

Since storing the records in an array in row order leads to a simple mapping from a node's position in the array to its parent, siblings, and children, the array representation is most commonly used to implement the complete binary tree.

Below is the above `MaxHeaps` array representations.

(h1)

index	0	1	2	3
value	10	8	6	3

(h2)

index	0	1	2	3	4
value	7	5	2	4	1

(h3)

index	0	1	2	3	4	5	6
value	10	9	6	1	2	4	5

A partial implementation of the `MaxHeap` is available in Appendix **C**. Feel free to use any method provided for the rest of this question.

For the testing question, you can use the above heaps (`h1`, `h2`, `h3`) and assume that they are already initialized as variables and have the elements as shown in the diagrams.

a. (12 points) Your job is to complete **four** additional tests. Be sure to give your tests descriptive names. We will be grading the quality of your tests and how well they cover the invalid inputs of this method. Each test should cover a different situation.

```
// – your test #1 should test rightChild
@Test
public void testrightChild () {

        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
}


// – your test #2 should test leftSibling
@Test
public void __________________________________________ () {

        ____________________________________________
        ____________________________________________
        ____________________________________________
        ____________________________________________
}


// –– your test #3 Should test hasNext()
@Test
public void __________________________________________ () {

        ____________________________________________

        ____________________________________________

}


// –––– your test #4 Should test next ().
// next removes the top ( largest value of the heap and return it )
@Test
public void __________________________________________ () {

        ____________________________________________

        ____________________________________________

        ____________________________________________

}
```

b. (10 points) We will now implement the `next()` method. `next` removes and returns the value at the top of the heap or throws a `NoSuchElementException` if there are no more elements. `next()` should preserve the `MaxHeap` invariant.

If there are more elements in the heap, then the item at the top of the heap is swapped with the last item in the heap. The size of the heap is decremented and the item at the top of the heap is put in its correct place using the `siftDown`.

`siftDown` takes an integer representing the position of a value in the heap and will move the value at i to its correct position. siftDown should maintain the Heap invariants.

For example: Given

```
        (h1)              (h2)                (h3)

         8                 10                  12
        / \               / \                 / \
      5     6            7    2              8    9
     /                  / \                 / \   / \
    3                  4   1               6   7 3   5
                                          /
   calling                               2
   h1.next();  will return 8
   h2.next();  will return 10
   h3.next();  will return 12

   And will produce the following heaps

         (h1)             (h2)                (h3)

          6                 7                  9
        /  \              / \                 / \
       5    3            4    2              8     5
                        /                   / \   / \
                       1                   6   7 3   2
```

```
public Integer next() {
    // (1) Handle empty heap case



   // (2) Swap top of the heap with last value
   // (3) Decrement the heap size




  // (4) call siftDown
  // (5) return the appropriate value



}
```

c. (8 points) We will now implement `isLeaf`.

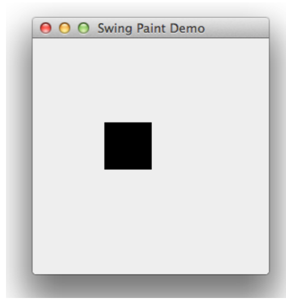
```
/**
 * Check if node at pos is a leaf node
 *
 * @param pos of current value
 * @return true if pos a leaf position, false otherwise
 */
public boolean isLeaf (int pos) {







}
```

d. (5 points) Now implement `hasNext`.

```
// return true if the heap has more elements
// hint: use the invariants!
public boolean hasNext () {






}
```

6. Java Swing Programming (10 points)

The code in Appendix **D** implements a simple Java GUI program in which a 50x50 black box follows the mouse cursor around the window. It looks like this (the mouse cursor is not shown):



The following true/false questions concern this application and Java Swing programming in general.

a. True ☐ False ☐

The type `MyPanel` is a subtype of `Object`.

b. True ☐ False ☐

The instance variables `x` and `y`, declared on lines 23 and 24, can only be modified by the methods of the `MyPanel` class or the methods of any inner classes of `MyPanel`.

c. True ☐ False ☐

The `mouseMoved` method on line 28 is called by the Swing event loop in reaction to the user moving the mouse in the main window of the application.

d. True ☐ False ☐

The `paintComponent` method on line 42 is only invoked once, at the start of the application.

e. True ☐ False ☐

The call `super.paintComponent()` on line 43 invokes the `paintComponent` method defined in class `JPanel`.

f. True ☐ False ☐

If the user replaced the code on line 31 (i.e. the call to `repaint()`)
with `paintComponent(new Graphics())`, then the behavior of the application would not change.

g. True ☐ False ☐

The anonymous class defined on line 27 implements or inherits all members of the `MouseMotionListener` interface.

We now replace lines 5–9 with the following code, which uses the new Java 8 “lambda” syntax. Note that the code below compiles.

```
1 SwingUtilities.invokeLater(
2    () -> { createAndShowGUI(); }
3 );
```

h. True ☐ False ☐

This code will have the same functionality as the original code shown in the Appendix.

i. True ☐ False ☐

The code on line 2 above creates a new `Object` in Java that has the static type `Runnable`.

j. True ☐ False ☐

The `GUI` class and the `createAndShowGUI()` method share the same stack and heap in the Java ASM.