Assignment 3: Cobol Re-engineering

by Ryan Wilson-Perkin

“The use of Cobol cripples the mind.” - Dijkstra

Well, we were warned. I certainly did not expect this much difficulty from Cobol. I've taken to wondering if this course was ordered as such in order to fuel our descent into madness, or simply to make us grateful for the tools we have today. Either way, the goal has been accomplished. There's nothing inherently *bad* about Cobol, it's just so different from anything that I've encountered before that it feels *wrong*. As if the code I am writing couldn't possibly be translated into anything a machine could understand.

Cobol is very data-oriented. All of the file descriptors for relevant data files are declared at the beginning of the program, prior to any of the procedures that would work on them. The structure of the file is defined explicitly and bytes are addressed individually and translated into data structures. I'm sure this makes it very easy to read and write records to simple data files, rather than relying on complex data bases. Tangentially, it also means that I couldn't figure out how one would prompt for a filename to read from, since that would happen after the file declaration and so my program lacks that functionality.

Cobol is also very process-oriented. This is evidenced by it's structuring of program into multiple divisions and sections, each with their own particular syntax and declarations. I actually kind of like this. Not enough to use it as my language of choice (far from that), but it's interesting to see from a language-design point of view that someone decided that everything should have it's place. The love of process is also visible in the way that Cobol deals with “sentences” and “paragraphs”. Where other languages would encapsulate functionality inside of functions, Cobol opts instead to use named paragraphs, which seem to be no more than a block of code bookended by go-to statements. At least the language designers had the common decency to hide the go-tos from us.

The algorithm itself was simple enough, and so this provided a nice balance between the work of learning a new language, and the relative ease of translating the algorithm. The process goes:

- Translate the current roman numeral into decimal equivalent
- Add the decimal to the sum
- If the decimal is larger than the last one, remove the last one twice from the sum
- Repeat until out of roman numerals

Re-engineering

The re-engineering process consisted of five components: turning go-tos into conditionals and loops, breaking functionality out into paragraphs, adding ability to input numeral all on one line, adding ability to recognize lowercase numerals, and documentation.

Removing Go-Tos

I tackled the simplest go-tos to start, and help me get warmed up to Cobol. In the **conv** module there were six labels, and associated jumps, being used for evaluation of the current letter. This structure lent itself well to a switch-case block as I know it from C. Cobol doesn't have switch-case, but it does have an equivalent construct in the form of evaluate-when. This structure *evaluates* the result of an expression and runs the statement *when* an equivalent expression is found. The beauty of the evaluate-when usage is that it also has a built-in *default* option known as “other”. If no expressions matched (ie. we are looking at a letter, for which we have no associated decimal value) the “other” conditional kicks in, and we can set an error flag.

The go-tos found in the romanA3_1 program were tougher because some were being used for simple conditionals, and some for looping conditionals. The simple conditionals had equivalent functions in the if-end-if forms of modern Cobol, which allow the user to clearly define which sections should get run in which cases. The looping conditionals were sometimes replaced with explicit loops, but in some cases I found the Cobol primitive perform-until to be a better match. The functionality to be looped (eg. translating each new string) was clearly independent enough of the other code to justify factoring it out into it's own paragraph. By doing so, I ended up with a “translate” paragraph that I could invoke in a perform-until, with the condition that the loop stop when end of file is reached. End of file was determined at a lower level (reading lines of input) and percolated up to the loop level through the use of a simple flag.

Modularization

One paragraph has already been mentioned, that which was used to “translate” the current roman numeral string into a decimal equivalent. There were other similar functions that lent themselves well to modularization within this program, the first of which is “write-prompt”. The initial program was not particularly intuitive, because it relied on the user hitting enter enough times to trigger the translation of the current buffer. I modified the program to instead print a prompt asking the user for a new roman numeral. This functionality simply writes two lines to stdout, the first is the prompt itself, and the second is a “>” symbol to mirror the one found in many terminals. This symbol is an indicator (at least to a particularly geeky crowd) that the program is waiting for input.

The next feature to factor out was the translation of the input the user typed. This is found in the “get-roman” paragraph and works threefold. At first, it resets the roman numeral string to all spaces, to avoid interference from previously translated values. Then, it reads from

standard input into the roman string, and sets the end-of-file flag if our input stream is closed (can be generated interactively by pressing control and 'D').

There is also a feature for calculating the number of characters entered by the user, found in “compute-roman-len”. This isn't truly necessary, the **conv** module could simply stop processing when it hits a space, but I find it's better to be explicit. The resulting length is stored in the roman-len variable and is used when calling the **conv** module.
Finally, there is a feature for printing the title of the program.

This isn't truly necessary, since it comprises 2 lines that are both simple write statements, but it is good practice for future-proofing to break it into it's own section. This allows for the title to be turned on or off, using an if statement. After all, the UNIX philosophy is that programs should be non-verbose by default, with the option to print more information. It is my belief that the title of the program should be found in the man pages, not the output of the program, but I left it in in case the professor was looking for a program similar to the one initially provided.

Input numerals on a single line

A deceivingly simple feature to implement. The program was already reading input in, 80 characters at a time. The difference is, the original program was only looking at the first character and treating the rest as padding. I simply re-organized the string-like structure being read into, and used every character rather than just looking at the first one.

To be explicit, the original structure was called INPUT-AREA and had an attribute IN-R which was a single character string, followed by an attribute FILLER which was a 79 character string. In my modification, I use a structure called roman with a single attribute s which is a 30 character array. I chose the number 30 because that was the maximum size of the passed array in the original program. Naturally, this limits the program to dealing with roman numeral strings of length 30 at most. Thus the largest value that can be produced is 30 M's, equivalent to the decimal number 30,000.

Recognizing lowercase numerals

This was the easiest feature to implement since I already had a very straightforward way to check individual numerals into decimal equivalents. Where I previously had 7 lines, comparing the current letter to the letters 'I', 'V', 'X', 'L', 'C', 'D', 'M', I now had 14 after adding their lowercase equivalents.

Documentation

Despite it's flaws, Cobol's natural language approach to syntax does lend itself to self-documentation rather nicely. Given how clear the structures became, it felt redundant to add comments to the program at all, but I did so anyways to explicitly detail what each section did. These comments can be found before each important section of code, explaining it's purpose and post-conditions.

Reading from a file

As mentioned in the introduction, I couldn't find any relevant information on prompting for a filename at runtime and subsequently opening nor reading from it. If the user wishes to use the program with values from a file, they can use file redirection from the command line to accomplish it. This can be done with “./roman < filename”.

Reflection

Re-engineering vs. re-implementing

Were I to start this process from scratch, I would opt instead to re-implement the program, even if it were to be done in Cobol. The algorithm itself is not at all complex, and comprises only a very small part of the overall program. I am confident that I could write this kernel of the algorithm in the original **conv** module and simply re-implement the read/write calls that surround it.

Problems encountered

Documentation for the various Cobol standards can be tricky to interpret, and forums abound with bad information. I imagine this is due to the enterprise nature of the language, and that it's uses at large companies usually consisted of a large amount of reference material internal to the company. Unfortunately, this means that I had a lot of trouble looking up features such as the aforementioned opening of files at runtime. The documentation provided in the class notes was very helpful, but I would think that anyone trying to learn Cobol without such a reference would be very lost.

A separate problem was one of interoperability of standards. On my personal laptop, I downloaded a copy of the OpenCobol compiler which appears to be a frontend to gcc. On the lab computers, we are using a different compiler that has subtlely different warnings and invocations that mine. That meant changing a lot of command line flags at the last minute in order to ignore warnings generated by “type-punned” pointers. The full set of command line flags used can be found in my Makefile.

Program length

The final program consists of two files, totalling 165 lines. Removing whitespace and comments compresses this down to 116 lines of functional program. The original program, similarly compressed consisted of 111 lines. As we can see, the program barely increased in size, but made drastic improvements in legibility, and functionality. Were we to omit the added features, the new program would be smaller than the original and still be cleaner.