Babylonian Square Root Re-engineered

by Matt Breckon

Let me be clear: Cobol is the worst language I've ever used. It leaves nothing to be desired and feels as if it purposely hinders the user. It was designed for businessmen and gives off the impression that it was also designed by businessmen. I hope in my future career I never encounter this beast again.

Process

Migration: Cobol 74 to Cobol 85

Converting the original Babylonian square root Cobol program to a more modern dialect proved to be easy once the algorithm was fully understood. To track the process of re-engineering the code, git was used for version control. This allowed myself to make small changes with descriptive commit messages to track the timeline of re-engineering the program.

At First Glance

Initially, the code was excruciating to decipher. The language already screamed at me how unreadable it was. The structure of different divisions to separate attributes, variables, and procedural code seemed tedious and unnecessary. The arbitrary syntax of X, 9, V, and S to distinguish data types was perplexing, let alone the use of the clause picture making even less sense. Also, the nonsensical level number of 77 when declaring variables still leaves me confused; I'm sure there is an answer to this, but my apathetic interest in Cobol stifles further investigation.

Once I was able to remove the horrid taste from my mouth, the first step was saying good bye to the eye-bleeding all caps and converting to a more modern lower case syntax. Such a simple and small change that drastically improved my desire to continue this project. This was followed by modifying all occurrences of the "picture" statement to the shorter "pic" purely out of preference.

The next step in modernization was turning if statements into proper blocks of code appending an "end-if" after the block of code. This step also included removing the more English-like nature of Cobol's code structure for arithmetic operators such as "is greater than" to "is >".

```
-  if in-z is greater than zero go to b1.
+  if in-z is > 0
+    go to b1
+  end-if.
```

The following commit was when I realized that the use of “is” is not syntactically necessary and was then removed from every line it was used.

The next thing in re-engineering to a more modern Cobol that needed to be tackled was restructuring any and all uses of the “go to” statement. This was when I realized it would be easier to rewrite the entire algorithm from the ground up instead of navigating the logic behind the “go to” headache.

Design Restructure

From here, the core of the program was completely redesigned. Paragraphs s1, b1, and s2 were completely removed and all that remained was the original read in from a file and the result output to the user.

Having an understanding of how the Babylonian square root algorithm worked allowed for a much simpler design structure of a paragraph named “sqrt” which made a call to a *function* called “calc” located below the “finish” paragraph that ended the program. By placing a paragraph after the occurrence of the “stop run” statement, I realized I could now utilize functions within Cobol.

```
+sqrt.
+  compute g = in-z / 2.0.
+  compute g2 = g + 1.0.
+  perform calc until g = g2.
                          ⋮
 finish.
   close input-file, standard-output.
+  stop run.
+
+calc.
+  compute n = in-z / g.
+  move g to g2.
+  compute g = (g + n) / 2.0.
```

Once the complexity of the previous design of the algorithm was gone, this restructuring from the ground up demonstrated an identical algorithm without the use of legacy “go to” statements.

Design Improvement

Once the program was more inline with Cobol 85’s structures and syntax, the next step was to remove the restraint of reading a file containing a number to calculate and instead to prompt the user for input. This was easily achieved through the use of the “accept” statement to retrieve a line from standard input and store it directly into a variable. From here, the use of file-input was obsolete to the program design and was removed from the code.

Once the user input was working correctly, an “if” statement was added to check the user’s input for a negative value. If true, it would print out an error and the program would perform the “finish” paragraph to close standard output and end the program.

With everything working nicely, the code was then refactored to call an external subprogram called “sub-sqrt” to perform the actual Babylonian square root calculation. This step was exceptionally easy as it only involved making a new file called “sub-sqrt.cob” containing the code previously within the main program, explicitly declaring “linkage” variables within the subprogram, and then making a call to the subprogram within the main program.

Finally, with everything working as intended, the last step involved looping the user’s input to allow the calculation of any amount of numbers until the user purposely quits the program. This portion was straightforward enough that it does not require any discussion; however, I purposely made the design decision that if the user inputs a negative number, it will exit the program instead of simply displaying an error and looping again. This allowed for both error checking and exiting the program to be all rolled into one.

Discussion

Learning Curve

Generally speaking, Cobol was the hardest language I’ve ever tried to learn after C. However, context is key; learning C was only difficult because it was my introduction to the world of programming. Learning logical operations, loops, functions, and so on for the first time takes time and a learning barrier to overcome. Once the foundations of programming were established, learning a new language has always been less of a chore and more of a

minor time sink. However, Cobol decided to flip that on its head by making everything a little more difficult than it needed to be. I personally found the language to be easy to learn as a whole, but difficult in grasping the little things. Little things such as the previously mentioned arbitrary syntax X, 9, V, and S for variable declaration and having to declare the size of numbers, writing code with an English-style syntax, and using statements similar to assembly code instead of C-like operations. Personally, I feel what made the language so difficult initially was combining the unfamiliar syntax of Cobol with the logical design and structure of the original program.

Structures

The structures, or lack thereof, within Cobol that initially made things difficult was the utilization of functions. The desire to simply declare a function that would only be executed when called is second nature in any useful programming language. Cobol, with its silly paragraph design, restricts this without some funny paragraph declarations after a paragraph containing “stop run”. Although this was my initial workaround to using functions, I realized this wasn’t proper and implemented an external module to call as my function. Modules are great in any language; however, modules being the only way to structure a function is horrible design.

Furthermore, Cobol’s data structure leaves little to be desired. Instead of declaring a variable as an integer or floating point, a number is instead declared as a catch-all type where the maximum number of digits must be declared. All of this too in the context of a “picture”, which still makes little sense to me.

Overall, I rate Cobol a solid 3/10.