CIS 120 Final Exam December 15–18, 2020

SOLUTIONS

1. **OCaml and Java Concepts** (20 points) (2 points each)

   Indicate whether the following statements are true or false.

   a. True ☒ False ☐

   In OCaml, the intended behavior of an abstract type is defined by its interface and its properties, not its implementation.

   b. True ☐ False ☒

   In OCaml, it is possible to use the sequencing operator `;` to execute multiple expressions and return multiple values.

   c. True ☒ False ☐

   In OCaml, making a function tail recursive is an optimization for the ASM that makes it use a constant amount of stack space.

   d. True ☐ False ☒

   An OCaml abstract type is analogous to an anonymous inner class in Java.

   e. True ☐ False ☒

   In our OCaml GUI libraries, a `listener` must update the state of its child widgets before passing down the event for subsequent event handling.

   f. True ☒ False ☐

   In Java, if you create a new class `A`, you can use the **`super`** reference to call methods from the superclass and the **`super`** reference is guaranteed to be non-**`null`**.

   g. True ☐ False ☒

   In Java, if `s` and `t` are values of type `String` such that `s.equals(t)` evaluates to **`true`**, then `s == t` always evaluates to true.

   h. True ☐ False ☒

   In Java, even if code successfully compiles, it is possible for the dynamic dispatch of a method invocation to fail and throw an Exception if the Java ASM cannot find the required method in the given type or its supertypes.

   i. True ☐ False ☒

   In Java, it is possible to create an object having the dynamic type `List<Integer>`.

   j. True ☐ False ☒

   In Java, if a class `C` implements the `Iterator<E>` interface, an instance `o` of class `C` can be used in a **`for`**`-each` loop like this: **`for`** `(E e : o) {...}`.

2. **Java Code to OCaml** (14 points total)

Consider the `Animal` interface and the `Cat` class shown in Appendix A. We want to translate these into OCaml code and maintain the same properties and behavior as much as possible.

(a) (2 points) Which of the following will be a correct translation of the `Animal` interface? (Note that some answer choices might not compile successfully.) (Choose one.)

☐

```
type animal = {
  get_name : string -> unit;
  distinguishing_feature : string -> unit;
}
```

☒

```
type animal = {
  get_name : unit -> string;
  distinguishing_feature : unit -> string;
}
```

☐

```
module type ANIMAL = sig
  type 'a animal

  val get_name : 'a animal -> string
  val distinguishing_feature : 'a animal -> string
end
```

☐

```
module type ANIMAL = sig
  type 'a animal

  val get_name : 'a animal -> unit -> string
  val distinguishing_feature : 'a animal -> unit -> string
end
```

(b) (2 points) Which of the following will be true for a correct translation of the `Cat` class? (Choose one.)

☐ It should satisfy the relevant type/signature (from above) by providing appropriate implementations for both methods and *optionally* keep track of the `name` variable.

☐ It should satisfy the relevant type/signature (from above) by providing appropriate implementations for both methods and *additionally* keep track of the `name` variable by using un-encapsulated immutable state.

☒ It should satisfy the relevant type/signature (from above) by providing appropriate implementations for both methods and *additionally* keep track of the `name` variable by using encapsulated mutable state.

(c) (10 points) Consider the Java code below that iterates over a list of Cats. For each option below, state whether it would do the analogous thing in OCaml. If not, explain what it would do instead.

```
for (Cat c : cats) {
    System.out.println(c.getName());
}
```

You can assume that there is a list `cats` that is defined earlier and in scope. Regardless of the answer choices for the previous two parts, you can assume that if `c` is of type `cat` then `c.get_name ()` will return a `String` that's the name of the cat.

The definitions of `transform` and `fold` are provided in the Appendix.

a. True ☐ False ☒

```
fold (fun hd acc -> hd.get_name () ^ acc) "" cats
```

If False is chosen above, what would this code do instead?

_____ Answer: This will return a string of all the names concatenated together (and not print).

b. True ☐ False ☒

```
transform (fun x -> x.get_name () ) cats
```

If False is chosen above, what would this code do instead?

_____ Answer: This will return a list of strings of all the names together (and not print).

c. True ☒ False ☐ (`List.rev` *reverses a list*)

```
fold (fun x _ -> print_endline (x.get_name ())) () (List.rev cats)
```

If False is chosen above, what would this code do instead?

_____

d. True ☐ False ☒

```
begin match cats with
| [] -> ""
| hd::tl -> hd.get_name () :: tl
end
```

If False is chosen above, what would this code do instead?

_____ Answer: This will not compile.

e. True ☒ False ☐

```
let rec f (c : animal list): unit =
  begin match c with
  | [] -> ()
  | hd :: tl ->
    print_endline (hd.get_name ()); f tl
 end

;; f cats
```

If False is chosen above, what would this code do instead?

_____

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

This problem refers to several *new* Java classes or interfaces `A`, `B`, `C`, `D`, `E`, and `F`. The exact relationship between them and whether they are classes or interfaces is unknown. For each sub-problem below:

- *Mark all answer choices that apply*.
- The various classes and interfaces are distinct. (i.e., `A` is not the same as `B`.)
- The provided code should be legal Java code (i.e., it will not cause any compile-time or run-time errors), unless specified.
- Each sub-problem is independent and *does not* depend on the preceding/succeeding sub-problems.
- Each answer choice is independent as well and *does not* depend on any of the other answer choices.

(a) (3 points)

```
A a = new B();
```

- ☒ `A` must be a supertype of `B`
- ☐ `A` must be a subtype of `B`
- ☒ `A` can be an interface
- ☒ `A` can be an abstract class
- ☐ `B` can be an interface
- ☐ `B` can be an abstract class

(b) (3 points)

```
A a = new B();
C c = a.m1();
```

- ☐ The method `m1` must be defined in the type definition for `A`
- ☐ The method `m1` must be defined in the type definition for `B`
- ☒ The method `m1` can be overriden in the type `B`
- ☐ The method `m1` must be overriden in the type `B`
- ☐ `C` must be an interface
- ☒ `C` can be an interface

(c) (3 points)

```
1 C<Set> list = new D<Set>();
2 LinkedList<C> list = new LinkedList<C>();
```

- ☒ Line 1 is an example of subtype polymorphism
- ☒ Line 1 is an example of parametric polymorphism
- ☐ Line 2 is an example of subtype polymorphism
- ☒ Line 2 is an example of parametric polymorphism
- ☐ `D` must be defined via simple inheritance
- ☐ `D` cannot be defined via simple inheritance

(d) (3 points)

```
1  D d = ... // code omitted
2  ...       // code omitted
3  ...       // code omitted
4  d.m2();
```

- ☐ It is always possible to know statically the exact method `m2` that will be executed on the line 4
- ☒ The variable `d` could refer to different objects on line 1 and line 4
- ☒ It is always possible to know the static type of `d`
- ☐ It is always possible to know the dynamic class of `d`
- ☐ It is possible to change the static type of `d`
- ☒ It is possible to change the dynamic class of `d`

(e) (4 points)

```
1  E e = ... // code omitted
2  ...       // code omitted
3  ...       // code omitted
4  ((C) e).m3();
```

- ☒ It is possible that Line 4 will not compile
- ☒ It is possible that Line 4 will compile and throw a `ClassCastException` when run
- ☒ It is possible that Line 4 will compile and throw some other exception than `ClassCastException` when run
- ☒ It is possible that Line 4 will compile and throw no exceptions when run

(f) (4 points) The entire method body for a method `m3` in class `F` is shown below:

```
1    // omitted code
2    this.m4(); // m4 throws an IOException
3    // omitted code
```

- ☐ The method `m3` will never compile
- ☒ It is possible for the method `m3` to compile, if the omitted code includes a **`try-catch`** block
- ☒ It is possible for the method `m3` to not compile, even if the omitted code includes a **`try-catch`** block
- ☒ It is possible for the method `m3` to compile, even if the omitted code does not include a **`try-catch`** block

(g) (5 points) Consider exception handling in Java with **`try-catch-finally`** blocks:

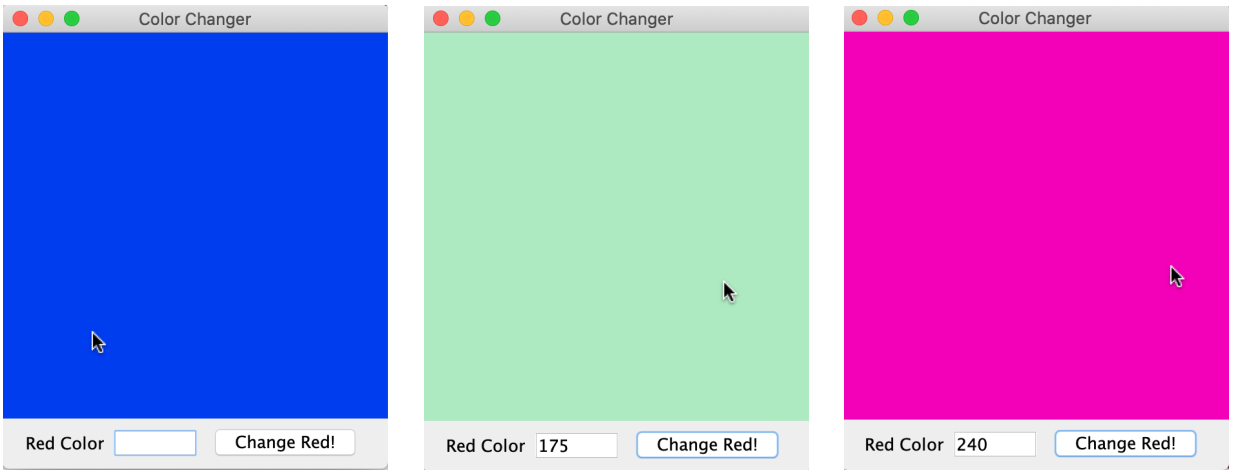
- ☒ For a given **`try`** block, it is possible to have multiple **`catch`** blocks
- ☒ It is possible for a **`catch`** block to throw an Exception
- ☐ It is required to use a **`finally`** block if you use a **`try-catch`** block
- ☐ The **`finally`** block only gets executed if the code in the **`try`** block throws an Exception
- ☐ The use of a **`try-catch-finally`** block guarantees that no Exceptions are thrown

4. **Java Swing Programming** (16 points total)

These questions refer to the code in Appendix B for a simple Swing application similar to the `OnOff` demo from lecture. The `ColorBulb` sets its background color based on the values of the `red`, `green`, and `blue` integers. These values are updated in the `ColorChanger` class as follows:

- The `red` value comes from the value the user enters in the text field.
- The `green` value comes from the `x` coordinate of the current mouse position.
- The `blue` value comes from the `y` coordinate of the current mouse position.

Below are pictures of the user interface after some interactions:



(a) (2 points) There are nine occurrences of the **`new`** keyword in the `run` method (lines 38–76). How many of them correspond to anonymous inner classes? (Your answer should be in the range 0 – 9.)

_____ Answer: 2

(b) (4 points) From the Pennstagram homework (HW06), recall that RGB values should be in the range 0 to 255 (both inclusive). What would happen if the user entered a value greater than 255 in the textbox and clicked the "Change Red" button? (Hint: See the relevant Javadocs.) (Choose one.)

- ☐ It will set the red component to 0.
- ☐ It will set the red component to 255.
- ☐ The code will throw an `IndexOutOfBoundsException`.
- ☒ The code will throw an `IllegalArgumentException`.
- ☐ The red component will remain unchanged and the code will not throw any exceptions.

(c) (5 points) Consider the `ActionListener` for the button (lines 56–64). Which of the following are true? (Mark all that apply.)

- ☒ The argument to the `addActionListener` method has the static type of `ActionListener`.
- ☐ The argument to the `addActionListener` method has the dynamic class of `ActionListener`.
- ☐ The `paintComponent` method of `ColorBulb` will be invoked (directly or indirectly) every time the user clicks the button.
- ☒ It is possible to replace this code using Java 8's new `lambda` syntax.
- ☒ It is possible that this may throw an `Exception` when the user clicks the button.

(d) (5 points) Consider the `ColorBulb` class (lines 1–32). Which of the following are true? (Mark all that apply.)

- ☒ The type `ColorBulb` is a subtype of `Object`.
- ☒ There is exactly one object in the Java ASM heap that has the type `ColorBulb` when the `ColorChanger` program is running.
- ☐ The `paintComponent` method will be invoked (directly or indirectly) *only* when the user clicks the button.
- ☒ The `ColorBulb` constructor (lines 4–8) will automatically call the constructor of `JComponent` *before* executing the rest of the code (lines 5–7).
- ☒ If we deleted the `@Override` annotation on Line 10, then the behavior of the application would not change.

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

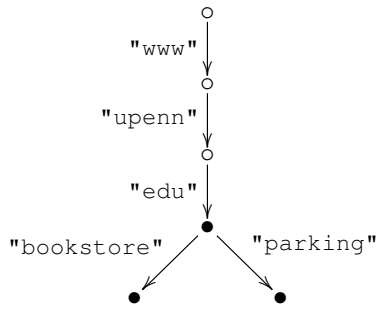
In this problem, we implement a (simplified) Java `Collection` class called a `PathSet` that is designed to efficiently store *sets* of lists of strings. Such a collection might be useful for indexing a web site, where we might want to keep track of a set of "parsed" URLs or file folder paths. We will follow the design process to complete this implementation.

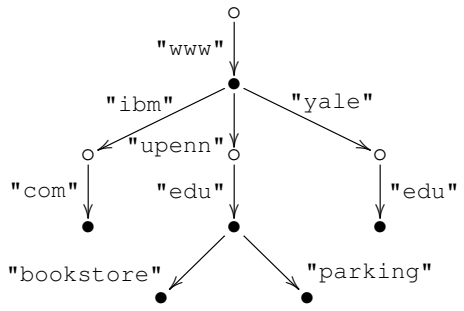
Step 1: Understand the problem We can think of a URL like `www.upenn.edu/parking` as a *list* of *strings*, where we have parsed the URL into constituent components (separated by punctuation). We could write this list as `["www", "upenn", "edu", "parking"]`. We will therefore represent such URLs as Java objects of type `List<String>`. We will call such a list a "path".

If we want to store a large number of URL paths we could use one of the standard Java collection classes like `TreeSet<List<String>>`, but this can be wasteful because many of the URLs could be similar. For instance, it is likely that we will see many URLs of the form `www.upenn.edu/<DIRECTORY>` , and so many of the paths in our collection will start with the common sublist `["www", "upenn", "edu"]`. We can exploit this similarity to save space by arranging the set of paths as a kind of tree whose edges are labeled by `Strings` and whose nodes are labeled by `booleans`. A list of strings in the set corresponds to a single *path* through the tree (starting from the root). Two lists that start with the same sublist will share the common path. We call this collection a `PathSet`.

There are a couple subtleties. First, unlike the binary trees we've previously studied, each node can have an arbitrary number of children—for example, there can be many directories in the `www.upenn.edu` domain and they'll all share a common path. Second, one list in the set may extend another: we should be able to store both the lists `www.upenn.edu` *and* `www.upenn.edu/parking`, which share a sublist, but the set *might not* have just the list `["www"]`. Thus, each node carries a **`boolean`** marker that indicates whether the sequence of edges from the root to the node corresponds to a path in the set.

For example, if we add the three lists `["www", "upenn", "edu", "parking"]` and `["www", "upenn", "edu"]`, and `["www", "upenn", "edu", "bookstore"]` to an empty `PathSet`, we could draw the resulting tree as shown on the left below. A • node marker means that the path ending at the node is in the set, and the ○ node marker indicates otherwise. If we later add the three lists `["www", "yale", "edu"]` and `["www", "ibm", "com"]` and `["www"]` to this collection, we end up with the tree on the right.





(There are no questions on this page.)

a. (2 points) Which tree pictured below represents an *empty* `PathSet`?

☒ ☐ ☐  ☐ 

b. (2 points) ☒ True or ☐ False The number of paths contained in a `PathSet` collection depicted as a tree is the same as the number of • nodes appearing in the tree.

Step 2: Design the interface For this problem, we'll use a cut-down collection interface called `SimpleCollection`, which is shown in Appendix C.1. The methods of this interface are intended to satisfy the same specification as those given by the full Java Collections library[1]. There are other Java interfaces mentioned, including `List<E>` and `Iterable<E>`, whose documentation you are welcome to use.

Appendix C.2 contains a complete implementation of the `SimpleCollection<List<String>>` interface in terms of a `PathSetNode` class that you will implement below. Here we consider the interface itself. Note that the interfaces and documentation permit `add` and `contains` to raise `NullPointerException` when given a **`null`** list as input.

a. (2 points) According to the method types specified by `SimpleCollection`, which of the following methods might raise an uncaught `ClassCastException` when called? (Mark all that apply or check the "None" box if none apply.)

☒ `isEmpty` ☒ `add` ☒ `contains` ☒ `iterator`

☐ None might

b. (2 points) According to the JavaDocs for the Java `Collection` interface, which of the following methods might raise an uncaught `ClassCastException` when called? (Mark all that apply or check the "None" box if none apply.)

☐ `isEmpty` ☒ `add` ☒ `contains` ☐ `iterator`

☐ None might

c. (2 points) According to the method types specified by `SimpleCollection`, which of the following methods might raise an uncaught `IOException` when called? (Mark all that apply or check the "None" box if none apply.)

☐ `isEmpty` ☐ `add` ☐ `contains` ☐ `iterator`

☒ None might

[1] see `https://docs.oracle.com/javase/8/docs/api/?java/util/Collection.html`

Step 3: Write test cases

a. (2 points) One of the most important properties of a collection is the relationship between the `add` and the `contains` methods. Complete the JUnit test case below by filling in the blanks to make it so that a correct implementation of `PathSet` passes this test.

```
@Test
void testAddContains() {
    List<String> url1 = Arrays.asList(new String[] { "www", "upenn", "edu" });
    List<String> url2 = Arrays.asList(new String[] { "www", "upenn", "edu", "parking" });
    List<String> url3 = Arrays.asList(new String[] { "www", "upenn", "edu", "bookstore" });
    List<String> url4 = Arrays.asList(new String[] { "www" });

    PathSet s = new PathSet();
    Assertions.assertTrue(s.isEmpty());
    Assertions.assertFalse(s.contains(url1));
    Assertions.assertFalse(s.contains(url2));
    Assertions.assertFalse(s.contains(url3));
    Assertions.assertFalse(s.contains(url4));

    s.add(url3);
    Assertions.assertFalse(s.isEmpty());
    Assertions.assertFalse(s.contains(url1));
    Assertions.assertFalse(s.contains(url2));
    Assertions.assertTrue(s.contains(url3));
    Assertions.assertFalse(s.contains(url4));

    s.add(url2);
    Assertions.assertFalse(s.isEmpty());
    Assertions.assertFalse(s.contains(url1));
    Assertions.assertTrue(s.contains(url2));
    Assertions.assertTrue(s.contains(url3));
    Assertions.assertFalse(s.contains(url4));

    s.add(url1);
    Assertions.assertFalse(s.isEmpty());
    Assertions.assertTrue(s.contains(url1));
    Assertions.assertTrue(s.contains(url2));
    Assertions.assertTrue(s.contains(url3));
    Assertions.assertFalse(s.contains(url4));

    s.add(url4);
    Assertions.assertFalse(s.isEmpty());
    Assertions.assertTrue(s.contains(url1));
    Assertions.assertTrue(s.contains(url2));
    Assertions.assertTrue(s.contains(url3));
    Assertions.assertTrue(s.contains(url4));
}
```

Step 4: Implement the code As mentioned above, Appendix C.2 provides a complete implementation of the `PathSet` class. This handles the "interface" mismatches and simply delegates the actual work of the implementation to a `PathSetNode root` object. Here we implement that class. You may assume that the file has these imports:

```
import java.util.List;
import java.util.LinkedList;
import java.util.Map;
import java.util.TreeMap;
```

A `PathSetNode` represents a node in the tree. As shown below (the code is also available in Appendix C.3), each node has a **`boolean`** field `isLast`, which marks whether it is a terminal node of a path in the set. A node's `children` field stores the outgoing edges of this node as a `TreeMap<String, PathSetNode>` collection [2]. Each entry `e` in this map represents one edge of the tree: the key (a `String`) is the edge's label, which maps to the `PathSetNode` child value. A leaf of the tree is a node that has no children.

In addition to marking whether "internal" nodes are last, *your implementation should maintain the invariant that every leaf node has* `isLast` *set to* **`true`**. We have given you an implementation of `isEmpty()` that exploits this invariant. The provided constructor builds a node that represents an empty set of paths.

a. (10 points) Complete the `add(List<String> path)` method, which adds a new path to the set represented by the `PathSetNode`.

NOTE: Your implementation *should not* mutate the provided `path`!

(Fill in the code on the next page.)

b. (10 points) Complete the `contains` method, which returns **`true`** if the given path is in the set and **`false`** otherwise.

NOTE: Your implementation *should not* mutate the provided `path`!

(Fill in the code on page 14.)

Hint: Our reference implementations for these methods use about 10 lines of code each. If your solution is substantially longer than that, seek a simpler path.

[2] `https://docs.oracle.com/javase/8/docs/api/index.html?java/util/TreeMap.html`

```
public class PathSetNode {
    private TreeMap<String, PathSetNode> children;
    private boolean isLast;

    public PathSetNode() {
        this.children = new TreeMap<String, PathSetNode>();
        this.isLast = false;
    }

    public boolean isEmpty() {
        return (!this.isLast && children.isEmpty());
    }

public void add(List<String> path) {
    PathSetNode current = this;
    for (String s : path) {
        if (current.children.containsKey(s)) {
            current = current.children.get(s);
        } else {
            PathSetNode child = new PathSetNode();
            current.children.put(s, child);
            current = child;
        }
    }
    current.isLast = true;
}
```

```
public boolean contains(List<String> path) {
    PathSetNode current = this;
    for (String s : path) {
        if (current.children.containsKey(s)) {
            current = current.children.get(s);
        } else {
            return false;
        }
    }
    return (current.isLast);
}
```

Step 5: Iterate!

The last step of the design process is to refine and improve the design. In particular, it still remains to satisfy the `Iterable<List<String>>` part of the `SimpleCollection` interface. Recall that this means that we must equip `PathSet` with a method `Iterator<List<String>> iterator()`, which gives in iterator object that enumerates the paths of the collection. The iterator should produce them using *lexicographic (dictionary) order*. We have given you an implementation that does this using the `PathSetNode` helper method `toListOfPaths()`.

a. (3 points) As usual, we first proceed by with test cases. Consider the test below with some code omitted:

```
@Test
void testIterator() {
    List<String> url1 = Arrays.asList(new String[] {"a"});
    List<String> url2 = Arrays.asList(new String[] {"a", "b", "c"});
    List<String> url3 = Arrays.asList(new String[] {"a", "b", "d"});
    List<String> url4 = Arrays.asList(new String[] {"a", "b", "a"});
    List<String> url5 = Arrays.asList(new String[] {"a", "c"});

    PathSet s = new PathSet();
    Assertions.assertTrue(s.isEmpty());

    /* OMITTED CODE */

    Iterator<List<String>> it = s.iterator();
    Assertions.assertTrue(it.hasNext());
    Assertions.assertEquals(url1, it.next());
    Assertions.assertEquals(url4, it.next());
    Assertions.assertEquals(url2, it.next());
    Assertions.assertEquals(url3, it.next());
    Assertions.assertEquals(url5, it.next());
    Assertions.assertFalse(it.hasNext());
    Assertions.assertThrows(NoSuchElementException.class, () -> {it.next();});
}
```

Which of the following statements correctly describe the omitted code such that the test case compiles and a good implementation of `PathSet<T>` passes the test? Assume that the omitted code contains no JUnit assertions. (Mark all that apply.)

☐ The omitted code must contain `s.add(url1)` before any other calls to `s.add()`.

☒ The omitted code must contain at least one call to `s.add(url3)`.

☒ The omitted code may contain three calls to `s.add(url5)`.

☐ The omitted code could include the two lines shown below:

```
PathSet t = s;
t.add(Arrays.asList(new String[] {"b"}));
```

☐ The omitted code might raise an uncaught `NoSuchElementException`.

☒ The omitted code could include the two lines shown below:

```
Iterator<List<String>> another = s.iterator();
another.next();
```

b. (10 points) Write a method for the `PathSetNode` class that returns the list of all paths stored in the `PathSet` rooted at the node. The list should be sorted in lexicographic order. (We have reiterated the initial part of the class declaration for your reference.)

Hint: Recall that a `TreeMap` object's `entrySet()` method returns key-value entries sorted by their keys.

Hint: Use recursion.

```
public List<List<String>> toListOfPaths() {
    List<List<String>> result = new LinkedList<List<String>>();
    /* If this node is last, add the empty list as a path */
    if (this.isLast) {
        List<String> l = new LinkedList<String>();
        result.add(l);
    }
    /* Process each edge of this node */
    for (Map.Entry<String, PathSetNode> e : children.entrySet()) {
        PathSetNode n = e.getValue();
        /* Add all the children prefixed by the edge name */
        List<List<String>> tails = n.toListOfPaths();
        for (List<String> l : tails) {
            l.add(0, e.getKey());
            result.add(l);
        }
    }
    return result;
}
```

CIS 120 Final Exam — Appendices

A Java to OCaml Code

A.1 `Animal` interface and `Cat` class

```
public interface Animal {
    public String getName();
    public String distinguishingFeature();
}


public class Cat implements Animal {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String distinguishingFeature() {
        return "chase the mouse!";
    }
}
```

A.2 Higher-order list processing functions in OCaml

```
let rec transform (f: 'a -> 'b) (l: 'a list): 'b list =
 begin match l 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
   | h :: t -> combine h (fold combine base t)
 end
```

B Java Swing Code

```
1  class ColorBulb extends JComponent {
2      private int red, green, blue;
3
4      public ColorBulb() {
5          red = 0;
6          green = 0;
7          blue = 0;
8      }
9
10     @Override
11     public void paintComponent(Graphics gc) {
12         gc.setColor(new Color(red, green, blue));
13         gc.fillRect(0, 0, 300, 300);
14     }
15
16     @Override
17     public Dimension getPreferredSize() {
18         return new Dimension(300, 300);
19     }
20
21     public void setRed(int red) {
22         this.red = red;
23     }
24
25     public void setGreen(int green) {
26         this.green = green;
27     }
28
29     public void setBlue(int blue) {
30         this.blue = blue;
31     }
32 }
33
34 public class ColorChanger implements Runnable {
35
36     @Override
37     public void run() {
38         JFrame frame = new JFrame("Color Changer");
39         frame.setLayout(new BorderLayout());
40
41         ColorBulb color = new ColorBulb();
42         frame.add(color, BorderLayout.CENTER);
43
44         JPanel panel = new JPanel();
45         frame.add(panel, BorderLayout.SOUTH);
46
47         JLabel label = new JLabel("Red Color");
48         panel.add(label);
49
50         JTextField redTextField = new JTextField(5);
51         panel.add(redTextField);
52
53         JButton button = new JButton("Change Red!");
54         panel.add(button);
```

```
55
56          button.addActionListener(new ActionListener() {
57              @Override
58              public void actionPerformed(ActionEvent e) {
59                  String r = redTextField.getText();
60                  int red = Integer.parseInt(r);
61                  color.setRed(red);
62                  color.repaint();
63              }
64          });
65
66          color.addMouseMotionListener(new MouseAdapter() {
67              @Override
68              public void mouseMoved(MouseEvent e) {
69                  color.setGreen(e.getX() % 255);
70                  color.setBlue(e.getY() % 255);
71                  color.repaint();
72              }
73          });
74          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
75          frame.pack();
76          frame.setVisible(true);
77      }
78
79      public static void main(String[] args) {
80          SwingUtilities.invokeLater(new ColorChanger());
81      }
82  }
```

C Java Code For PathSet

C.1 SimpleCollection

```
public interface SimpleCollection<E> extends Iterable<E> {
  boolean isEmpty();
  void add(E x);
  boolean contains(Object o);
}
```

C.2 PathSet.java

```
import java.util.Iterator;
import java.util.List;

public class PathSet implements SimpleCollection<List<String>> {

    private PathSetNode root;

    public PathSet() {
        this.root = new PathSetNode();
    }

    @Override
    public boolean isEmpty() {
        return root.isEmpty();
    }

    @Override
    public void add(List<String> path) {
        root.add(path);
    }

    @Override
    public boolean contains(Object o) {
        @SuppressWarnings("unchecked")
        List<String> path = (List<String>) o;
        return root.contains(path);
    }

    @Override
    public Iterator<List<String>> iterator() {
        List<List<String>> lls = root.toListOfPaths();
        return lls.iterator();
    }
}
```

C.3 `PathSetNode.java` (excerpt)

```
public class PathSetNode {
    private TreeMap<String, PathSetNode> children;
    private boolean isLast;

    public PathSetNode() {
        this.children = new TreeMap<String, PathSetNode>();
        this.isLast = false;
    }

    public boolean isEmpty() {
        return (!this.isLast && children.isEmpty());
    }

    public void add(List<String> path) {
      /* TODO */
    }

    public boolean contains(List<String> path) {
      /* TODO */
      return false;
    }

    public List<List<String>> toListOfPaths() {
      /* TODO */
      return null;
    }
}
```