Arthur Liou
CS362
Due: 2/17/19

Midterm 1

You are to test a program called **paraphrase** that takes two file names as input:

> ./paraphrase **in.txt out.txt**

The program does the following:

1. Reads a string in the file **in.txt**
2. Searches the **out.txt** file for the matching string
3. Displays all the lines that contain that string.

Notes:

- The string is any sequence of characters. See **Example-1** below.
- The search is case sensitive. See **Example-2** below.

Example-1: input.txt contains **CS362** and out.txt contains **Welcome To CS362 Software Engineering II**

> ./paraphrase in.txt out.txt
>Welcome To CS362 Software Engineering II

Note: in this example, the line in the file out.txt containing the string CS362 was displayed as output.

Example-2: input.txt contains **cs362** and out.txt contains **Welcome To CS362**

> ./paraphrase in.txt out.txt
>

Note: in this example, no output was displayed because paraphrase did not match cs362 with the line containing the string CS362.

Deliverables:

1. Describe how you would test **paraphrase**. Try to cover as many as different scenarios, including valid (good/positive), invalid (bad/negative), and boundary cases. (**40 points**)
2. Describe how you would check the correctness of the output. (**10 points**)
3. Describe how you would check if your tests are thorough. (**10 points**)

Please see next page for my responses.

Deliverable Responses:

1) Describe how you would test **paraphrase**. Try to cover as many as different scenarios, including valid (good/positive), invalid (bad/negative), and boundary cases. (**40 points**)

I would test **paraphrase** by using a combination of valid, invalid, and boundary cases. I plan to use primarily branch coverage.

For my personal reference when composing my code, I'll pseudocode a barebones function of paraphrase program here:

```
def paraphrase(inFile, outFile) {
    phrase = inFile.phrase; //Code not example, but this is to extract the phrase we're looking for and place it in a string dataType
    for each line in out.txt {
        splitString = line.split(" "); //splits up the line into an array of words, which are determined as those that are separated by a space)
        for each element in splitString {
            if (splitString[element] == phrase) { //if phrase found
                printf(line): //print line to output / console
                break; //Stop looking at this line and go to the next
            }
        }
    }
    return 0; //end of function
}
```

Disclaimer: This function doesn't account for phrases in the in.txt file.

To account for this, I would require a separate function within this program that is similar. This separate function would split the in.txt's phrase into an array and grab the first element of the string to compare again any line within the outFile. If a comparison is found, then the function would proceed to compare the 2nd element of the inString to the next element of that line in the line being compared. In the example below, the program would ID "Frosty" as the first element of that string, then compare input[2] /"the" to output[4] / "the" until the entire phrase is found to be paraphrased or not (see output 2)

Input: "Frosty the Snowman"
Output: "Children love Frosty the Snowman"
Output2: "Children love Frosty the Corgi"

Now for getting that disclaimer out for the way, for the main test cases, I'm going to grab the two examples given to us as a positive and negative test case. For positive, negative, and edge cases, I will provide a given scenario and the expected behavior / output of the test case.

Positive Test Cases (Expected to Pass)

- Expected Test Case1: input.txt contains **CS362** and out.txt contains **Welcome To CS362 Software Engineering II.**
    - Outputs the out.txt phrase to the console.

- Positive Test Case 2: input.txt contains **CS362** and out.txt contains **Welcome To CS362 Software Engineering II. \n I love that Wendy Roberts teaches CS362 \n I love generating test cases in cs362.**
    - Outputs the first two lines.
- Greater Test Case 3: input.txt contains **"Frosty the Snowman"** and out.txt contains **"Children love Frosty the Snowman"**.
    - Outputs the out.txt phrase to the console.

Negative Test Cases (Expected to Fail)

- Expected Test Case 2: input.txt contains **cs362** and out.txt contains **Welcome To CS362.**
    - Does not output anything to console
- Negative Test Case 2: input.txt contains **QA** and out.txt contains **Welcome To CS362 Software Engineering II - Testing. \n I Love QA Testing \n I want to become a QA Engineer.**
    - Output: Line 2 and 3.
- Greater Test Case 3: input.txt contains **"Frosty the Snowman"** and out.txt contains **"Children love Frosty the Corgi".**
    - Does not output anything to console

Edge / Boundary Cases

- Empty in.txt is a " " / space: Program continues on as normal.
    - To ensure this edge case is accounted for, we would need to "modify" the function for the edge case where if there was a space in a line, determined by if the splitString from the function above has more than one element in the resulting array, then we'd automatically return the whole line.
- Empty in.txt or Empty out.txt: // Technically two test cases.
    - No output to console, as nothing to use to compare in in.txt or nothing to compare against in out.txt
- Possible (but unlikely since I'd assume that anything in the in.txt is a string by default) edge case is one where the phrase in in.txt is not a string, but an incorrect data type, like a number. To resolve this edge case, in the program, we could:
    - Exit out of the program early, citing "Error: Incorrect Data Type in in.txt"
    - Covert "anything" in in.txt to a string. Example: number 7 now becomes "7"

2) Describe how you would check the correctness of the output.

To check the correctness of my output (of my test cases), I would review the input / output in relation to the logic of the expected behavior of the program and it's requirements (and NOT it's current logic). In an ideal scenario, the generating of these test cases would be done with a QA team or at least a paired programmed to check that my intended inputs and outputs of my test cases are correct. From there, I can use the generated and built out test cases to test the correctness of the output of the paraphrase program. I also like the idea (from HW3) of "Expected Behavior: x Actual Behavior: y". Utilizing assert() would be very helpful (at least for C) but as the professor mentioned in a previous Canvas page, it may be beneficial to

use output statements instead of assert() because assert() will halt the test suite and not allow gcov to grab comprehensive statistics of the coverages for my test cases.

3) Describe how you would check if your tests are thorough.

I would attempt to identify what kinds of coverages that my test fits and ensure that I have close to 100% coverage. For example, there's four kinds of coverage: graph, logic, input space parititioning, syntax-based coverage. And within graph coverage, there's statement, branch, path, data flow, model-based testing. I focused more on branch coverage, but to use just one type of coverage, even if it ensures 100% is not ideal. Thus, fitting my above test cases into other types of graph coverage, such as statement, path, and data-flow made more sense.

Generally, I would take a look to see how my test cases fit within black and white box testing. Preferably, I would use a combination of both. I'd ID my positive and negative test cases as black box and my edge cases as more white box as how I view those test cases are as follows: in running through program, it will pass if and only if the program has edge case catching for those specific edge case. Otherwise, no code to catch edge cases = not passing.