Mitchell Van Braeckel, 1002297, mvanbrae@uoguelph.ca

CIS*4150: Software Testing and Reliability
A2 - Specifications and State

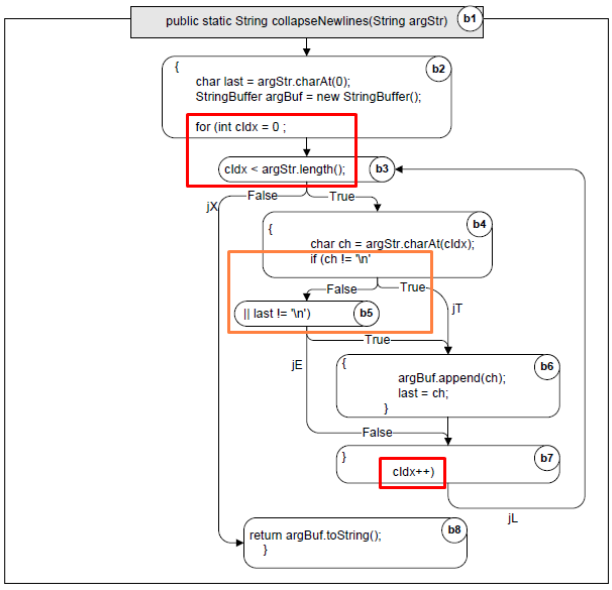
Part 1: Questions from the Book

Question 5.7

Can the number of basic blocks in the control flow graph representation of a program ever be greater than the number of program statements? If so, how? If not, why not?

The number of basic blocks in the control flow graph representation of a program can be greater than the number of program statements. Although a basic block typically coalesces adjacent, sequential statements of source code, in some cases a single syntactic program statement is broken across basic blocks to model control flow within the statement. We know that each node in a control flow graph is a basic block and these typically are equal to the number of program statements; however, we learned that sometimes single statements are broken into more than one node to model control flow within the statement (i.e. internal flow of control). For example, we have seen a for statement broken into its constituent parts of initialization, comparison, and incrementation, each of which appears in separate nodes in the control flow, thus causing the number of basic blocks to be greater than the number of program statements. Furthermore, complex predicates (eg. conditional logic seen in if statements such as OR (||), AND (&&)) can also be broken across basic blocks when modeling control flow since the next conditional within the logic gate may or may not be evaluated based on the result of the first. The two images below (that we've seen in the textbook and weekly course material) illustrate this example (red shows the for loop parts, and orange the conditional parts):

```
1   /**
2    * Remove/collapse multiple newline characters.
3    *
4    * @param String string to collapse newlines in.
5    * @return String
6    */
7   public static String collapseNewlines(String argStr)
8   {
9       char last = argStr.charAt(0);
10      StringBuffer argBuf = new StringBuffer();
11
12      for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
13      {
14          char ch = argStr.charAt(cIdx);
15          if (ch != '\n' || last != '\n')
16          {
17              argBuf.append(ch);
18              last = ch;
19          }
20      }
21
22      return argBuf.toString();
23  }
```

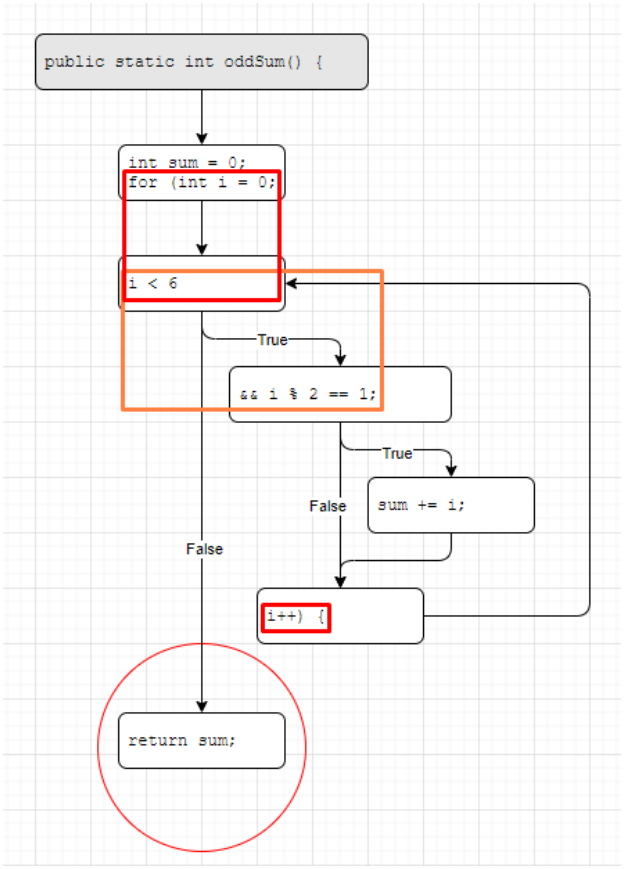


Furthermore, consider a function that returns the sum of odd integers between 0 and 5:

```
public static int oddSum() {
      int sum = 0;
```

```
        for (int i = 0; i < 6 && i % 2 == 1; i++) {
              sum += i;
        }
        return sum;
    }
```

We can see from its code that there are 4 program statements, whereas its control graph has 6 basic blocks (red shows the for loop parts broken across multiple nodes, and orange shows the logical if statement conditional parts broken across multiple nodes):



Part 2: State Machine

Considering the GPA library that we looked at in A1, evaluate the system, in terms of its ability to calculate a GPA for a year, assuming that we only consider 1 term, a student does not take more than 2 courses in a term, maximum of 3 students (which isn't directly relevant to the state machine), the only possible grades are: 'A', 'B', 'F' (without +/- decoration), and is applicable to both the Java and Python implementations.

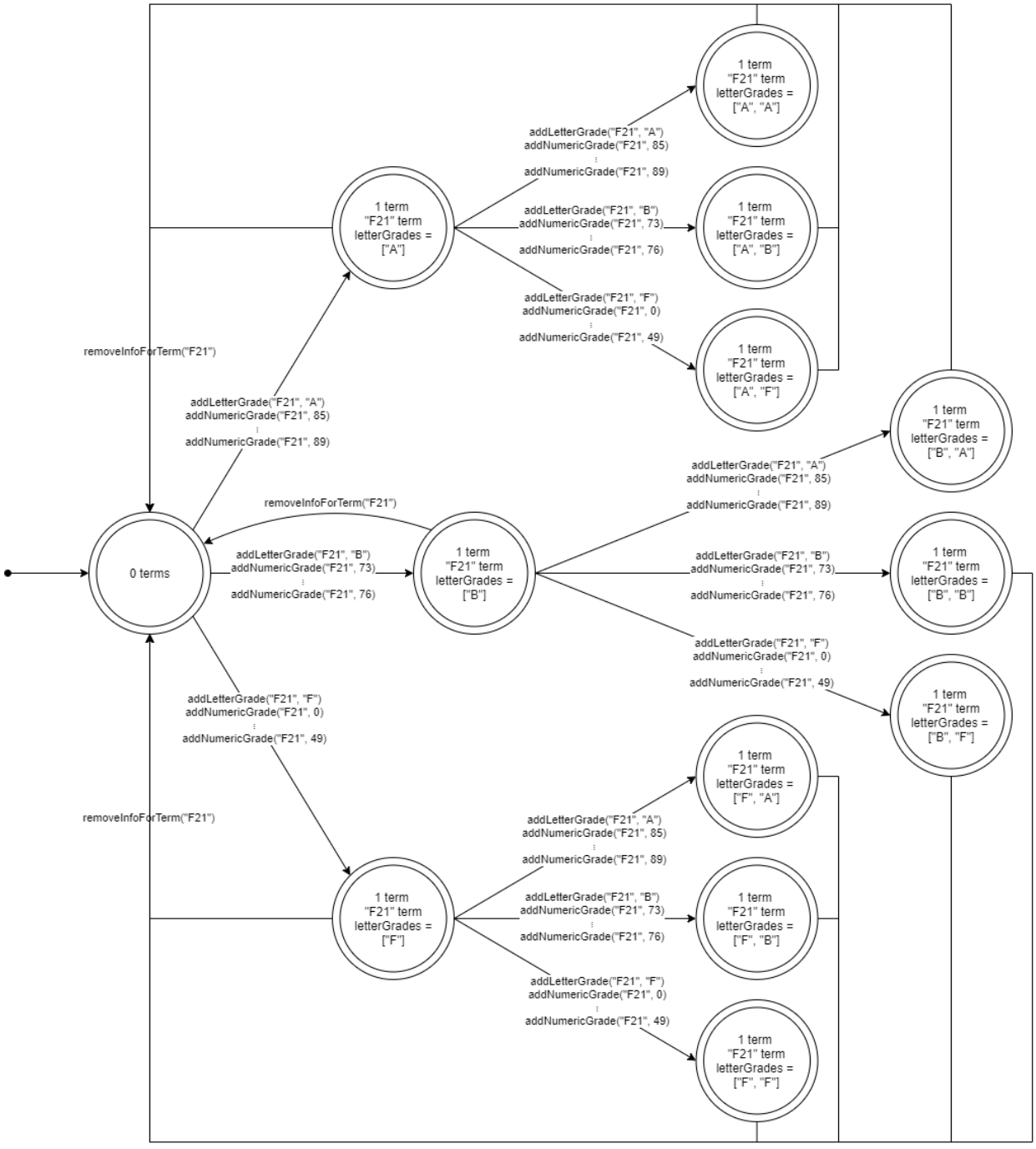
In the state machine based on Java, an actual parameter for `String term` of `addLetterGrade` and `addNumericGrade` is required when illustrating how a state may change, so I'll use the term "F21" as the one term we're considering for the student. Also note that instead of writing out all possible `addNumericGrade` function calls that would result in a change of state between nodes, I will add "..." or "⋮" to imply the other `int`

`percentageGrade` actual parameters that weren't listed in between the lowest and highest bounds of the respective letter grade. To be more specific, inclusive percentage grade ranges 85-89, 73-76, and 0-49 correspond to a letter grade of "A", "B", and "F" respectively.

In the state machine, it shows the attributes of a `GPAcalculator` instance that are applicable to its ability to calculate GPA for a term. These instance variables are `int nTerms` and `TermInfo[] terms`; and we will focus on the following instance variables of `TermInfo`: `String termName`, `int nCourses`, and `String[] letterGrades`. We know that an instance of `GPAcalculator` will have a student name and student ID, but these don't directly impact its ability to calculate GPA for a term. Similarly, we know that a `GPAcalculator` instance will have a `GPAconverterTool` when it's instantiated, but this also doesn't directly affect its state in terms of its ability to calculate GPA for a term. However, since this state machine diagram should be applicable to both the Java and Python implementations, we only need the `terms` array and that term's `termName` and `letterGrades` array. **Therefore, each node in the following state diagram will illustrate the number of `terms` as "X term(s)" and for each term it will show "*termName* term letterGrades = []"**.

Note: As stated by Professor Andrew Hamilton-Wright on Nov. 6, the method `removeInfoForTerm` was accidentally omitted from the Python version, so the state machine is based on the Java spec and will use this method.

Note: I will also include a separate image in the submission in case this one is too difficult to read.



Part 3.a: Testing

As we are only covering a single term based on our assumptions, the tests will also only attempt to calculate the GPA for one term. We will include one test to show that overall GPA is the same since there's only one term. Note that the tests will assume that we have already initialized a GPA calculator, there are no additional arguments to the test case function, expected GPA is based on officially sourced https://www.whatsmygpa.ca/ from

https://www.uoguelph.ca/uaic/faq/grades/how-do-i-calculate-gpa-using-my-guelph-grades, other expected results are based on inspection and documentation available, and all grades are equal credit weight. Also note that for each of the below test cases, I briefly go over the affected/related states with the "Covers state" list that starts at "0 terms" each time and transitions to other states based on the procedure steps (i.e. the code steps listed) of the test case. Unless otherwise stated, Python tests are the same as the Java tests.

- **Calculate term GPA when there are no terms**
    - Covers state: 0 terms

```
getTermGPA("F21")
```

    - Expected output: negative infinity (from documentation, we know that when the provided term is not found it should return negative infinity)
- **Remove a term when there are no terms**
    - Covers state: 0 terms

```
removeInfoForTerm("F21")
```

    - Expected output: error value (-1 in java, error in python | from documentation, we know that -1 is returned if term is not found in java | this will cause an error in python because this function was accidentally omitted in the python version)
- **Remove a term when there is one grade**
    - Covers state: 0 terms → 1 term → 0 terms

```
addLetterGrade("F21", "A")
removeInfoForTerm("F21")
```

    - Expected output: term is removed and there are no terms left (0 in java, error in python | from inspection, we know that the index of the term is returned when the term is removed in java | this will cause an error in python because this function was accidentally omitted in the python version)
- **Remove a term when there are two grades**
    - Covers state: 0 terms → 1 term → 2 terms → 0 terms

```
addLetterGrade("F21", "A")
addLetterGrade("F21", "B")
removeInfoForTerm("F21")
```

    - Expected output: term is removed and there are no terms left (0 in java, error in python | from inspection, we know that the index of the term is returned when the term is removed in java | this will cause an error in python because this function was accidentally omitted in the python version)

Note: For the above 3 test cases, more permutations and combinations can be created by replacing the letter grade being added.

- **Calculate term GPA when letter grades are ["A"]**
    - Covers state: 0 terms → A

```
addLetterGrade("F21", "A")
getTermGPA("F21")
```

    - Expected output: 4.0

- **Calculate term GPA when letter grades are [“B”]**
    - Covers state: 0 terms → B

```
addLetterGrade(“F21”, “B”)
getTermGPA(“F21”)
```

    - Expected output: 3.0
- **Calculate term GPA when letter grades are [“F”]**
    - Covers state: 0 terms → F

```
addLetterGrade(“F21”, “F”)
getTermGPA(“F21”)
```

    - Expected output: 0.0
- **Calculate term GPA when letter grades are [“A”, “A”]**
    - Covers state: 0 terms → A → AA

```
addLetterGrade(“F21”, “A”)
addLetterGrade(“F21”, “A”)
getTermGPA(“F21”)
```

    - Expected output: 4.0
- **Calculate term GPA when letter grades are [“A”, “B”]**
    - Covers state: 0 terms → A → AB

```
addLetterGrade(“F21”, “A”)
addLetterGrade(“F21”, “B”)
getTermGPA(“F21”)
```

    - Expected output: 3.5
- **Calculate term GPA when letter grades are [“A”, “F”]**
    - Covers state: 0 terms → A → AF

```
addLetterGrade(“F21”, “A”)
addLetterGrade(“F21”, “F”)
getTermGPA(“F21”)
```

    - Expected output: 2.0
- **Calculate term GPA when letter grades are [“B”, “A”]**
    - Covers state: 0 terms → B → BA

```
addLetterGrade(“F21”, “B”)
addLetterGrade(“F21”, “A”)
getTermGPA(“F21”)
```

    - Expected output: 3.5
- **Calculate term GPA when letter grades are [“B”, “B”]**
    - Covers state: 0 terms → B → BB

```
addLetterGrade(“F21”, “B”)
addLetterGrade(“F21”, “B”)
getTermGPA(“F21”)
```

    - Expected output: 3.0
- **Calculate term GPA when letter grades are [“B”, “F”]**
    - Covers state: 0 terms → B → BF

```
addLetterGrade(“F21”, “B”)
addLetterGrade(“F21”, “F”)
getTermGPA(“F21”)
```

    - Expected output: 1.5
- **Calculate term GPA when letter grades are [“F”, “A”]**
    - Covers state: 0 terms → F → FA

```
addLetterGrade(“F21”, “F”)
addLetterGrade(“F21”, “A”)
getTermGPA(“F21”)
```

    - Expected output: 2.0
- **Calculate term GPA when letter grades are [“F”, “B”]**
    - Covers state: 0 terms → F → FB

```
addLetterGrade(“F21”, “F”)
addLetterGrade(“F21”, “B”)
getTermGPA(“F21”)
```

    - Expected output: 1.5
- **Calculate term GPA when letter grades are [“F”, “F”]**
    - Covers state: 0 terms → F → FF

```
addLetterGrade(“F21”, “F”)
addLetterGrade(“F21”, “F”)
getTermGPA(“F21”)
```

    - Expected output: 0.0
- **Calculate overall GPA when letter grades are [“B”, “A”]**
    - Covers state: 0 terms → B → BA

```
addLetterGrade(“F21”, “B”)
addLetterGrade(“F21”, “A”)
getOverallGPA()
```

    - Expected output: 3.5
- **Calculate term GPA equals overall GPA when letter grades are [“F”, “A”]**
    - Covers state: 0 terms → F → FA

```
addLetterGrade(“F21”, “F”)
addLetterGrade(“F21”, “A”)
getTermGPA(“F21”)
getOverallGPA()
```

    - Expected output: The two calculated GPAs are equal (both are 2.0)

Note: For all tests, more permutations and combinations can be created by replacing the addLetterGrade with its respective addNumericGrade with percentage grade value within the proper range. To be more specific, inclusive percentage grade ranges 85-89, 73-76, and 0-49 correspond to a letter grade of “A”, “B”, and “F” respectively.

Part 3.b: Testing Source Code

Please see attached code files for tests written based on the above design. Specifically, I have modified the `RunTests.java` and `buildJavaTests.sh` files to work with the `TestGPACalculations.java` that I created to write tests for A2 based on my state diagram and the designed test cases above (I added some extra too). I will submit these within my

previous A1 submission so that the tests can be run the same way as indicated in the README for A1 (note that the README does not contain specific info pertaining to A2). This way, you should be able to more easily run the tests if you wish to.

To summarize the test results, all actual outputs matched the expected outputs with a few exceptions. When a term is not found, the result is positive infinity instead of negative infinite. Due to a problem with converting percentage grades to letter grades, any `addNumericGrade` call that has an actual parameter equal to the lowest value in a letter grade's range will result in a GPA of positive infinity. The above summary was for the Java tests I created. There is a problem with creating Python tests because the class name is the same as the module name so I am unable to initialize a gpa_calculator instance, so these tests have been excluded from the submission.

Part 4: Overall Summary of Implementation Differences

Overall, there weren't many differences between the Java and Python implementations since they used the same interface and method signatures for the most part. It's important to note that the `removeInfoFromTerm` method does not exist in the Python version. As stated by Professor Andrew Hamilton-Wright on Nov. 6, the method `removeInfoForTerm` was accidentally omitted from the Python version, so the state machine is based on the Java spec and will use this method and any Python test that attempts to use this method will fail. Therefore, the tests designed won't work in python. Ideally, we would have another way of accessing and removing a term; however, it does not exist in the Python version so in order to do and test this, we would have to manually manipulate the instance. Furthermore, although the GPA calculator class of both Java and Python have similar states; they differ subtly. For instance, in Java there exists an additional attribute `int nTerms` and in Java's `TermInfo`, there exists additional attributes `int nCourses` and `float[] percentageGrades` (note: in this case, when I say attributes, I mean instance variables). In addition, the constants in Java for max courses per term and max terms per student are 5 and 4, whereas in Python they're respectively 6 and 31.