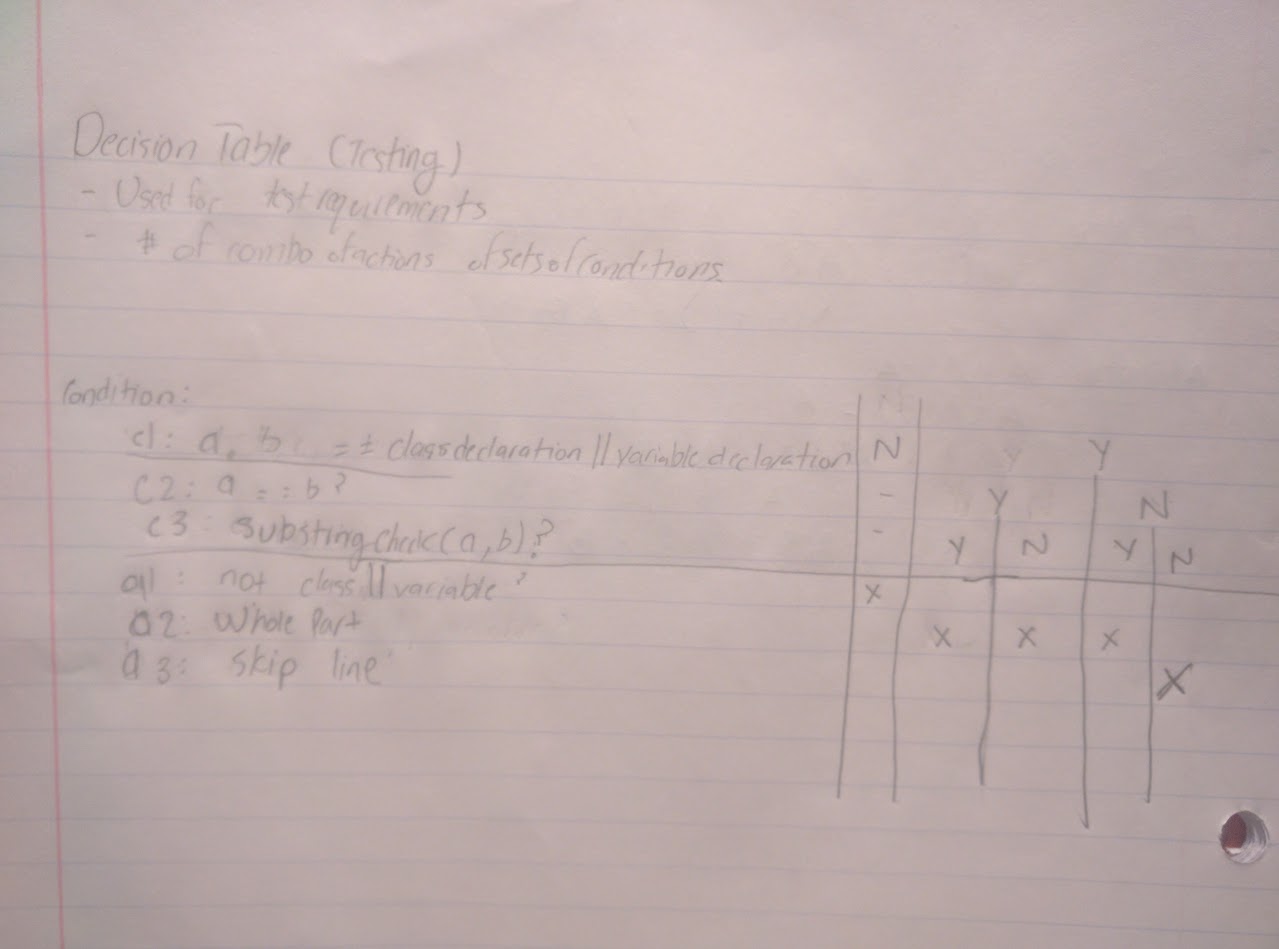
**Whole-Part Check (Done by Jomar Dimaculangan)**

This check contains a substring checker which is the main function used when traversing the detailAST tree. With this in mind, I tested the substring checker with different variables for valid and invalid parameters, so I could create a black-box test – Decision Table.

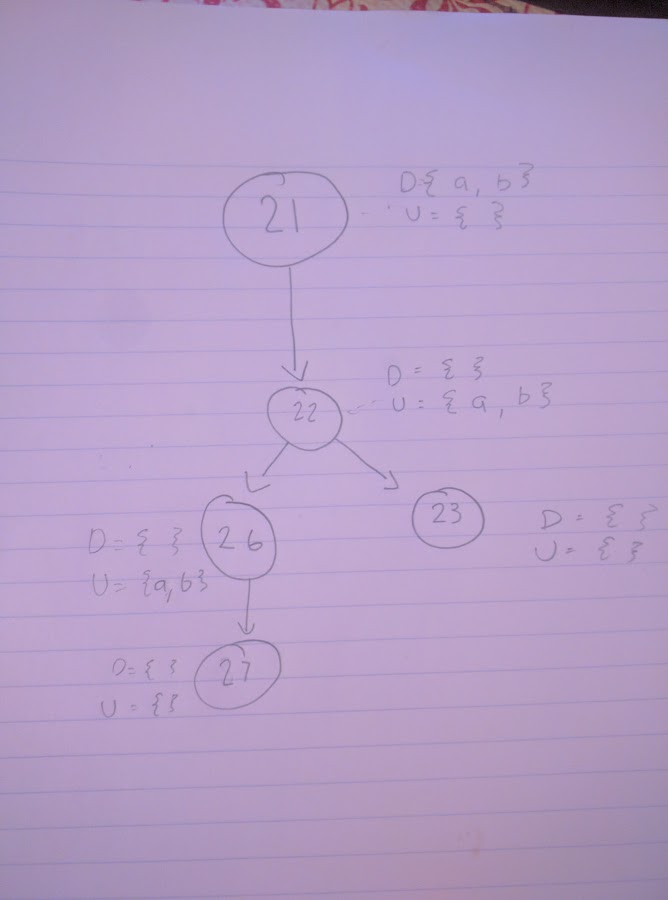
subStringChecker() Method:

Everytime visitToken() is called, subStringChecker is used whenever a manmade class, variables, loops, conditions, etc. declaration is used in the java file. This method will then be used for both my black-box and white-box testing.

For **black-box** testing, I created a decision table because it decision tables lists out which situations will cause the check to be invalid, or valid, without really looking at the code. Also, according to Prof. Venera, it’s an ideal for describing situations where number of combinations of actions are taken under varying sets of conditions. This is exactly the case for Whole-Part check. As I said earlier, inside visitTokens(), we check to see if the childAst is a declaration. If it isn’t, the condition false, or error. Now we check if the declaration is already used before. Let’s call the first class declaration A and the second declaration (whatever it may be) B. One case would be that B is exactly the same as A. Although the system might not let this happen, its also a valid Whole-Part Check. For this case, A==B, this will return True and both will be highlighted as Ambiguous or Whole-Part. We also don’t even have to check if B is a substring of A because it is always valid that B is a substring of A if they are equivalent. Now let’s say that A != B. We can’t really tell that this is valid or invalid whole-part check yet because we have to make sure that B is a substring of A. So we go one step further and check if B is a substring of A. If subStringChecker(a,b) is true, we can say that it’s a whole part. Otherwise, it doesn’t pass any of the decision table and we skip the line.

Below is the decision table I made which covers the description above:

For my **white-box** testing, I followed the three step procedure we were taught during lectures. I created a **CFG** or a control-flow graph which covers loops, if-else statements (conditions), cases, and recursion. We also were taught to list out definitions and uses in each node. In my case, the node number I used is the same line number I used inside the code. Below is the CFG for my subStringChecker() method with test cases.



The second step of white-box testing is creating a **def-use** table. We will provide all of C-uses and P-uses inside this table. This is to see which variables are used to decide if predicates are true or when a variable is used to compute other variables. We will also use this table to see if the coverage is sufficient enough for our test.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Word | Line 21 | Line 22 | Line 23 | Line 24 | Line 25 | Line 26 | Line 27 | Line 28 |
| A | 22,26 | - | - | - | - | - | - | - |
| B | 22,26 | - | - | - | - | - | - | - |

A and B are both declared in line 21 and only in line 21. This would mean that we would list out whenever it is being used, which is the conditional “if(a is in b)” in line 22 and else{}, in line 26.

The third and final step of **white-box** testing is creating **test-case**s and seeing if all paths are covered. For the test-cases, we will input valid strings as a and b in the function subStringChecker(string a, string b). One test case set that would satisfy “all-uses” criterion would need to hit line 21, 22, and 26. The first test case would be a = “StringList” and b=”String”. This would hit line 22 but not 26. In order to cover 26, we would need a test case that returns false and goes to the else{} statement. In this example, we’ll use a = “notSubString” and b = “wholepart”. Below is the table that represents this data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | A | B | Def use case for A | Def use case for B |
| T1 | “StringList” | “String” | <21,22> | <21,22> |
| T2 | “notSubString” | “wholepart” | <21,26> | <21,26> |

**Overloaded-Identifier Check (Done by Jomar Dimaculangan)**

The overloaded-identifier check uses the same function as Whole-Part function, subStringChecker(). For the sake of avoiding redundancy, look back at the Whole-Part function section for the black-box and white-box testing for this function.

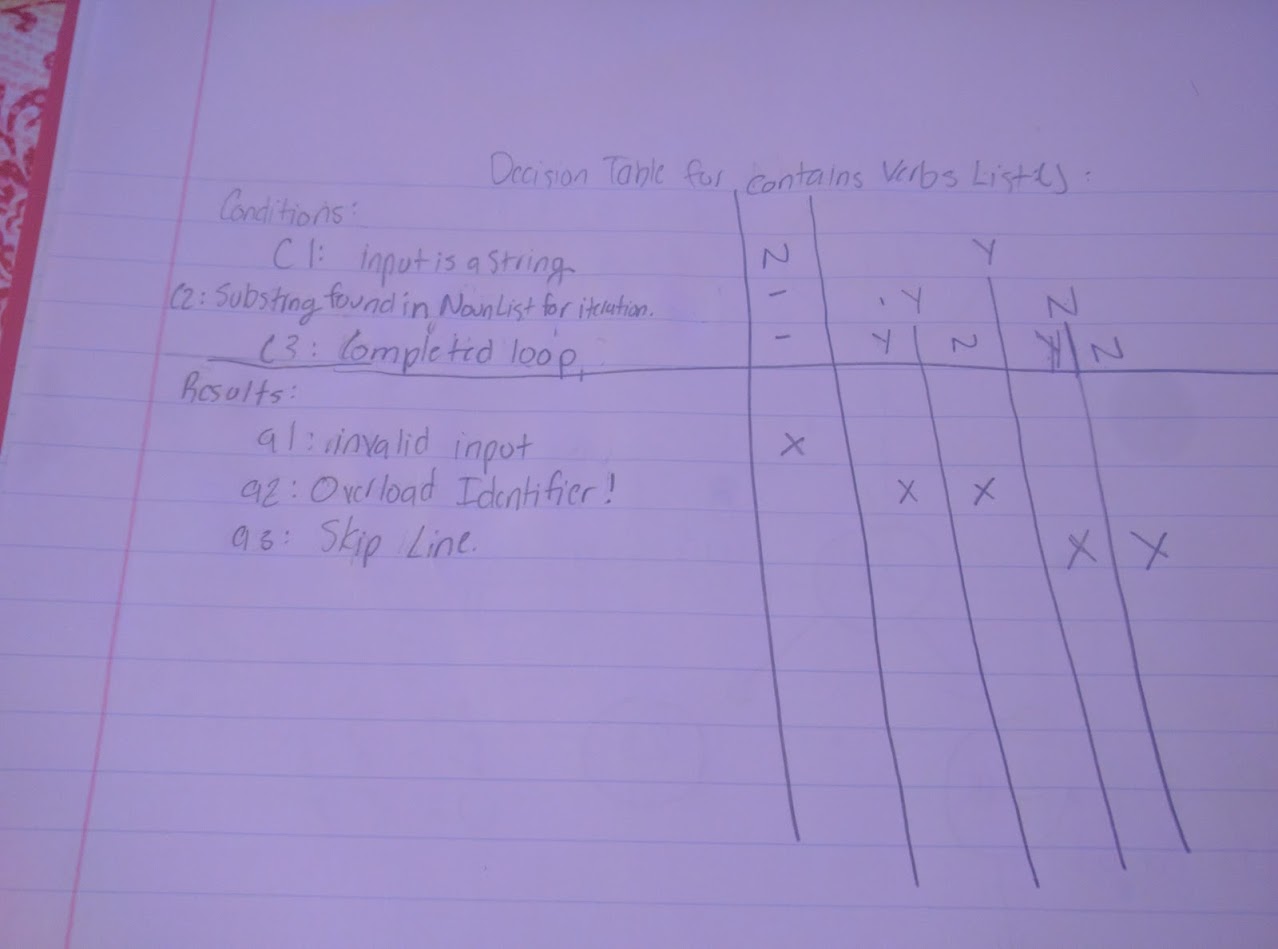
The difference between Whole-Part and Overloaded-Identifier white-box and black-box testing is that we have to check if a string is inside a noun or verb list. For this function, I created a containsVerbsList() and containsNounList() for black-box and white-box testing. \*DISCLAIMER: I was able to populate the whole noun list in my local device by putting my whole path name …/bin/etc/nounlist.txt BUT EVERY SYSTEM is different so I manually put in commonly used nouns and verbs for the sake of the checkers.

populateNounList() and populateVerbList():

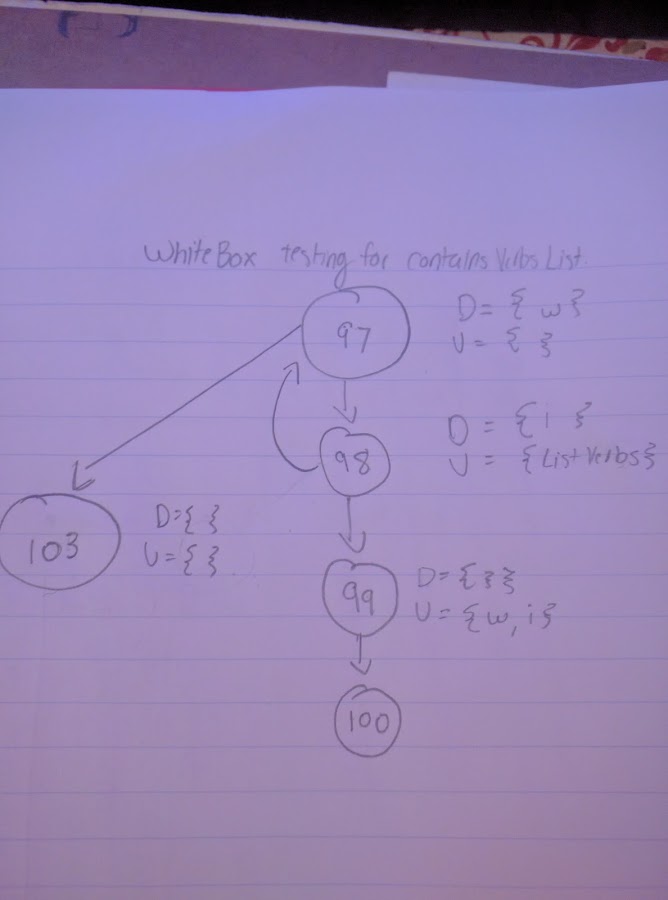
Since this doesn’t take any parameters and is always just populates the list with commonly used nouns and verbs, depending on what list it is on. Because there are no parameters and the behavior is always the same, black-box and white-box testing wouldn’t make an impact. To test if the inputs correct, however, is inside the unitTests in deliverable 1.

containsVerbsList() and containsNounsList():

Since these methods are very similar, I’ll be showing a black-box test for containsNounList() and white-box test for containsVerbsList(). The only difference is that containsNounsList() checks to see if a string is a substring of a Noun and containsVerbsList() checks to see if a string is a substring of a Verb inside their respective list. The parameters for both is a string. In the check’s case, the string is found while traversing inside the visitToken() as either a variable name, or a class name.

 For the black-box test, I used a **decision table**. A decision table is best used for these two functions because it lists outs the situations on which the test cases are either valid or invalid. I created three conditions, one checking if the input is a string. Another one is checking if the string is found in the nounList, in the same iteration, and another condition is that there are no more strings in the list. Below is the decision table for these two functions:

For **white-box testing**, again, we were taught to follow a three step procedure during lectures by creating a **CFG,** p-use and c-use tables, and then input test cases. Below has nodes that corresponds to the line number inside the code for ease of readability.



The next step is creating a table where predicate uses and computable uses are made. Again, this is used for checking if a test case goes in a line (in our case node) or not.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Variable | 97 | 98 | 99 | 100 | 101 | 102 | 103 |
| W | 99 |  |  |  |  |  |  |
| i |  | 99 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

W and i are both strings and are used to check if i is a substring of w. Since i is used declared only in line 98, we check where it is used. In this case, line 99. For w, w is declared in line 97 and is used in line 99.

The third and final step is creating a set of test case or sets of test cases that covers the “all-uses” criterion. Below, we would need to have a word that is a verb, and one that isn’t a verb. This would touch both lines 98 and 99.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | w (inputted word) | i (word in verb list) | Def use cases for w | Def use cases for i |
| T1 | “dogFunction” | “dog” | <97,99> | <98,99> |
| T2 | “run” | “dog” | - | - |