CIS*3190 A3 Design Report

Re-Engineering a Roman Numeral Converter from COBOL74 to Modern COBOL

Michael Tran

0524704

Synopsis

For this assignment we were tasked with re-engineering a Roman numeral converter from Cobol74 to modern Cobol. The program converts a number expressed in Roman numeral to its decimal equivalent. The program uses the Roman numerals I, V, X, L, C, D, and M - all others are considered invalid. The decimal equivalents of the above numerals are 1, 5, 10, 50, 100, 500, and 1000. We had 4 main tasks to consider while re-engineering this program.

1. Migrate the legacy Cobol program while removing all legacy features and attempt to modularize the program more
2. Allow the user to input a Roman numeral on one line instead of individual Roman numeral characters one at a time
3. Allow the program to read in a file containing a series of Roman numerals to be converted
4. Allow the program to use either lowercase or uppercase Roman numerals

Before attempting to re-engineer the program, I first had to learn the basics of Cobol which would allow me to convert Cobol74 to modern Cobol. After learning much of the basics of Cobol, I spent time looking at the legacy Cobol program given to us to gain a better understanding of what is happening in the program. Whenever I would get lost in the code I would consult the flow chart that was given and make sense of what the legacy code was trying to achieve. While examining the legacy code line by line I would locate where I would have to modify or remove legacy features found in the program. At the same time I grouped parts of the code where it would make sense to have a subroutine to execute the code to modularize the program. After reading through the code, I started to map out how I wanted to approach the legacy code.

The first step included getting the full Roman numeral string from the console, converting the string to uppercase and getting the string length. I started with allowing the user to input a string of Roman numeral characters and see if I would be able to capture it and display it back to the screen. After a bit of research on how to do this I was able achieve this task. Afterwards I had to research how to convert the whole string all to uppercase. Luckily Cobol has verbs called inspect and convert which allowed me to convert the whole string to all uppercase which would allow an easier time of converting the Roman characters to its decimal form. To get the string length, the verb inspect came in handy again. Combining the verb with tallying allowed me to count all the characters in the string and count the number of white spaces to find the string length. After getting some functionality working, I tested to make the strings inputted from the user has the correct string length and is converted to all uppercase. Once this was confirmed to be executing correctly I went on to the next step.

The next step involved was being able to pass the string to the external function that would convert the Roman numeral to its decimal value. I first started with re-engineering the conv.cob file that was given to modern Cobol. In this file had the majority of the go to statements that I had to remove. The biggest problem I ran into while doing this was dealing with the IF statements. In my experience with other languages, I would just use an if-else if- else structure to accomplish what I wanted, however, I was never able to get it to work correctly. After doing some research I came across something called Evaluate. This was an alternative to using a set of nested IF statements to test several conditions. It was comparable to a switch case in c programming that I was easily able to understand and use. This helped me solve my problem and allowed me to continue re-engineering the file. After finishing re-writing the converting function to modern Cobol, I tested whether the string being passed into the function was properly being converted. I used an assortment of Roman numerals and confirmed whether the function was properly converting Roman numeral to its decimal equivalent. On top of that I made sure that invalid Roman numerals would make the function stop converting the Roman numeral and give a prompt to the user that there was an illegal character. Once this was executing perfectly, I made the loop converting the Roman numeral into a subroutine and tested it a final time to make sure the transition was done correctly.

The next step included getting the program to infinitely loop until the user types "Q" to quit the program and formatting what the output should look like. This process was not too difficult to handle as I just took the formatted table from the legacy code and modernized it and had a big loop to carry out all the functionality. Before moving onto file processing, I tested to make sure that the output was correctly formatted, the prompts for the user was helpful and intuitive and that I could keep converting Roman numerals until I wanted to exit the program. Once this was all confirmed to be working I took the next step of file processing.

File processing was one of the 4 main tasks that we had to implement into the program. Before attempting to code this, I had to read a few examples of how to read files. Fortunately, Cobol makes this very easy to execute. I was able to read in a file, and display the contents of that file back. Once I was able to do this, I had to make a minor adjustment that would allow the user to process whatever file they wanted. Afterwards, it was just carrying out the same process as converting user input from the console to convert the Roman numerals to the decimal equivalent. Once this was confirmed to be working flawlessly I moved onto the last step of allowing the user to keep processing a file one after another or manually convert Roman numerals and try to modularize the program.

For the last step, I decided to keep asking the user if they wanted to read in a new file after it was finished processing the file before. If they decided no, it would go into a manual mode where the user can either quit the program or continue to manually input Roman numerals to be converted. After getting all of this to execute seamlessly I broke up the program into small subroutines. I found this way to modularize the program to be more efficient than having a few external functions which meant separate files. In all I had 4 subroutines which dealt with reading in a file and converting it, converting user input, printing the results and a subroutine to carry out all of the above. Afterwards I would run a few test cases to make sure everything was working properly and that I was happy with how the program was running.

Identify Legacy Features

Legacy Code	Re-Engineered Code	Comments
IDENTIFICATION DIVISION. PROGRAM-ID. ROMANNUMERALS	identification division. program-id. romanNumerals.	• In both files of the legacy code, it was written all in uppercase. • In the re-engineered code it was all written in lowercase
FD STANDARD-INPUT. 12 01 STDIN-RECORD PICTURE X(80). 13 FD STANDARD-OUTPUT. 14 01 STDOUT-RECORD PICTURE X(80).	accept "…" display "…"	• In the data division of the legacy code, standard input/output were used • In the re-engineered code I was the verbs accept and display
77 N PICTURE S99 USAGE IS COMPUTATIONAL. 77 TEMP PICTURE S9(8) USAGE IS COMPUTATIONAL. 77 RET PICTURE S9 USAGE IS COMPUTATIONAL-3.	01 inputLen pic 99 value zero 01 romanValue pic 9(8) 01 err pic 9	• Some of the variable names in the legacy code were not very informative. • I renamed them with better identifiers
IF D IS GREATER THAN PREV COMPUTE SUM1 = SUM1 - 2 * PREV.	if romanVal is greater than prev compute total = total - 2 * prev end-if.	• Converted the legacy if statements to modern Cobol to end the IF statement with end-if.
IF S(I) IS NOT EQUAL TO 'I' GO TO B1. MOVE 1 TO D. GO TO 3.	evaluate romanCh(i) when 'I' move 1 to romanVal	• Nested if statements were used with no end-ifs. • Re-engineered using an evaluate statement
IF S(I) IS NOT EQUAL TO 'I' GO TO B1. MOVE 1 TO D. GO TO 3.	evaluate romanCh(i) when 'I' move 1 to romanVal	• Go to statements were removed using the evaluate statement

Questions and Answers

1. For a program of this size, I believe you could have easily re-written this program from scratch in a language such as C. There isn't much logic to the program other than checking if the current character is a Roman numeral and what value to give to it. Also

instead of having two files you could have a single file for this program. However, Cobol deals with files much better than other languages I've experienced.

2. There weren't too many problems that I faced while re-engineering the program. At first Cobol seemed like a difficult language to understand because it doesn't really look like a programming language. Also the way variables are declared I had to get used to. There is no int data type to say it's an integer type variable, rather you have to choose the correct picture characters to describe the variable. If you were to use the wrong characters the output/input may be not what you were expecting. So that was a minor problem I ran into. Another small annoying problem was when to use the period. If it was within a perform, or if statement no period was needed until the end of the block of code. I had a hard time remembering this which always leads to compiler problem. Other than those problems listed, the language was not too difficult to use.

3. My program was much longer than the original legacy code. Compared to the legacy code, my program is much more structured and easier to read. The legacy code relied on multiple go to statements which allowed the code to be very short. I removed all the go to statements completely by using correct loops and decision making statements which adds length to the code but makes it much easier to read and follow what is happening in the program. Also, I took advantage of using subroutines to carry out small tasks which allowed to the program to be modularized, which allows better readability.

4. In my personal opinion, I believe this was a good approach of re-engineering the legacy code. When you have code to look at you can gain a better knowledge of what the program might be trying to accomplish, rather than starting from scratch. However, it depends on the size of the program. For this assignment, you could have easily started from scratch and just used a flow chart to code this program. On the other hand, trying to re-engineer legacy code that is 10,000+ lines would be a difficult task to accomplish. Having the legacy code and a flow chart allows for better understanding and makes it simpler to tackle the problem of re-engineering the legacy code.

Being faced with learning a new language can sometimes be difficult. Having to learn the syntax of the language, how to declare variables, how to create functions/procedures and I/O can be difficult to grasp. Cobol at first looked very daunting to learn. The language was none I have ever seen before because how it looked like just pseudo code and not real code. However, slowly learning the basics I was able to slowly get into the groove of how to program in Cobol. The only problems were giving the correct picture characters to the variables to make it how you want the variable to look like and when to place a period to end a sentence. However, there were a lot of cool features that it has such as inspect, which made doing certain things much easier. Also I liked how easy it was to read files in Cobol. Overall the experience was very educational and something I could see myself doing in the future with re-engineering Cobol programs.