Brandon Chester

Cobol Text Analyzer

Program design process and strategy

For the Fortran re-engineering assignment, I ended up essentially rewriting all the logic of the program. While I did do the re-engineering in chunks, none of the code from the original source remained in the end. With this Cobol program, my process was much closer to what one would describe as re-engineering.

My first order of business was to examine what didn't function correctly in the program. This was done before I even took a look at the code, as I wanted to know from a high level what was definitely broken without having the code influence my opinion in any manner. The assignment itself helped in making this clear, as it mentioned how the program would make mistakes with the word counter when multiple spaces are clustered, could not recognize sentences ending in anything other than a period, and could only read lines of up to eighty characters. Even without looking at the code, these issues suggested that there were serious flaws in the logic behind the program.

After gathering some known issues from the assignment, I also discovered that the program did not work with the provided example files. At this point I had to go into the code to figure out the problem, and I ended up discovering that the program needed to come across a backslash character before it would terminate execution. This was obviously problematic, and it explained why the example text from the assignment had a strange trailing backspace.

Based on these known issues, and my examination of the code, I got to work on re-factoring the code so it would be easier to understand and reference. Much of this had to do with implementing proper end statements for various blocks like if-statements, perform blocks, read blocks, etc. I also converted the program to lowercase. This led to better code readability, which helped with my understanding of what exactly the code was doing, helping me to identify the logical issues in its methodology for analyzing sentences.

```
perform until endOfFileFlag = 1
    read inputFile into inputArea
        at end
            perform EndOfProgram
        not at end
            display inputArea
    end-read
end-perform.
```

After this analysis, I determined that, at the bare minimum, the logic for analyzing the lines in the text files would have to be rewritten from scratch. It made too many assumptions about the structure of the text file, and wasn't really salvageable. However, before I could get to work on that, I would need to refactor the higher level parts of the program. For example, I modified the code to read the input file into the code you see above, which helped me to ensure that the data was being read in correctly before moving on to the text analysis portions.

After confirming that the file was being read in correctly, I started working on refactoring the text analysis portion of the program. Like the original program, after reading in a new line, I perform another section of code eighty times, and traverse along the line to count the various metrics. The original program separated this into two sections, with the first being called `NEW-SENTENCE-PROC`, and the second being called `PROCESS-LOOP`.

`NEW-SENTENCE-PROC` worked only on the basis of the file having the trailing backspace, so I decided to rewrite this part of the program entirely. I brought the logic into a single paragraph called AnalyzeLine, which increments the counter until it hits eighty, and checks each character in the data read in from the file.

AnalyzeLine makes use of two flag variables called isWordChunk and isNumberChunk. Upon encountering an alphabetic character, it sets isWordChunk to one, and increments the number of words found in the file. That flag is only set back to zero upon encountering a space, indicating the end of the word. The code for tracking the number of numbers works on the same principle. In addition to those checks, the program also increments the number of sentences whenever it encounters a period, an exclamation mark, or a question mark.

The last part of the program to re-engineer was the output section. This code is essentially left as it was, apart from some cleanup to bring the code up to more modern Cobol standards, such as replacing verbose division statements with compute statements. An output for the number of numbers in the program has been added as well. I would be lying if I said I understood exactly how it prints out the records on a single line, so this is a case of "if it ain't broke, don't fix it", which seems very applicable to legacy software modernization in general.

Ease of learning Cobol

I would consider myself a decently experienced programmer at this point, and I've worked in many different languages with many different paradigms. I didn't really find Cobol difficult to learn, although I'm not sure if I would really say that I've learned Cobol simply by doing this assignment. I suppose it would be more accurate to say that I was able to learn enough about Cobol to apply that knowledge in the context of this assignment. I was able to re-engineer a Cobol program to properly perform the task of analyzing text files, but there are still many parts of the Cobol language that I don't really understand.

For example, I don't really understand how the record structures work, and I don't understand the relevance of level-numbers, with the distinction between level 01 and 77 still being entirely unknown to me. I also don't really understand the block syntax, which requires a period only for the very last line. My assumption is that this is because the language treats the block as one enormous statement, with the block formatting being purely a visual aid, but I cannot confirm this with my limited knowledge of the language.

As a language, Cobol is quite verbose, and it's pretty clear that it was designed with non-programmers in mind. Once you adjust to the syntax and the verbosity, there's not much difficulty in writing simple procedural code in Cobol if you have knowledge of writing the same sort of programs in other languages like C. I certainly wouldn't want to write a large program in Cobol, as it doesn't appear to have the features of organizing data into structures and manipulating them that are generally necessary in large software projects. Ultimately, the basics of Cobol are not difficult to learn, but I would not call myself an experienced Cobol programmer, and I wouldn't want to try to write a large program in it.

What made Cobol challenging

Due to the nature of this assignment, I didn't have many large difficulties with programming to meet the requirements. I will say that the lack of typical loop structures in Cobol leads to an unnecessarily complex program structure, making it difficult to write code and understand the logic and control flow after the fact. I think Cobol's mechanisms for reading and writing to the display and to files are somewhat strange as well. For example, I left the file output code essentially as it was, as I don't really understand how it works and it seems to work well enough for this purpose so I didn't want to introduce bugs or break its behaviour by applying large changes without an adequate understanding of the existing functionality.

What I liked about Cobol

Cobol does have a few features that I enjoy. For example, I think using the operators "or" and "and" instead of the C-style II and && are much better. They don't take up much additional space, and are more readable and comprehensible even to non-programmers. The fact that many modern languages still use the C-style logical operators is something of a disappointment.

What I dislike about Cobol

The thing I disliked most about Cobol was how verbose it is. I don't actually dislike the structure of operations like "add 1 to x"; i's not really much longer than writing "x = x + 1". However, when working with more complex operations, the additional keywords that are required compared to an equivalent operation in a typical language make it very tiresome. For example, a command like:

```
DIVIDE NO-OF-SENTENCES INTO NO-OF-WORDS GIVING AVER-WORDS-SE
ROUNDED.
```

is so long, I can't fit it on a single line in a twelve point font. Obviously newer versions of Cobol provide the compute statement, but that ties into another issue I have with Cobol, which is that there's never a clear best way to perform an operation. The language really drags around a lot of technical debt from old versions, with many legacy functions that have been replaced by newer ones, but nothing to indicate what is now considered the "right way" to write a Cobol program.