

CS5550 - Phase C



Khan, Hassan

Raphael, Samuel

Ma, Yichuan Philip

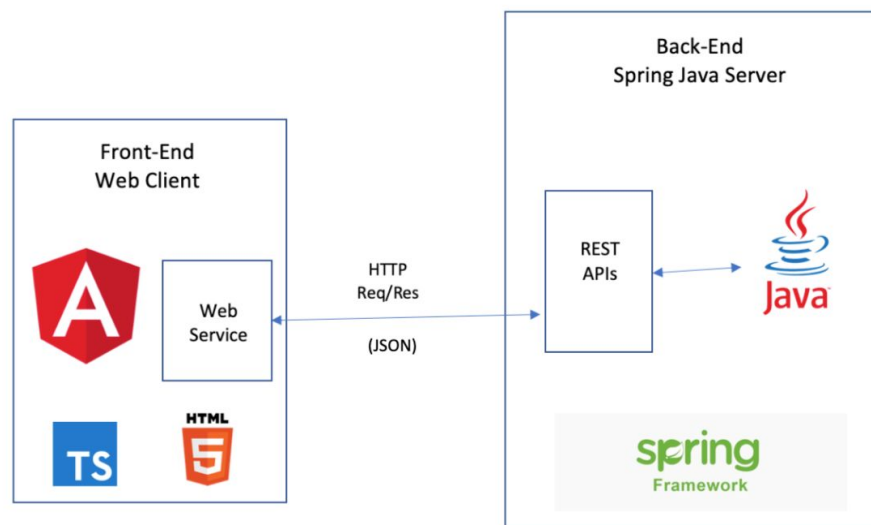
Mousset, Pierre-Alexandre

I - Plagus Introduction

Plagus is a code plagiarism checker for Python 3 and JavaScript (ES6), written in Java. Originally, we selected TypeScript and Python as programming languages. However since ANTLR did not have an official grammar for TypeScript, we chose to go with JavaScript. The functionalities that we were able to detect were the following:

- Exact same code (LCS)
- Comments plagiarism
- Variable/function renaming
- AST Node Type Counter

II - System architecture



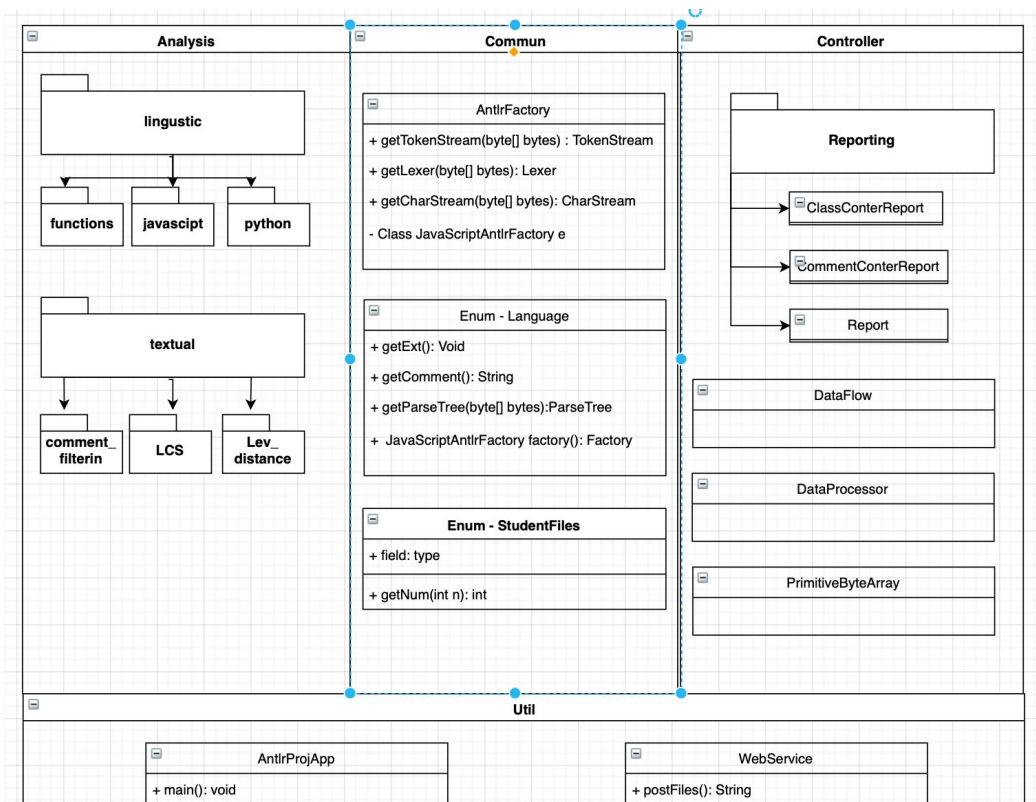
Front-End Architecture

For our front-end, we built a web application using the Angular framework (HTML, CSS, TypeScript). The main logic of the front-end (plagus-frontend in Project_src) is in the app.component.ts file (this is where we use TypeScript to control the logic in the view). The main HTML to present to the web DOM is in the app.component.html file. The front-end uses the services/WebService.ts file to communicate with our back-end Spring server. The front-end and back-end communicate through HTTP protocols, through a RESTful API. Based on the results received from the back end, the front-end displays results on LCS analysis, Comment Analysis, AST Node Type Counters, and Function Analysis

Back-End Architecture

For our back-end, we used the Spring framework to allow our Java back-end to be able to communicate with our front-end, thus communicate with the user. We accomplish this by building a RESTful API, that can be reached by incoming HTTP POST requests. This API is also where we receive files from the front-end (2 zip files, 1 for each set) and where we send a JSON response back to the server where the results are presented. We handled I/O from the client by reading in data from Java's NIO1/2 tools, including the ability to create virtual Zip filesystems, and traverse them with the Java Streams API. For response data, we used the Gson library, and due to our use interfaces like Path and Token, had to create our own serialization/deserialization objects.

III- Back-End Design



Our main plagiarism detection logic implementations are tucked away in our analysis subpackage that is part of our model. This subpackage contains two more subpackages: linguistic and textual. As their names suggest, they represent linguistic and textual analysis, respectively.

The linguistic subpackage contains lexer, parser, base listener, and base visitor classes for Python 3 and JavaScript. These classes were automatically generated by ANTLR when we generated parsers from the G4 grammar files for the respective languages that we are targeting. Additionally, the

linguistic subpackage contains custom listener classes that extract functions from Python 3 source code files and rename all parameters and variables, including the function name, in the function bodies into canonical names, along with a class that represents the metadata for a function.

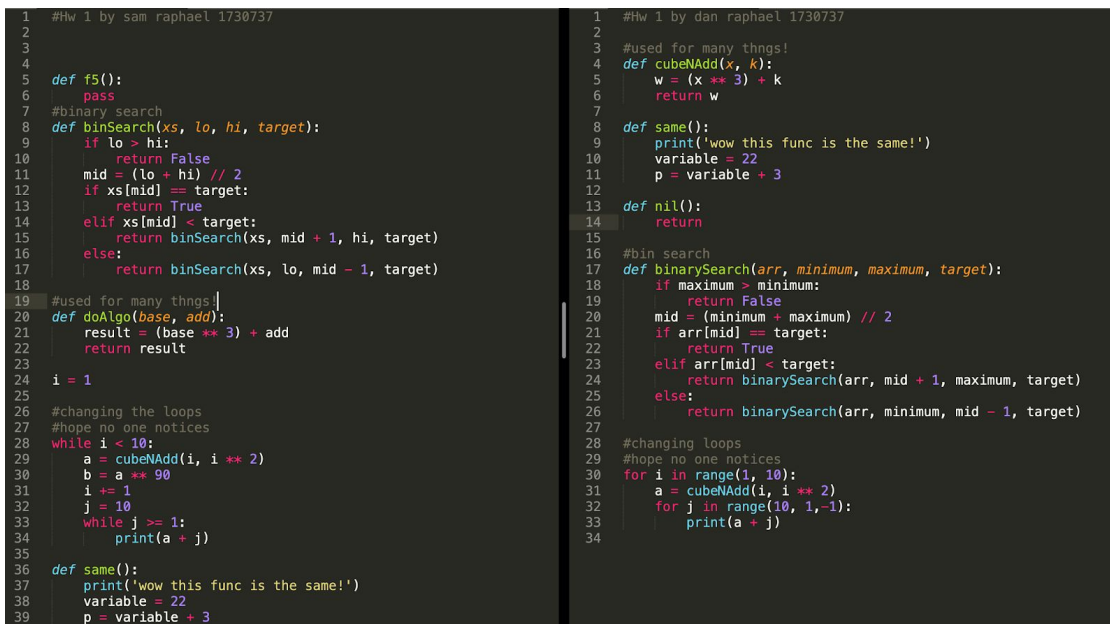
The textual subpackage contains implementations for comment filtering, LCS, and Levenshtein distance (for comment and function analysis). LCS, or Longest Common Subsequence, was used to detect partial similarities between two student submissions. Our implementation of the Levenshtein distance function is used to find closeness between any two functions from the students' submissions that are suspected to be similar to each other. These can determine closeness between submissions rather than exact similarities, so that we can calculate the percent similarity values between them, like any good plagiarism detection tool would.

IV - Algorithms

Textual:

- Longest Common Substring: One of the easiest ways to detect similarities between two set of programs is to compare the code. We implemented an algorithm that is able to find the largest sequence of code that is present in both programs in the same order. We designed that algorithm to work with two different lists of tokens. The Token has multiple useful properties such as `getLine()` and `getType()`. The `getType()` method allows us to filter some of the Tokens in order to increase the accuracy of our analysis. For both languages, he filtered out white spaces and new lines.

Example:



```
1 #Hw 1 by sam raphael 1730737
2
3
4
5 def f5():
6     pass
7 #binary search
8 def binSearch(xs, lo, hi, target):
9     if lo > hi:
10         return False
11     mid = (lo + hi) // 2
12     if xs[mid] == target:
13         return True
14     elif xs[mid] < target:
15         return binSearch(xs, mid + 1, hi, target)
16     else:
17         return binSearch(xs, lo, mid - 1, target)
18
19 #used for many thngs|
20 def doAlgo(base, add):
21     result = (base ** 3) + add
22     return result
23
24 i = 1
25
26 #changing the loops
27 #hope no one notices
28 while i < 10:
29     a = cubeNAdd(i, i ** 2)
30     b = a ** 90
31     i += 1
32     j = 10
33     while j >= 1:
34         print(a + j)
35
36 def same():
37     print('wow this func is the same!')
38     variable = 22
39     p = variable + 3
```

```
1 #Hw 1 by dan raphael 1730737
2
3 #used for many thngs!
4 def cubeNAdd(x, k):
5     w = (x ** 3) + k
6     return w
7
8 def same():
9     print('wow this func is the same!')
10    variable = 22
11    p = variable + 3
12
13 def nil():
14     return
15
16 #bin search
17 def binarySearch(arr, minimum, maximum, target):
18     if maximum > minimum:
19         return False
20     mid = (minimum + maximum) // 2
21     if arr[mid] == target:
22         return True
23     elif arr[mid] < target:
24         return binarySearch(arr, mid + 1, maximum, target)
25     else:
26         return binarySearch(arr, minimum, mid - 1, target)
27
28 #changing loops
29 #hope no one notices
30 for i in range(1, 10):
31     a = cubeNAdd(i, i ** 2)
32     for j in range(10, 1, -1):
33         print(a + j)
34
```

Analysis result:

Choose ZIP File 1: p1.zip

Choose ZIP File 2: p2.zip

Choose File Type: ☒ Python ☐ JavaScript

Report

LCS Analysis

Comment Analysis

AST Analysis

Function Analysis

LCS Analysis

Set 1	Set 2
<pre>defsame():print('wow this func is the same!')variable=22p=variable+3</pre>	<pre>defsame():print('wow this func is the same!')variable=22p=variable+3</pre>
Lines: 31 - 34	Lines: 6 - 9

- Comment Analysis: We discovered that when people plagiarise code, they often forget to change the comments. Hence, we have a functionality that parses all the comments in both sets of files, and does an n^2 comparison between all comments. We use Levenshtein distance (which measures the closeness between two strings) to determine if comments are similar. We have the capability to parse multi line comments as well. We use a levenshtein distance of 5 to determine if two comments are similar.

Comment Analysis Example:

Plagus

Choose ZIP File 1: p1.zip

Choose ZIP File 2: p2.zip

Choose File Type: ☒ Python ☐ JavaScript

Report

[LCS Analysis](#)

[Comment Analysis](#)

[AST Analysis](#)

[Function Analysis](#)

Comment Analysis

Set 1	Set 2	Levenshtein Dist.
#Hw 1 by sam raphael 1730737	#Hw 1 by dan raphael 1730737	2
#binary search	#bin search	3
#used for many thngs!	#used for many thngs!	0
#changing the loops#hope no one notices	#changing loops#hope no one notices	4

Structural

To rename all variables in a function body, including the function name, we do the following:

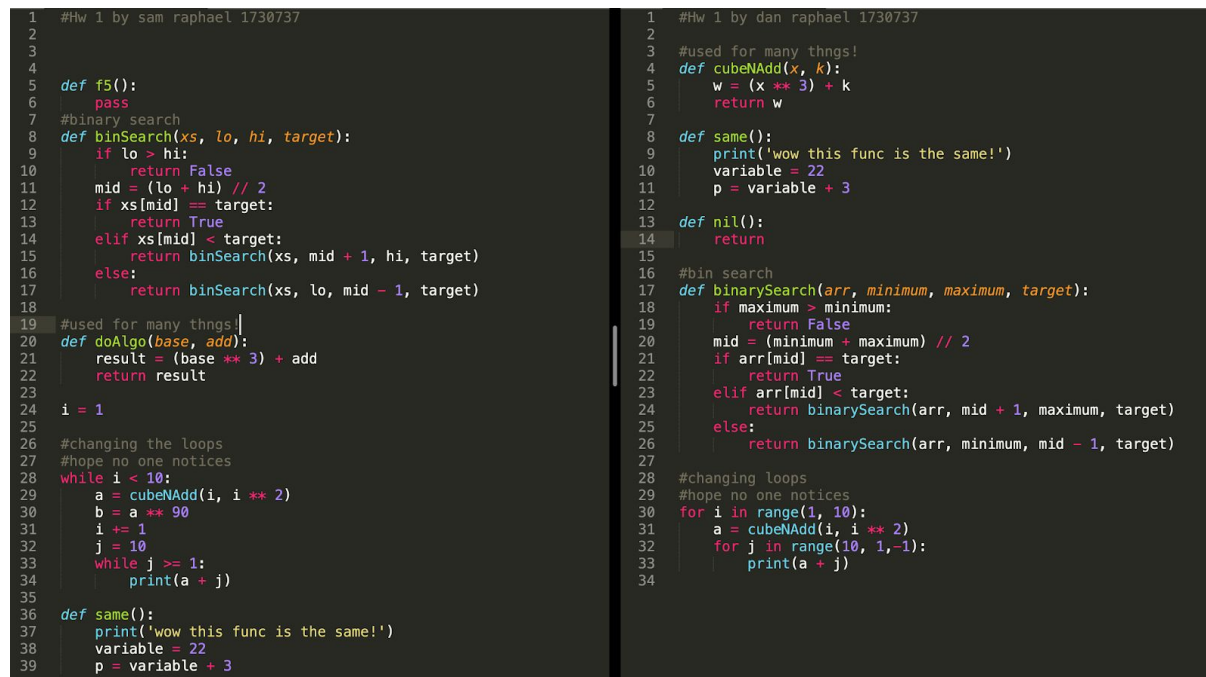
- Gather all tokens from a source code file using the ANTLR-generated Lexer class.
- Make a parse tree using the ANTLR-generated Parser class from the Lexer's tokens.
- Use the custom-made Function Listener to extract function bodies from source code through a single walk of the parse tree, by placing each function definition context (`Python3Parser.FuncdefContext`) object into an instance of the function metadata class `FuncData`.
- For each function extracted, walk through its context, also a parse tree, using the custom-made Variable Listener to rename all variables into canonical names (i.e. `var1`, `var2`,...), including the function name (in case of recursive functions), and generate a new function body string.
- Use the newly canonized functions to detect plagiarism amongst the submissions using Levenshtein distance.

More specifically, we implemented two custom listeners that extended the base listener class generated by ANTLR (`Python3BaseListener` for Python 3). The base listener class has a couple of methods, an enter rule method and an exit rule method, for each and every parser rule in a language's grammar (G4) file, and a programming language's grammar file has a lot of parser rules as well as lexer rules. Additionally, it has methods for terminal nodes (i.e. individual tokens) along with one for error nodes.

The first custom listener class that we implemented simply stores function declarations in a list. It only overrides one method from the original superclass, one for entering a function definition rule, and that method simply makes a new function metadata object with the function definition context object and puts the metadata object into the list. It also has a getter method for the list of metadata objects. This listener is run for an entire source file, and the function definition method is called for every function definition that it encounters.

The second custom listener class that we implemented takes in a reference to a StringBuilder and builds the canonized version of the function. It contains a map that maps the original variable name to its canonized name ($x \rightarrow \text{var0}$, $y \rightarrow \text{var1}$, etc.). Variables encountered for the first time by the listener are put into the map, along with the function name and all of its parameters in a function definition. For every terminal node encountered, if it is a variable expression or function call, the canonized name is appended into the string builder. Otherwise, the original token is appended into the string builder. There is one listener for every function in a source file. This is what we use to make canonized function bodies to detect variable renaming and function definition swapping.

Example (notice the binSearch and binarySearch functions):



```
1 #Hw 1 by sam raphael 1730737
2
3
4
5 def f5():
6     pass
7
8 #binary search
9 def binSearch(xs, lo, hi, target):
10     if lo > hi:
11         return False
12     mid = (lo + hi) // 2
13     if xs[mid] == target:
14         return True
15     elif xs[mid] < target:
16         return binSearch(xs, mid + 1, hi, target)
17     else:
18         return binSearch(xs, lo, mid - 1, target)
19
20 #used for many thngs!
21 def doAlgo(base, add):
22     result = (base ** 3) + add
23     return result
24
25 i = 1
26
27 #changing the loops
28 #hope no one notices
29 while i < 10:
30     a = cubeNAdd(i, i ** 2)
31     b = a ** 90
32     i += 1
33     j = 10
34     while j >= 1:
35         print(a + j)
36
37 def same():
38     print('wow this func is the same!')
39     variable = 22
40     p = variable + 3
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

```
1 #Hw 1 by dan raphael 1730737
2
3
4 #used for many thngs!
5 def cubeNAdd(x, k):
6     w = (x ** 3) + k
7     return w
8
9
10 def same():
11     print('wow this func is the same!')
12     variable = 22
13     p = variable + 3
14
15
16 def nil():
17     return
18
19
20 #bin search
21 def binarySearch(arr, minimum, maximum, target):
22     if maximum > minimum:
23         return False
24     mid = (minimum + maximum) // 2
25     if arr[mid] == target:
26         return True
27     elif arr[mid] < target:
28         return binarySearch(arr, mid + 1, maximum, target)
29     else:
30         return binarySearch(arr, minimum, mid - 1, target)
31
32
33 #changing loops
34 #hope no one notices
35 for i in range(1, 10):
36     a = cubeNAdd(i, i ** 2)
37     for j in range(10, 1, -1):
38         print(a + j)
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Analysis result:

Submit Files

Report

LCS Analysis

Comment Analysis

AST Analysis

Function Analysis

Function Analysis

Set 1	Set 2	Levenshtein Dist.
<pre>defvar0(var1,var2,var3,var4): ifvar2>var3: returnFalse var5=(var2+var3)//2 ifvar1[var5]==var4: returnTrue elifvar1[var5]<var4: returnvar0(var1,var5+1,var3,var4) else: returnvar0(var1,var2,var5-1,var4)</pre>	<pre>defvar0(var1,var2,var3,var4): ifvar3>var2: returnFalse var5=(var2+var3)//2 ifvar1[var5]==var4: returnTrue elifvar1[var5]<var4: returnvar0(var1,var5+1,var3,var4) else: returnvar0(var1,var2,var5-1,var4)</pre>	2
<pre>defvar0(var1,var2): var3=(var1**3)+var2 returnvar3</pre>	<pre>defvar0(var1,var2): var3=(var1**3)+var2 returnvar3</pre>	0
<pre>defvar0(): var1('wow this func is the same!') var2=22 var3=var2+3</pre>	<pre>defvar0(): var1('wow this func is the same!') var2=22 var3=var2+3</pre>	0

Abstract Syntax Tree Data Analysis

We created what we referred to as a Class Counter, or more accurately, a count table of different types of AST nodes. By itself, it requires some scrutiny, but can be useful. How many documents have 134 and 130 expressions respectively as well as the same number of for loops?--a question we can answer. This process obviously needs to be automated, and much more in-depth analysis is possible with this type of data, but we were not able to get to that, just display the counts. We provided an example in class in which the number of `for` loops in the counter was 2, while the number of `while` loops was 0. This was compared against a document in which the reverse was true. In general one thing we could have done would have been to compare the sum of the types of loops--compare the sum of switches and ifs between two documents or sets of documents.

Example:

Choose File Type:
☒ Python
☐ JavaScript

Submit Files

Report

LCS Analysis
Comment Analysis
AST Analysis
Function Analysis

AST Analysis

Rule	Set 1 Value		Set 2 Value	
	sort asc	sort desc	sort asc	sort desc
Exprlist		0		2
For stmt		0		2
Augassign		1		0
File input		1		1
Pass stmt		1		0
While stmt		2		0
Subscriptlist		2		2

V - Software Engineering Practices

We have an extremely heterogeneous team for that project: some of our teammates are extremely comfortable with software development concepts while others wrote their first line of code a few months ago. Those disparities not only increased the complexity of the project, but also provided opportunities for less experienced teammates to learn from mature developers while experienced developers develop their communication skills.

Development Practices:

- Pull Requests: Team members worked on their feature branches. When ready Pull Requests would be created and viewers would be assigned. Once approved, the Pull Request would be merged into Master
- Scrum meetings: Daily stand-up through Slack/Google Hangouts/In-person
- Deadlines: goals are a great way to deadlines helps you to get things actually done.
- Pair Programming: Some team members were more skilled than others, hence pair programming allowed us to better tackle tasks.

Communication:

- We learnt that communication is the key in any software development project when Pierre implemented the Longest Common Subsequence instead of Longest Common Substring. By using too many acronyms and lack of communication, we lost time during the first half of November. However through better communication, we were able to better resolve any conflicting interests and interpersonal problems.

We use a few builders: one for the LCSMiniReport, in which we require a Gson serializable object with over 8 different values. Due to the volume of data to be packed in to the object, a builder seemed to be an appropriate choice. We used another builder pattern for the report, but it turned into not just a builder, but a good debugging tool. When testing a single report, I only had to call on the build method relevant for that. For the Report, we use a public interface to build, implementing a private class type for returning it in a static method. It is debatable whether this technique should have been used or not if the LCSMiniReport. It has so many setter methods that, for a class with no computation, holding very discrete and unchanging data, probably should not have reached over 300 lines. However, due to the minimal size of the actual report, and the nature of builders tending towards high coupling, we left them together.

We have a dual-layer abstract factory for all things ANTLR. First, we quite literally have an abstract factory, common methods in the ANTLRFactory, and abstract methods implemented in the Python and JavaScript versions. These however, are not used publicly. They are package-private and supplied as return values of instance methods in the Language enum method factory(). Not only are the factories abstract, they are also lazy singletons.

The combination of ParseTree objects and Listeners are a form of visitor--just with predetermined visit routes. This design allowed us to exploit the hierarchy of the tree first to find functions, and then to rename them. We tried using visitors, but they require a more fleshed out understanding of the grammar, as well as providing an extra thing to keep track of and test--which we did not require.

VI - Testing

Coverage: Plagus Testing ×

86% classes, 74% lines covered in package 'com.plagus_server'

Element	Class, %	Method, %	Line, %	Branch, %
analysis	89% (25/28)	68% (95/1...	72% (391/...	77% (53/68)
common	100% (7/7)	95% (21/22)	96% (31/32)	100% (0/0)
controller	88% (8/9)	71% (27/38)	83% (146/...	66% (4/6)
util	75% (3/4)	17% (3/17)	38% (13/34)	100% (1/1)
AntlrProjApplication	0% (0/2)	0% (0/1)	0% (0/2)	100% (0/0)
webService	100% (1/1)	100% (1/1)	85% (12/14)	0% (0/2)

VII - Feedback Homework 5

Feedback received	Action we took
Lack of documentation and comments	Spent half a day working on documentation
Lack of testing	We assigned one member of the team to do testing
Poor project structure	We did some major design changes between the phase B and phase C
Meaningless variables and classes name	When we worked on documentation after we received the feedback, we also renamed few variables and classes
Had source files outside of src	All files outside of src have been removed

IIX - References

LCS algo: <https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/>

LCS algo: <http://oaji.net/articles/2017/1948-1513933134.pdf>

Python lexer: <http://www.dabeaz.com/ply/ply.html>

ANTLR tutorial: <https://tomasetti.me/antlr-mega-tutorial/>

ANTLR Book: <https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference>

Levenshtein Distance: <https://www.baeldung.com/java-levenshtein-distance>

MOSS Plagiarism Detector Implementation: <https://theory.stanford.edu/~aiken/moss/>

Inspect AST: <https://javaparser.org/inspecting-an-ast/>

AST implementation: https://github.com/fyrestone/pycode_similar

Algorithms: http://www.cse.psu.edu/~sxz16/papers/ISSTA_alg_plagrism.pdf

A comparison of code similarity analysers :
<https://link.springer.com/article/10.1007/s10664-017-9564-7>

Book: Gang of Four (GOF), 1994