# CIS*3190 - Assignment #3 Reflection

To start the assignment, I first read a lot about Cobol because I heard that it was completely different than any language I have ever used before, which was very true. It took a long time to grasp how to program in Cobol, things like all of the declarations at the top, where you declare variables, and how to even use them. Declaring the variables was the weirdest part for me, because there are no keywords like string, integer, float, etc. It was hard to realize how you declare an integer, and how you declare a float.

After learning a bit about Cobol, I took at the previous code for the Babylonian Square Root calculator to try and piece together exactly what I needed to make. I felt as if the assignment itself wasn't going to be hard, but understanding what I needed to re-engineer was the harder portion of the task. I took time to type in each line/block of code into a previous Hello World program I compiled, to see what is happening step by step. This helped a lot because I could add a bit of code, run it and see what is happening, and comment it for when I write my version of the calculator. After spending a few hours doing this, I better understood Cobol, and exactly what I was to make.

The first step I took to re-engineering the calculator was to take apart the previous calculator and build the components piece by piece. This approach modularized the calculator, as well as making the coding very easy to do.

The first component I created was a function that allowed user input, since we were to design a better way of getting user input. My solution was to use an infinite loop, where the user can type in a number, hit enter, and see the square root result displayed to them as many times as they want, until they press control-c to exit the program. It displays all of the prompts to the user, and once they input the number it is stores it into a variable that can be later retrieved by the square root function I made after. I ran into some problems here because of the variables, and how you input integers into a Cobol program. I declared my integers as 4 digits only, so inputting the number 91847 was not happening correctly. Eventually I re read the notes, and found my answer and fixed this.

The second component I created was the most important, which was the function that calculated the actual square root itself. To make it easier, instead of making the function external at first (in another file), I made the function internal and just called it from within the same file. I just wanted to make sure I was producing the correct result before I migrated the function into it's own file, as the assignment stated. To do the Babylonian square roots algorithm, I needed to make an initial approximation. I googled around to see how people calculated this, and found that the easiest and simplest way was to just divide the original number by 2 and start there, so that is what I did. The original approximation is the inputted number divided by 2, and after I calculate this I enter a loop that goes 33 times (it was randomly selected, but I found that the answers are precise enough, so that is what I used). In this loop is where I divide the original number by the approximation, add the result of the division to the approximation, and average the numbers by dividing by 2. This loop over and over, using the

new approximation each time, and after looping the 33 times, an answer for the square root is produced. After testing different random of different sizes numbers, and using a calculator to check the answer, I had this function working correctly.

The next component I needed to make was a function that prints the final result to the user. I used the output format from the assignment page since I felt this was a great way to display the result. This was a very easy function to make, it is pretty self explanatory (display the result, in a pleasing way).

The next step was to make the square root function external. I created a new file (main is sqrt.cob, sqroot.cob was the new file that would contain the square root function) to hold the square root function. After a lot of googling, I found a good template to call a program from within another. I used this template to create my second file, since it had the new linkage division, for what I think is only for linking the 2 programs together in terms of parameters, so that they may pass information between each other. After this and some trial and error with compiling them (used a shared library originally, later found out you can just compile it like two C programs where you specify each source file in one line), I had the function getting called from the first program.

After more testing, and returning to the assignment page, I noticed I am not accounting for negative number input, which should not produce a result. I went to my program and inputted a negative number, and noticed it just treated the number as a positive. After some googling, I realized that I did not make the original input a signed number. So I did this, added a check for the user inputting a negative number, and added a 'INVALID INPUT' message as the result. If the number was correct, I use the MOVE function to put the data into an unsigned variable, so that when the number is printed, it is printed without a + sign.

After even more testing, I realized that it is only calculating the first number correctly, and the rest are always incorrect. I thought I was not zeroing out variables correctly, so I went and zero'd out all of the variables in the square root function file. This did not solve the problem, so I had no idea what could be wrong. I had assumed the counter in the function would always be 0 when the function ran, because that is how it works in every other language when you call an external function, so after setting the counter to be 0, the loop was now entering and exiting correctly every time.

After some final testing, I was able to conclude it is now working correctly, commented all of the logic in the 2 files, added function headers, as well as file/program descriptions to both files.

**Given your knowledge of programming, was Cobol easy to learn?**

Cobol was by far the hardest and weirdest language I had to learn. The first part I had a hard time with was why you need to add all of the division declarations at the top, as in almost every other language I used you never had to do anything like this. The next is what went into these sections, since to specify variables you needed to do it in the data division section, not at the top of a function like normal.

The next weird part was the way you declare the variables. There are no keywords like integers, floats, strings, etc, but instead you used this 'picture' concept which I found very odd,

and you also have to label them with a number which I still don't quite understand what the point of that is. To make variables different, you just change the format of this 'picture', for example integer is '01 a PICTURE 99', which is a 2 digit integer. To make a float, it is '01 b PICTURE 9(11)V9(6)', so it is hard to differentiate the 2 when you are just learning the language.

The concept of the '.' ending the program/function was a huge beginners trap as well. In a normal block of code, you can put the period at the end of the line without any problems, which would leave you to believe this is how you end a line. But when you put it at the end of statement in an if block or a loop, you get a bunch of errors about the if block or loop not being ended correctly. It was very hard to pick up on the fact that this period was prematurely ending the block, and you can't have it there.

In summary, I feel that Cobol had a lot of beginner traps, where someone who never learned the language before gets caught in these odd situations where they can't figure out what is wrong. It is also a completely different language in terms of the syntax and how to use variables, so it made it very difficult to learn in my opinion.

**What structures made Cobol challenging to program in?**

There are many things that made Cobol difficult to program in. The first structure was how functions cannot return values. They can just manipulate global variables, and that is how information is passed between them. This was challenging because the entire time at school, you are taught to avoid global variables, and here you are stuck using them without a choice.

The second structure that made it challenging for me was the way you read the code, as you can't format it to look similar to a C program, since you have all of these divisions and weirdly setup functions. It is hard to read over and debug the code you already wrote when you are constantly trying to figure out what is doing what, where as in C you can easily tell which parts of the code you want to look at, and what is irrelevant. This was the worst part for me.

A third structure that made Cobol challenging was the periods that I mentioned in the previous question. It is tricky to keep track of where you need a period and where you don't, and often times you find yourself reading a big block of code to figure out why it isn't compiling, only to figure out you added or are missing a period on a specific line.

A fourth structure that made this language challenging to use was how you assign variables. Instead of using '=' you have to move the values into a variables, similar to assembly language programming. The reason this was challenging because it is not how you do it in any other language, so doing this was odd to get used to. Opposite to this, the way you call functions on variables (like the built in ROUNDED function or SQRT function) you do use the '=' sign to assign the result to a variable, which made things very challenging to get used to.

Finally, the last structure that made it difficult for me to program in Cobol was the fact that everything does not behave how you expect it to. For instance, I assumed when you declare an integer, it is signed by default and if you store a negative number it will just work, like in any other language. This is not true, as the default is unsigned in Cobol, and declaring a signed variable is not very intuitive. As well as when using the signed integer, when you print it you would assume if it was positive no sign would be displayed, and if it was negative a negative sign would be displayed. This is not the case, and when printing a positive signed

integer, you get a ‘+’ sign printed as well, and you it is annoying to work around this. Also a simple thing like removing leading zeros from an integer is very easy to remove in any other language I have used, except for this one. I googled around for how to do it, and some of the solutions were longer than my entire program, so I did not include this. Things you assume are easy are not at all easy to do, and things you know are hard to do are infinitely as hard to do, which is what made Cobol difficult to program in.