NATHAN VAN
ID: 44585152

Informatics 115, Fall 2022
Professor Iftekhar Ahmed

Homework 2
Due Thursday, October 20, 2022 at 11:59PM

Late policy: Every hour past this will result in 1% off your score.

Instructions: Please submit your written answers as one PDF file and your test program file (.java file for question 4) to CANVAS for Homework 2. The PDF file should be uploaded named "INF115-HW-10-24p" and the Java file should be uploaded to the CANVAS named "INF115-HW-10-24j". Please include your name and student ID in your documents (as a 1st page header in the PDF, and in the comments at the beginning of the file in the Java file). For the PDF, you may produce your answers with drawing/ diagramming software or simply use pencil and paper and scan/photograph the result. For example, you may choose to write your answers on a printed version of this homework, then scan it. However, please in the end, produce a single multi-page PDF instead of a series of single-page image files.

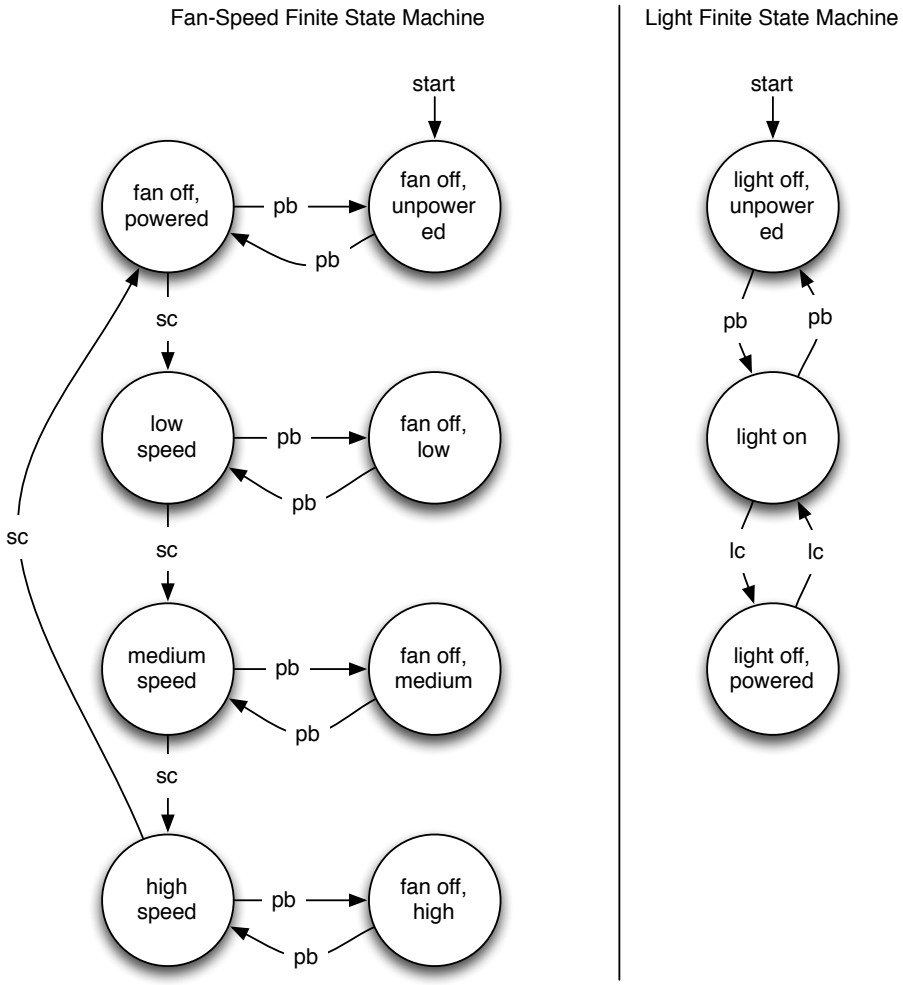
Plagiarism warning: Please be aware that although you may help each other to understand the concepts that we are practicing, you may not provide each other or receive the answers to these questions. Submitting someone else's work will be cause for me to take punitive actions, including reporting such activity to the academic dishonesty office.

Discussion forum: There is a Ed discussion thread for this homework. I encourage you to post clarifying questions there so that everyone can benefit from the answers, and so that the TA's and I do not need to answer the same questions repeatedly. I also encourage you to participate there: ask questions, and offer help to each other. Again, I am saying "help" not "give the answer". I will be monitoring the board.

1. Finite state machine verification [24 points]

Specification:
A ceiling fan has two pull cords attached to it and one power button that is mounted on the wall by the door to the room. The power button acts as a toggle for activating (and deactivating) the fan — we will call this the "power button". The pull cords control the fan speed and the light bulb. The "light cord" activates and deactivates the light bulb on the ceiling fan. The "speed cord" cycles the fan through four speed settings: "1: low speed", "2: medium speed", "3: high speed", and "0: off". When the power button is already on, and the power button is then pressed, all power to the fan is effectively shut off, regardless of whether its light is on or its fan is spinning. However, for convenience of use, when the power button is already off, and the power button is then pressed, the power to the fan is resumed in such a way as to make the light bulb turn on and the fan speed is resumed to the speed that it was running prior to shutting off the power button. When first installed, the power button is in the off position (i.e., "unpowered"), and the first time the power button is pressed, the fan speed is "off" and the light-bulb is "on".

Finite State Machines: ("pb" stands for "power button", "sc" stands for "speed cord", and "lc" stands for "light cord")



Program:

```
/* CeilingFan class represents the state of the ceiling fan. When created,
 * its fields correspond to the light and fan speed of the physical fan.
 * When destroyed, the fan and lights are turned off (effectively turning off
 * the power to the fan.
 */
public class CeilingFan {
      public int speed;
      public boolean isLightOn;

      public void CeilingFan() { // ceiling fan created when powered on
            speed = 0; // 0=off, 1=low, 2=medium, 3=high
            isLightOn = true;
      }

      public void speedCordPulled() { // called when user pulls speed cord
            speed = (speed + 1) % 4;
      }

      public void lightCordPulled() { // called when user pulls light cord
            isLightOn = !isLightOn;
      }
}

/* CeilingFanSystem class represents the state of the room, including the
 * power button and the ceiling fan. It should always be created .
 */
public class CeilingFanSystem {
      public boolean isPowered;
      public CeilingFan fan;

      public void CeilingFanSystem() {
            isPowered = false;
            fan = null;
      }

      public void powerButtonPushed() { // called when user presses button
            if (isPowered) {
                  fan = null; // destroy CeilingFan object to cut power
            } else {
                  fan = new CeilingFan(); // create CeilingFan
            }
            isPowered = !isPowered;
      }

      public static void main() {
            CeilingFanSystem system = new CeilingFanSystem();
      }
}
```

Notes: Do not worry about synchronization issues. You can assume that the CeilingFanSystem is started for each test case. You can assume that "`fan = null`;" will cause the garbage collector to immediately destroy the CeilingFan and thus turn off the light and fan. You can assume that when a user performs an action (e.g., pulling one of the cords or pressing the button), the corresponding methods are always called if available.

1a. [8 points] Check the finite state machines for completeness. To do this draw the state-transition table(s) corresponding to the machines. Draw the state-transition tables, explicitly. (We want to check them.) With this state-transition table, identify any missing transitions (by leaving those cells empty). Draw on the missing transitions to match the specification. (You can either redraw the machines or add onto mine and then scan or take a photo.)

FAN-SPEED STATE-TRANSITION TABLE

NOTE: OFF (00); LOW (01); MED (10); HIGH (11).

PB	SC	STATE	NEXT STATE	OUTPUT
0	0	OFF (UNPOWERED)	OFF (UNPOWERED)	0 0
0	**1**	**OFF (UNPOWERED)**		
1	0	OFF (UNPOWERED)	OFF (POWERED)	0 0
1	1	OFF (UNPOWERED)	LOW	0 1
0	0	OFF (POWERED)	OFF (POWERED)	0 0
0	1	OFF (POWERED)	LOW	0 1
1	0	OFF (POWERED)	OFF (UNPOWERED)	0 0
1	**1**	**OFF (POWERED)**		
0	0	LOW	LOW	0 1
0	1	LOW	MED	1 0
1	0	LOW	OFF (LOW)	0 0
1	**1**	**LOW**		
0	0	MED	MED	1 0
0	1	MED	HIGH	1 1
1	0	MED	OFF (MED)	0 0
1	**1**	**MED**		
0	0	HIGH	HIGH	1 1
0	1	HIGH	OFF (POWERED)	0 0
1	0	HIGH	OFF (HIGH)	0 0
1	**1**	**HIGH**		

LIGHT STATE-TRANSITION TABLE

NOTE: OFF (0); ON (1).

PB	LC	STATE	NEXT STATE	OUTPUT
0	0	OFF (UNPOWERED)	OFF (UNPOWERED)	0
0	**1**	**OFF (UNPOWERED)**		
1	0	OFF (UNPOWERED)	ON	1
1	1	OFF (UNPOWERED)	OFF (POWERED)	0
0	0	OFF (POWERED)	OFF (POWERED)	0
0	1	OFF (POWERED)	ON	1
1	**0**	**OFF (POWERED)**		
1	1	OFF (POWERED)	OFF (UNPOWERED)	0
0	0	ON	ON	1
0	1	ON	OFF (POWERED)	0
1	0	ON	OFF (UNPOWERED)	0
1	**1**	**ON**		

1b. [8 points] After completing the finite state machines with the missing transitions (found for question 1a), is the program source code consistent with the specification and finite state machines? If not, what changes would need to be made to the program to make it correct? (English sentences are fine for your answer.)

The program source code is **not** consistent with the specification and finite state machines. The program source code destroys the CeilingFan object when the power is cut; which results in the fan speed resets to 0 (off) when the power button is pushed again (create a new CeilingFan object). This is not what the specification and the fan-speed finite state machine are expected of the fan system: the fan system is supposed to resume the fan speed when the power button is pushed. In order to meet the specification requirements, the powerButtonPushed() function in the CeilingFanSystem class must be revised to set save the **fan object into a temp. object that is created outside the powerButtonPushed function and then set fan = null. When the else statement is run, set the fan object to the temp. To resume the previous fan speed.**

```
public CeilingFan temp = null;
public void powerButtonPushed() {
        if (isPowered) { temp= fan; fan = null; }
        else { fan = temp; }
        isPowered = !isPowered;
}
```

1c. [8 points] Specify a test suite that covers all transitions in the machines at least once. For each test case, list its sequence of inputs, its expected output (i.e., states after the inputs), and its pass/fail status as it would be if implemented as in the program source code as listed above. (In other words, *without* the fixes that you described in 1b.)

Example format:

test case 1: Inputs: pb, pb, sc, lc ; Expected output: fan off, light off; Status: pass

Inputs: <pb, sc, sc, sc, lc, pb> ; Expected output: fan off, light off, high, powered; Status: pass

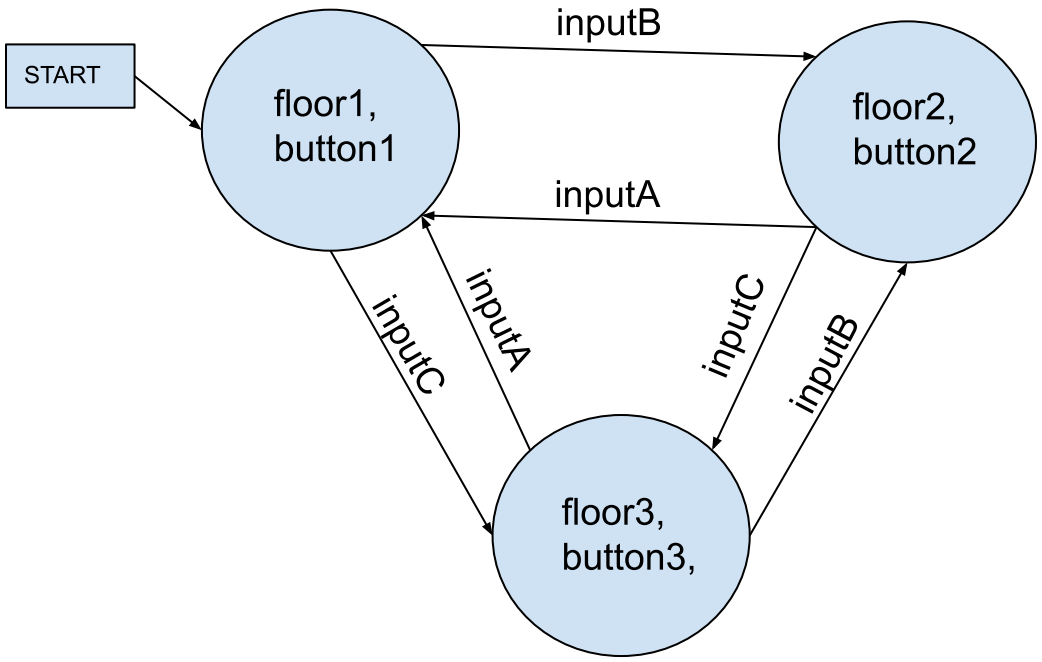
2. Finite state machine creation [12 points]

Specification:

The program is an elevator simulation program. The program simulates an elevator in a building that has three floors: floor1, floor2, and floor3. The elevator has three buttons, labeled "button1", "button2", and "button3", which correspond to the floor that the elevator should go next. When the button for a floor is pressed, the elevator should go to that floor. The elevator simulator starts with the elevator on floor1.

Note that for this problem, these three buttons are the only buttons (i.e., the only inputs), and we are only modeling the inside of the elevator, not the outside.

2a. [8 points] Draw the finite state machine to describe this elevator system.



2b. [4 points] Create a test suite that covers all transitions.

Format example:

test case 1: inputA, inputB, inputC

test case 2: inputD

test case 3: <inputA, inputC, inputB>

3. Combinatorial Testing [14 points]

(a) You have an iOS app that is “universal,” meaning that it should run on the iPhone 8, iPhone X, and iPad. The app works differently whether it is in the “online” mode or in the “offline” mode. And, additionally, the user can perform an in-app purchase to remove advertisements that appear. You want to test ***all combinations***. Complete and extend the table below to show all possible combinations that would need to be tested if exhaustive combinations are covered.

Device **iPhone 8, iPhone X, iPad**	**Internet connectivity** **(online, offline)**	**Revenue model** **(Ads, Purchase)**
iPhone 8	online	Ads
iPhone 8	offline	Purchase
iPhone 8	online	Ads
iPhone 8	offline	Purchase
iPhone X	online	Ads
iPhone X	offline	Purchase
iPhone X	online	Ads
iPhone X	offline	Purchase
iPad	online	Ads
iPad	offline	Purchase
iPad	online	Ads
iPad	offline	Purchase

12 combinations

(b) Now, let's say, you did the analysis above and decided that that was overkill: it is more testing than you have resources to perform and still have time to develop. You, instead, decide that you would like to perform ***pairwise*** testing of these features. Complete and extend the table below to show all pairwise tests that would be necessary (i.e., satisfies the pairwise criterion, but no more).

Device	Internet connectivity	Revenue model
iPhone 8	online	Ads
iPhone 8	offline	Purchase
iPhone X	online	Purchase
iPhone X	offline	Ads
iPad	online	Ads
iPad	offline	Purchase

6 combinations

(c) Now, let's say, you did the analysis above and decided that even pairwise testing was too much for your budget. You, instead, decide that you would like to test all features, but do not care about their combinations — in other words ***1-way*** testing. Complete and extend the table below to show all 1-way tests that would be necessary (i.e., satisfies the 1-way criterion, but no more).

Device	Internet connectivity	Revenue model
iPhone 8	online	Ads
iPhone X	offline	Purchase
iPad	online	Purchase

4. JUnit / Input partitioning / Black box testing [32 points]

In the CANVAS for Homework 2 there is a Java Jar file called TriangleType.jar. Download this file to your computer. This jar file contains a class called TriangleType, which contains a public static method called triangleType(). This method takes three integers as input and outputs an integer. The three input variables each describe the lengths of each side of a triangle. The lengths of the sides of the triangle should be less than or equal to 1000. The output of this method will be one of 5 possible values: 1 for a scalene triangle, 2 for an isosceles triangle, 3 for an equilateral triangle, 4 for values that do not describe a triangle, and 5 for values that are out of bounds.

Using Eclipse (or any IDE), create a new project. Under the Build Properties for the project, add the jar file as an external library. Create a new "JUnit Test Case." The new JUnit Test Case Class should be named TriangleTypeTest. The "Class Under Test" is TriangleType. Use JUnit4 (or JUnit5). You will be creating test cases for the method TriangleType.triangleType().

4a. [8 points] Describe, in English, how you would partition the input space for this program.

As defined in lecture 4.1, the **input space** of a program P consists of k-tuples of values that could be input to P during execution. **In order to partition the input space for the TriangleType class, we need to find all possible sets of the three sides: (Side1, Side2, Side3) where the lengths of the sides of the triangle should be less than or equal to 1000.**

4b. [8 points] For the partitions that you identified in 4a that are amenable to boundary-values, what are the boundary values that you can identify for each partition? For the partitions that you identify in 4a that are not amenable to boundary-values, what are the representative values for each partition?

Amenable: [0 <= (Side1 || Side2 || Side3) <= 1000] == [0, 1, 2, 3, . . ., 1000]

Not Amenable: [(Side1|| Side2 || Side3) < 0 AND/OR **(Side1|| Side2 || Side3) > 1000]**
==
[..., -3, -2, -1] AND/OR **[1000, 1001, 1002, … , n]**

4c. [16 points] Create at least one JUnit test method for each of the representative values and boundary values described in 4b. Ensure that your test methods do run, and that they pass or fail when appropriate.

Save the .java file and submit the TriangleTypeTest.java file and submit that file (by itself) separately to CANVAS from your written homework. Please do not include your entire Eclipse project — please just submit the TriangleTypeTest.java file. (In addition to the PDF for the written portion of this homework.)

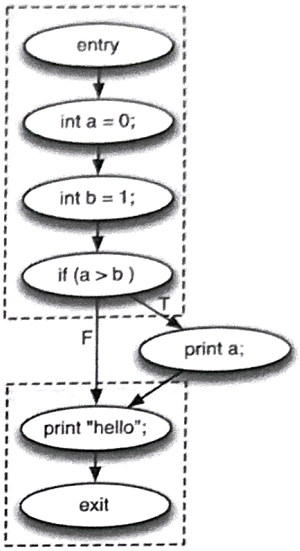
TriangleTypeTest.java file upload separately.

5. Control Flow Graphs [18 points]

For the next three programs, draw the control flow graph. Then, draw a rectangle around the basic blocks with a dotted line. In the nodes, put the source text. Draw all statements as nodes and represent the control flow for each node with the edges that flow would traverse if those nodes were reached. Format your answers like the following example.

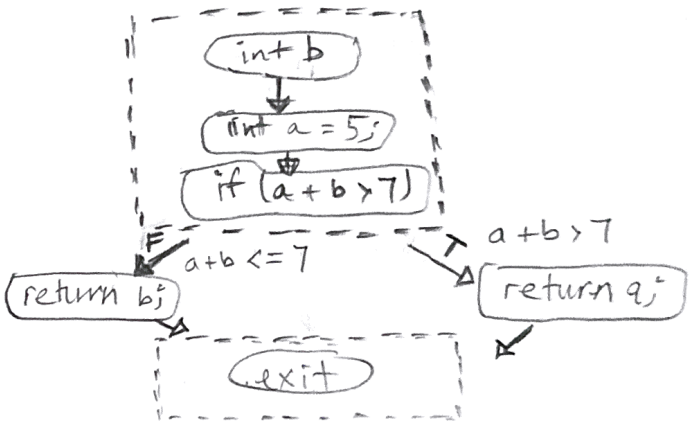
Example:

```
void w() {
      int a = 0;
      int b = 1;
      if ( a > b ) {
            print a;
      }
      print "hello";
}
```



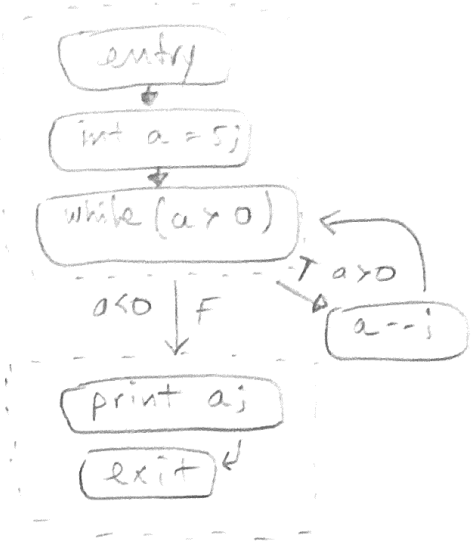
5a. [6 points]

```
int x(int b) {
      int a = 5;
      if (a+b > 7) {
            return a;
      } else {
            return b;
      }
      print a+b;
      return a+b;
}
```



5b. [6 points]

```
void y() {
      int a = 5;
      while (a > 0) {
            a--;
      }
      print a;
}
```



5c. [6 points]

```
void z(int x, int y) {
      if (x > 0) {
            for (int i = 0; i <= 5; i++) {
                  print i;
            }
      } else if (y > 0) {
            print y;
      } else {
            print 0;
      }
}
```