CIS 120 Final Exam December 17, 2019

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: _________________________

- Please wait to begin the exam until you are told it is time for everyone to start.
- 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.
- Please write your name and Pennkey (e.g., `stevez`) 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 and Java Concepts** (15 points) (1.5 points each) Indicate whether the following statements are true or false.

a. True ☐ False ☐

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

In OCaml, defining a new type (as shown above for a BST) is sufficient for the compiler to automatically infer if a pattern match statement is non-exhaustive.

b. True ☐ False ☐
In OCaml, defining a new type (as shown above for a BST) is sufficient for the compiler to automatically infer the type's invariants.

c. True ☐ False ☐
In the OCaml ASM, stack bindings are mutable by default whereas in the Java ASM, they are immutable by default.

d. True ☐ False ☐
In OCaml, if a function returns an option type (`'a option`), the caller cannot use the `'a` directly without first checking if the returned value was `None` or `Some`.

e. True ☐ False ☐
In our OCaml GUI libraries, a container widget (like `hpair`) can handle an event by passing it onwards to one (or more) of its child widgets or ignoring the event completely.

f. True ☐ False ☐
In Java, the **`this`** reference is guaranteed to be non-**`null`**.

g. True ☐ False ☐
In Java, dynamic dispatch of a method invocation might require the Java ASM to search the entire stack to find the appropriate code to run next.

h. True ☐ False ☐
In Java, if we have a **`try`**-**`catch`**-**`finally`** block, the **`finally`** block will get executed even if the catch block raises some Exception.

i. True ☐ False ☐
In Java, it is possible to create an object of type `List<int>`.

j. True ☐ False ☐
In Java, the static type and dynamic class of an object can never be the same.

2. **Binary Search Trees with Duplicates** (23 points)

Recall the generic binary search tree in OCaml and its invariants. We want to modify this to store duplicates in the BST, henceforth called *BSTD*. Since we also want to preserve our BST invariants (and cannot store duplicates directly), we will modify each `Node` to store the value and a count of the number of times that value has been duplicated in the tree.

Here is our updated type definition:

```ocaml
(* A type of BST trees with duplicates *)
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * int * 'a tree
```

In addition to the usual BST invariants, we'll add one more invariant: **Every node has a count such that the count is greater than 0.** This has the following implications on inserting and deleting from the BSTD.

- When a new value is inserted into the tree, the count will get initialized to 1.
- When a duplicate value is inserted, the corresponding count gets incremented by 1.
- When a value is deleted, the corresponding count is decremented by 1.
- When the count becomes 0, the entire node will be deleted.

(a) (5 points) Mark the boxes for all trees below that **satisfy** the BSTD invariants. If a node is shown as (3, 2) below, the value is 3 and the count is 2.

(a) ☐

```
    (3, 2)
     /
  (1, 1)
   /
(0, 1)
```

(b) ☐

```
      (2, 1)
       / \
(1, 1)    (2, 2)
```

(c) ☐

```
        (4, 1)
         / \
(2, 1)   (5, 2)
      \
       (3, 6)
```

(d) ☐

```
          (3, 2)
           / \
     (2, 1)   (4, 4)
       /         \
(1, 3)            (8, 0)
```

(e) ☐

```
(1, 7)
```

(b) (4 points) Consider the following implementation of *buggy* insert for BSTD. Which of the following statements are true? (Mark all that apply.)

```
(* Inserts n into the BSTD t *)
let rec buggy_insert (t:'a tree) (n:'a) : 'a tree =
  begin match t with
    | Empty -> Node(Empty, n, 0, Empty)
    | Node(lt, x, count, rt) ->
        if x = n then Node(lt, x, count + 1, rt)
        else if n < x then Node (buggy_insert lt n, x, count, rt)
        else Node(lt, x, count, buggy_insert rt n)
  end
```

☐ The code is well-typed (i.e., it will compile without any errors)

☐ The code satisfies the BSTD invariants (i.e., if the input is a valid BSTD, the output will be a valid BSTD)

☐ The code uses the BSTD invariants (i.e., the code assumes that the input is a valid BSTD and leverages that information as part of the code)

☐ The code will *always* exhaust stack space when run since it's not tail recursive

(c) (4 points) Consider the following implementation of *buggy* lookup for BSTD. Which of the following statements are true? (Mark all that apply.)

```
(* Determines whether BSTD t contains n *)
let rec buggy_lookup (t:'a tree) (n:'a) : bool =
 begin match t with
  | Empty -> if buggy_lookup t n then true else false
  | Node(lt, x, _, rt) ->
    x = n || (buggy_lookup lt n) || (buggy_lookup rt n)
  end
```

☐ The code is well-typed (i.e., it will compile without any errors)

☐ The code requires the BSTD invariants (i.e., the code will work correctly only if the input is a valid BSTD)

☐ The code uses the BSTD invariants (i.e., the code assumes that the input is a valid BSTD and leverages that information as part of the code)

☐ The code will exhaust stack space *in certain cases* when run since it's not tail recursive

(d) (4 points) Consider the following implementation of *correct* `get_count` for BSTD. Which of the following statements are true? (Mark all that apply.)

```
(* returns the count of the number of times the value n is stored in the BSTD t. *)
(* if the value is not present, it should return 0. *)
let rec get_count (t:'a tree) (n:'a) : int =
  begin match t with
  | Empty -> 0
  | Node(lt, x, count, rt) ->
    if x = n then count
    else if n < x then get_count lt n
    else get_count rt n
  end
```

☐ The code is well-typed (i.e., it will compile without any errors)
☐ The code requires the BSTD invariants (i.e., the code will work correctly only if the input is a valid BSTD)
☐ The code uses the BSTD invariants (i.e., the code assumes that the input is a valid BSTD and leverages that information as part of the code)
☐ The code will exhaust stack space *in certain cases* when run since it's not tail recursive

Recall the definition of generic `fold` functions for lists, which is given in Appendix A. Consider the following modification to the `fold` function that now takes in a BSTD `t` as input.

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

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

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

☐ Sum up the values in the tree
☐ Sum up the counts 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 always exhaust stack space when run since it's not tail recursive
☐ It is ill-typed

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

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

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

3. **Java Typing, Inheritance, and Dynamic Dispatch** (25 points)

This problem refers to two interfaces and several classes that might be part of a program for keeping track of vehicles in Harry Potter. You can find them in Appendix B.

(a) (2 points) Which of the following classes are an example of simple inheritance in Java (either explicitly or implicitly)? (Mark all that apply.)

☐ `Car` ☐ `WeasleyFamilyCar` ☐ `HagridsMotorcycle` ☐ `SubTyping`

(b) (4 points) Which of the lines of code are an example of subtype polymorphism in Java? (Mark all that apply.)

☐ Line 44 ☐ Line 45 ☐ Line 47 ☐ Line 48

(c) (4 points) Which of the lines of code are an example of parametric polymorphism (i.e., generics) in Java? (Mark all that apply.)

☐ Line 44 ☐ Line 45 ☐ Line 47 ☐ Line 48

(d) (3 points)

```
_______ motorcycle = new HagridsMotorcycle();
```

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

☐ `Flyable`	☐ `HagridsMotorcycle`	☐ `Car`
☐ `Driveable`	☐ `WeasleyFamilyCar`	☐ `Object`

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)

(e) (3 points)

```
Flyable thestral = new Flyable();
```

☐ Legal Code

A. The static type of `thestral` is ___________________________.

B. The dynamic class of `thestral` 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) (3 points)

```
Car c = new WeasleyFamilyCar();
c.drive();
```

☐ Legal Code

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

- ☐ "Car says let's go for a drive"
- ☐ "WeasleyFamilyCar says let's go for a drive"
- ☐ 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):

____

(g) (3 points)

```
Car c = new WeasleyFamilyCar();
c.fly();
```

☐ Legal Code

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

- ☐ "Hagrid says I was borrowed from Sirius Black"
- ☐ "WeasleyFamilyCar says let's avoid the Whomping Willow"
- ☐ 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) (3 points)

```
Driveable d = new WeasleyFamilyCar();
HagridsMotorcycle motorcycle = (HagridsMotorcycle) d;
motorcycle.fly();
```

☐ Legal Code

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

- ☐ "Hagrid says I was borrowed from Sirius Black"
- ☐ "WeasleyFamilyCar says let's avoid the Whomping Willow"
- ☐ 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):

____

4. **Java Design Problem** (45 points total)

The command-line (e.g. terminal) `grep` utility is a classic and handy tool. Given a keyword string and the name of a text file, it prints out all the lines of the file that contain the keyword. Each printed line is prefixed by its line number (starting from 1), which makes it easy to find uses of the keyword in the file.

For example, suppose that we have a text file named `notes.txt` containing the data shown below:

```
notes.txt
My Notes:
CIS 120 is fun!
NETS 112 is easy.
CIS 160 is great.
CIS 121 will be amazing.
```

Running our grep utility on `notes.txt` with the keyword `"CIS"` produces the following output, which shows the lines (and line numbers) that contain `"CIS"`:

```
2:CIS 120 is fun!
4:CIS 160 is great.
5:CIS 121 will be amazing.
```

On the other hand, if we use grep to search `notes.txt` for the keyword `"ea"` instead, we get the following output because both `easy` and `great` contain the substring `ea`:

```
3:NETS 112 is easy.
4:CIS 160 is great.
```

If we use grep to search `notes.txt` for the keyword `"foo"`, no output is printed because there are no lines that contain that substring.

In this problem, you will use the design process to implement grep in Java.

Step 1: Understand the problem The main part of this task is closely related to the `FileLineIterator` code that you wrote for the TwitterBot project, which suggests that we use an `Iterator` of some kind. It also requires doing file I/O, so we should think about how to handle exceptional cases.

Step 2: Design the interfaces Recall that an iterator is an object that yields a sequence of elements. The Java docs for the `Iterator<E>` interface are given in Appendix C.2.4.

We will implement a `GrepIterator` class that implements the `Iterator<GrepResult>` interface. To make the `GrepIterator` easier to test, we can construct it from an arbitrary `Reader` object. It also needs to know what keyword to search for. We thus arrive at the following constructor declaration:

```
GrepIterator(Reader r, String keyword)
```

Such an iterator scans through the `Reader` line-by-line, looking for lines that contain the given keyword. When it finds such a line, it yields a `GrepResult`, which is a pair of an **`int`** line number and a `String` containing the data on that line. Appendix C.1 contains the complete code for a `GrepResult` class. Note that `GrepResult` overrides the `toString()` method—it prints the data in the format `<line#>:<data>`. It also overrides the `equals()` method, which will be useful for writing test cases.

(No questions on this page.)

We have reproduced the `equals()` method of the `GrepResult` class below:

```
 1  @Override
 2  public boolean equals(Object obj) {
 3          if (this == obj)
 4                  return true;
 5          if (obj == null)
 6                  return false;
 7          if (getClass() != obj.getClass())
 8                  return false;
 9          GrepResult other = (GrepResult) obj;
10          if (line == null) {
11                  if (other.line != null)
12                          return false;
13          } else if (!line.equals(other.line))
14                  return false;
15          if (lineNumber != other.lineNumber)
16                  return false;
17          return true;
18  }
```

(a) (3 points) Would deleting lines 3 and 4 from the code change the possible outputs of this `equals()` method?

☐ Yes ☐ No

In one sentence, explain why:

(b) (3 points) Would moving line 9 before line 7 change the possible outputs of this `equals()` method?

☐ Yes ☐ No

In one sentence, explain why:

(c) (3 points) Suppose we replace lines 10–14 of the (original) code above with:

```
if (line != other.line) return false;
```

Does this change the possible outputs of the `equals()` method?

☐ Yes ☐ No

In one sentence, explain why:

Step 3: Write test code One benefit of using the `Reader` interface is that we can use a `StringReader` to write test cases without needing to use the file system. Note that within a string we can use the character `'\n'` to indicate an end-of-line marker. We have given you one example test case written in this style below.

```
@Test
public void testGrepFindsTwo() {
    StringReader sr = new StringReader("abc\nxyz\nbcd");
    GrepIterator g = new GrepIterator(sr, "bc");
    GrepResult gr1 = new GrepResult(1, "abc");
    GrepResult gr2 = new GrepResult(3, "bcd");
    assertEquals(gr1, g.next());
    assertEquals(gr2, g.next());
    assertFalse(g.hasNext());
}
```

(a) (2 points) Complete the following test case:

```
@Test
public void testGrepFindsNone() {
        StringReader sr = new StringReader("abc\nxyz\nbcd");
        GrepIterator g = new GrepIterator(sr, "q");




}
```

(b) (4 points) Consider the following test case, which is missing the argument to the `StringReader` constructor:

```
@Test
public void testGrep() {
        StringReader sr = new StringReader(__???__);
        GrepIterator g = new GrepIterator(sr, "th");
        GrepResult gr1 = new GrepResult(2, "this");
        GrepResult gr2 = new GrepResult(3, "that");
        assertEquals(gr1, g.next());
        assertEquals(gr2, g.next());
        assertFalse(g.hasNext());
}
```

Which of the following strings could be filled in for __`???`__ such that the test case is correct? (Mark all that apply)

☐ `"this\nthat"`

☐ `"foo\nthis\nthat"`

☐ `"foo\nthis\nthat\nbar"`

☐ `"foo\nthis\nthat\nthe other"`

(c) (4 points) Consider the following test case, which is missing the keyword argument of the `GrepIterator` constructor:

```
@Test(expected = NoSuchElementException.class)
public void testGrepRaisesNoSuchElement() {
        StringReader sr = new StringReader("abc");
        GrepIterator g = new GrepIterator(sr, __???__);
        g.next();
}
```

Which of the following strings could be filled in for __`???`__ such that the test case is correct? (Mark all that apply)

☐ `"a"`

☐ `"x"`

☐ `"abc"`

☐ `"xyz"`

Step 4.1: Implement the Iterator (20 points) Complete the code on the following page. Your implementation should satisfy the `Iterator<GrepResult>` interface. In the case that a call to the `readLine()` method throws an `IOException`, the iterator should behave as though it has no next element.

Note: Here (and throughout the exam) you may assume that appropriate **`import`** statements bring the libraries into scope; we omit them to save space.

Note: Your code does *not* need to close the `Reader` object.

Hint: You might want to think about what *invariant* that the state of your iterator maintains.

Hint: We have declared a helper method called `advance()` that you can use to move your iterator forward to the next result (if any). Complete it as you see fit, and use it where appropriate. (Think about the invariant!)

Hint: You can use the `String.contains()` method, whose javadocs are shown in Appendix C.2.3. Note that `String` implements the `CharSequence` interface.

```
public class GrepIterator implements Iterator<GrepResult> {
        private final BufferedReader br;
        private final String keyword;
        /* TODO: Add fields as needed */





        public GrepIterator(Reader r, String keyword) {
                br = new BufferedReader(r);
                this.keyword = keyword;
                /* TODO: Complete this constructor */





        }

        @Override
        public boolean hasNext() {
          /* TODO: Complete this method */



        }

        @Override
        public GrepResult next() {
          /* TODO: Complete this method */








        }

        private void advance() {
          /* TODO: Complete this method */











        }
}
```

Step 4.2: Implement `main` (6 points) We can now package the `GrepIterator` in a `main` method that accepts arguments from the command line, creates the appropriate `FileReader`, instantiates the `GrepIterator` and then prints out the resulting sequence of `GrepResults`.

```
1   public class Grep {
2     public static void main(String[] args) {
3       if (args.length != 2) {
4         System.out.println("Usage: grep <keyword> <file>");
5         System.exit(0);
6       }
7       String fileName = args[1];
8       String keyword = args[0];
9       FileReader f = new FileReader(fileName);
10      GrepIterator g = new GrepIterator(f, keyword);
11      while (g.hasNext()) {
12        GrepResult gr = g.next();
13        System.out.println(gr);
14      }
15    }
16  }
```

Unfortunately, the code above *does not compile* because there is an unhandled exception. Explain how and where to add an appropriate **`try`–`catch`** block to fix the above. The exception handler should print an appropriate error message.

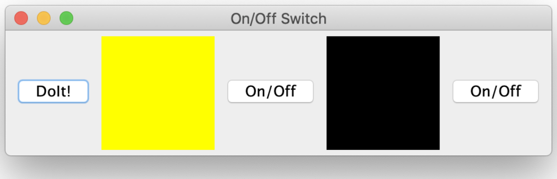
(a) (2 points) The code "**`try`** `{`" should be added just before line: _____________ . (There may be multiple correct answers, give one.)

(b) (4 points) Add the code below just before line: _____________ . (There may be multiple correct answers, give one.) Also, fill in the blank with as precise a type as possible.

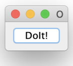
```
catch (_________________________________ e) {
  System.out.println("File " + fileName + " not found");
}
```

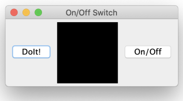
5. **Java Swing Programming** (12 points) (3 points each)

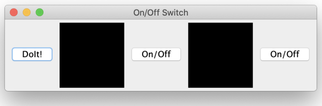
These questions refer to the code in Appendix D for a simple Swing application based on the `OnOff` demo from lecture. The `LightBulb` class is unchanged—it paints a square of black or yellow depending on the bulb state. The new code is in the `run` method of the `OnOff` class, which creates a user interface for manipulating the lightbulbs. Below is a picture of the user interface after some interactions:

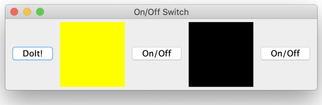


(a) Which of the following is a picture of the user interface when the app is first run? (Choose one.)

☐ 

☐ 

☐ 

☐ 

(b) There are seven occurrences of the **`new`** keyword in the `run` method (lines 19–49). How many of them correspond to anonymous inner classes? (Your answer should be in the range 0 – 7.)

____

(c) Which of the following best describes the effect of clicking the "DoIt!" button? (Choose one.)

☐ All of the light bulbs toggle from on to off and vice-versa.

☐ A new light-bulb and an On/Off button that controls it are created and added to the right side of the frame.

☐ A new light-bulb and an On/Off button that controls *all* of the bulbs are created and added to the right side of the frame.

☐ First, all of the light bulbs toggle from on to off and vice-versa and then a new light-bulb and an On/Off button that controls it are created and added to the right side of the frame.

(d) Note that the `OnOff` class implements the `Runnable` interface. This is useful because the `SwingUtilities.invokeLater` call (on line 52) does which of the following? (Choose one.)

☐ It uses dynamic dispatch to invoke the correct method implementation for `invokeLater`.

☐ It uses static dispatch to invoke the `OnOff` class's `run` method at an appropriate time.

☐ It creates a new thread that handles the event loop for the `OnOff` user interface.

☐ It uses polymorphism to ensure that the `OnOff` class's `run` method is called via parametric dispatch.