CIS*3190 A4 Design Report

Re-Engineering a Flesch Index Program written in PL/I to Ada

Michael Tran

0524704

Synopsis

For this assignment we got to choose between three topics to accomplish as our final project. I elected to choose the second topic which dealt with re-engineering a Flesch Index program from Programming Language One (PL/I) to Ada. The Flesch index is a formula that estimates the "readability" of written materials based on the average number of words per sentence and the average number of syllables per word. The readability index gives an idea of how easy or hard the material is to understand. A Flesch index score in the range of 0-30 meant that the readability would be very difficult such that only individuals with college/university education would be able to read. Scores of 50-60 are high school level and 90-100 should be readable by fourth graders.

We had 3 main tasks to consider while re-engineering this program:

1. Completely re-engineer the program from PL/I to Ada to compute the Flesch Index
2. Allow the program to also calculate the Flesch-Kincaid grade level
3. Allow the program to read in a file

Luckily for this assignment we were given the algorithm to be able to calculate both the Flesch and Flesch-Kincaid scores. However, before attempting to re-engineer the program, I first had to learn the basics of PL/I which would allow me to convert the program to modern Ada. After learning much of the basics of PL/I, I spent time looking at the algorithm before looking at the legacy PL/I program. The algorithm was very simple to understand because there were only 3 main keys to follow to calculate both scores. The 3 main keys were:

1. Counting the number of sentences.
    a. According to the algorithm what constitutes as a sentence is where a period, exclamation point, question mark, colon or semi-colon are found. This would count as an end of sentence mark.
    b. However, I believe that only where a period, exclamation point or question mark found should only qualify to count as a sentence, although I followed what was mentioned above to count as a sentence.
2. Counting the number of words.
    a. Each group of continuous non-blank characters counts as a word.
3. Counting the number of syllables.
    a. Each vowel in a word counts as one syllable subject to the following sub rules:
        i. Ignore final –ES, –ED, –E (except for –LE).
        ii. Words of three letters or less count as one syllable.
        iii. Consecutive vowels count as one syllable.

After studying the algorithm, I examined the legacy code to see how this was implemented. While examining the legacy code line by line I noticed that it was broken up into functions that would count the number of sentences, words and syllables, thus making it modular and the easiest legacy code that we have been given to read. After reading through the code, I thought it was easy enough to understand and decided to write the program from scratch while consulting the legacy code if I were ever to get stuck.

Detailed Approach to Re-Engineering the Legacy Program

The first step I took in the re-engineering process was to first be able to read in a file. Since it has been a while since I last coded in Ada, I had to consult my hangman code that I wrote for the second assignment where it had a function dealing with reading in files. After reviewing the program, I had to decide how I wanted to read the file. The two options I was thinking were reading the file one character at a time or reading one line of text at a time. After thinking about both options, I decided to go with the second option because when reading one line of text at a time, you could easily distinguish between the final word on that line from the beginning of a new word on a new line. The problem with the first option by reading one character at a time was I could not determine when a word would end at the end of one line and a new word would begin on a new line. After deciding between the two options I had to decide what type of string I wanted to use. Since Ada has a variety of different type of strings such as unbounded and fixed length strings to name a few, I decided to go with unbounded strings because I became more comfortable working with this type of string due to the hangman program that we wrote earlier in the semester. I also knew that unbounded strings have many functions that could help me with the program as well. Once this was all decided, I had to test that my program was able to print out a line of text from a test file full of text. After this was confirmed I moved onto parsing the line of text.

For the next step I had to consider how I would parse the line of text once I have stored it in the unbound string. Before parsing the line of text, I had to consider removing blank spaces at the beginning and end of the line of text so I could determine where the word starts and ends. Afterwards, I had to decide how I would parse the line of text. I decided that the easiest way was to go through the line of text character by character until a whitespace was found; this was done by looping through the line of text. Once a whitespace was found, I considered that to be the end of the word. To be able to parse that word from the line of text, I had to keep track of where the word would start and end. How I did this was by having an index variable be equal to the first letter of the word found and then whenever the next character is a whitespace, slice the word at that point and send it to be parsed. This is also how I kept track of how many words were in the text. While this would find every word that had a leading whitespace in front of it, it did not take care of the last word in the line of text. This was due to no leading whitespaces at the end of the line. The way I took care of this situation was while looping through the line of text, when the counter of the loop equals the length of the line of text; I knew I have found the last word in the line of text, therefore allowing me to parse the last word in the line of text. Once this was completed, I tested that the program was able to print every single word in the text on a single line. After it was confirmed that I was able to print every single word on individual lines, I moved onto counting the number of sentences found in the text.

To be able to count the number of sentences found in the text, I made this into its separate function. The function was very simple to implement because according the algorithm Dr. Rudolph Flesch developed I would only need to look for a period, exclamation point, question mark, colon or semi-colon at the end of a word for it to count as a sentence. When I first attempted to implement this function I decided to use a loop and switch case to find the sentence terminators. I believed this would be the most efficient and code readability friendly way of an approach, however, I ran into many problems with this method. Ada being very picky with strings would not let me loop through the string one character at a time and compare that character as a case. This was because I was using a string, which is different from an array of characters, trying to access the word as an array of characters versus a string, which Dr Wirth pointed out. I tried many other ways to access that character from the string by having a

character variable set to that element in the string but that still did not work. As a result, I had to think of a different way to approach this function. Since a switch case is the same as using if-else statements, I had to resort to using this method. This proved to be a much easier way to implement this function because Ada allowed me to compare the string character to a character. However, after implementing this method, it came to mind that going through the string one character at a time would be very inefficient because you really only need to look at the last character of that string. So I made the change from looping through the word to look for the terminator to checking the very last character in the string. Once I finish completing this function, I began to test that this function was detecting the terminators at the end of words to be able to count the number of sentences found in the text. After I confirmed by manually counting the number of sentences found in a text and comparing to what the program found, I was able to move onto counting the number of syllables found in the text.

For the next step, I made another function that would count the number of syllables found in the text. This function was a bit more difficult than the previous function because I had take into consideration the 3 sub rules that were mentioned in the Flesch Index algorithm. Before implementing this function, I had to determine how I would want to approach this function. I had to decide whether I wanted to write a big function that would take care of all 3 rules or did I want to break this function up into smaller programs. That's when I consulted the legacy code to help me decide how to tackle this problem. After noticing that the legacy broke up this function to two smaller programs, I decided to adopt the same approach. The function was going to take into account the last two rules listed:

1. Words of three letters or less count as one syllable.
2. Consecutive vowels count as one syllable.

To accomplish these two tasks I decided to make use of the if-else decision making method. According to the first rule; words of three letters or less count as one syllable, I decided to make one of the parameters of the function an unbounded string. This was due to the built in function called Length() that returns a natural which would give me the length of the word. With this I was easily able to distinguish words that were length of 3 or less which would count as one syllable. Otherwise, the word is greater than 3 and I would have to count the number of syllables found in the word. To accomplish this, I decided to loop through the word character by character and determine whether it was equal to a vowel. However, I must take into considering that consecutive vowels only count as one syllable. To achieve this, I had to use a boolean variable to keep track of consecutive vowels. Once a vowel was found, I would set the boolean to true and increase the vowel count. This would keep the vowel counter from increasing when encountering repeated vowels. The only time this flag resets and resumes counting vowels is when a character is not a vowel. Once I was able to achieve this, I ran tests to determine that this was working properly and it was counting the number of syllables correctly with these two rules enforced. Seeing that the function was executing properly, I moved onto the second part of counting syllables.

The second part of counting syllables was dealing with the first rule which involved ignoring the final –ES, –ED, –E (except for –LE) found in a word. Since the first function dealt with counting the number of syllables while ignoring the first rule, this function was to count the number of words that ended with –ES, –ED, –E (except for –LE) than subtracting the result found here from the number of syllables found in the first function to give you the final result of how many syllables were found altogether in the text. This approach was a bit different from what the legacy code was doing. In the legacy code, the function would determine whether the word contained the suffixes explained above and essentially remove it from the word before determining how many syllables were found. I decided on this approach because I believed that

this method was easier to implement in Ada due to the fact that strings are a bit difficult to manipulate compared to other languages.

The way I implemented this function was by first determining whether the word was longer than 3 characters otherwise the rule would not apply to the word. If the word qualified, I used nested if statements to determine whether the word contained the suffix that would result in subtracting from the total found in the first function. The first case I checked was if the word ended with the letter "E". To accomplish this, I would only check the final element in the string. Once it was determined that the letter was found, I than would check if the second last letter was an "L". If it was determined that the character was not equal to the letter "L", that meant the word contained the suffix to which was to be ignored. The same logic was applied to the second case where if the word ended with "ES or ED". The only difference was instead of looking at the last element, I would check the second last character was equal to the letter "E". Once it was determined that the suffix to be ignored was found in the word, it would increase a counter. This counter was used to find the final result of how many syllables were found in a text. However, I ran into a small problem with words containing a punctuation mark at the end of the word. Since I am only looking at the last two characters in the word, if a punctuation mark was found the results would not be correct. To remedy this situation, in the function that would count the number of sentences, I had a flag telling me whether a punctuation mark was found to allow me to remove it from the word, thus allowing the functions counting the number of syllables to work correctly. After implementing the two functions, I began testing whether it was calculating the number of syllables correctly with a custom text file. Once this was determined to be correct, I moved onto printing out the summary of what the program found.

The summary was to display to the user the number of words, sentences and syllables found in the text. It also included displaying what the Flesch index score and Flesch-Kincaid grade level were for the text being examined. To keep the program modular, this became a function as well. The only difficult part of this function was calculating both scores. I had just a bit of trouble type casting integers to floats to have the calculations be correct. After tinkering with correctly typecasting the values, the summary function was completed. Once this was complete, the program was nearly finished. The last objective was to make my program allow the user to keep reading files until they wanted to quit the program. This was easily implemented and moved the development of my program into the final stage of testing.

We were given finished test results for some sample texts to compare our results. I used these results as a reference to determine whether my program was executing properly. After running through all the tests, it was determined that the values my program calculated closely matched the reference values, thus concluding my final assignment of my university career.

Concluding Remarks

Overall this was the best choice between the three topics that were given because I only had to write one program compared to 5-8 programs that the other topics wanted you to do.