COBOL — Jason Nguyen (1013950)

Ah—yes—one of my favourite iterative improvement algorithms: the Babylonian method of square roots. A *divide-and-average* algorithm, the Babylonian square root can reach deadly levels of accuracy in less than 30 iterations even with large (~10 digit) radicands. I learned a lot about COBOL and the square root method during this assignment; re-engineering SQRT.COB was quite fun despite the many idiosyncrasies of COBOL.

The original code was quite archaic—it was created in an era that fussed about capitalization and maintaining the fixed format. Furthermore, the endless flurry of **go to** statements made tracing through the program a massive pain. I'll admit, it took me a long time to realize that the first half of the code was just for print formatting. Here is how I re-engineered this famous algorithm.

1. Aesthetics—Capitalization and Variables

Just like many punch card languages, everything in COBOL was formatted to CONSTANTLY SCREAM AT YOU. So, for the betterment of my sanity, uncapitalizing the code was the first thing I did. There were also several vague variable names that I changed. It was the first thing I thought of after converting the program to lowercase, as changing the names had no effect on the logic. I prefer to make as many of these idempotent code changes as I can.

The readability of the program improved as I made some variable name changes:

- **Z** was the user's input. I renamed it to **userInput**, a string variable. **userInput** would be entered by the user where I would parse it into **radicand**, a numeric variable.
- **X** and **Y** represent the previous and next guess, respectively. Because we are analyzing the error between the previous and current (next) guess in order to gauge accuracy, we need to keep track of the previous. I renamed **X** and **Y** to **guess** and **nextGuess**, respectively, but in the final rendition of the program I opted for **prevGuess** and **guess**.
- **DIFF** represents the epsilon or intended accuracy of the final answer. I renamed this to epsilon at first, but later hardcoded the value as an implementation detail—the user should have no say this this.
- **TEMP** represents the difference between the current guess and the previous guess. This is then used in the equation (TEMP / (guess + nextGuess)) in order to calculate the relative error between the two guesses, needed for epsilon accuracy judgement. In order to better convey the intentions of this variable, I renamed this variable to **guessDiff** at first, but later felt that it was unnecessary, so I removed it completely.
- **K** was the iterator variable for the **perform varying** (**for**) loop. It had a time-out of 1000 iterations. At first, I renamed it to **i**, but later on, I ended up removing the loop entirely—with a 10 digit number, the algorithm terminates on a small epsilon in less than 30 iterations; in order to actually reach 1000, we'd have to input 600 digits!!
- All other variables **WRITTEN-LIKE-THIS** I converted to be **writtenLikeThis**.

2. Inlining and Renaming Paragraphs

I found it strange how many constant print message records there were. There were records for the title line, the separator "underline", the column headings, among others. As each of these records were usually called *only once* and took up several lines in declaration, I saved about 20 lines removing these unnecessary statements and instead printed them inline as needed using **display**. Other formatted lines still used **write** for the time being, but in overhauling my inputting method, I also removed those.

In addition to inlining the constant table drawing, I renamed a few paragraphs in order to make tracing the program easier.

- **S1** became **GET_NEW_LINE**
- **B1** became **EXECUTE_BABYLON**
- **S2** became **MAKE_NEXT_GUESS**
- **E2** became **NEXT_ITERATION**.

Also, I removed **guessDiff** (formerly called **TEMP**). All it did was take the difference of the two values and make itself positive if it was negative. As this was basically an **abs()** call, I instead used the built-in equivalent, **abs()**, and inlined it at **MAKE_NEXT_GUESS** (**S2**).

```
MAKE_NEXT_GUESS.
    compute nextGuess rounded = 0.5 * (guess + radicand / guess).
    if (function abs(guess - nextGuess)) / (nextGuess + guess)
        is greater than epsilon go to NEXT_ITERATION.
    move radicandIn to outZ.
    move nextGuess to outY.
    write outLine from printLine after advancing 1 line.
    go to GET_NEW_LINE.
```

3. Integration of Paragraphs and Removal of Fallthrough

This part of the project had a lot of trial-and-error. Thankfully, because I still had file input, testing was automated. I made sure to save a copy of the correct output so that every time I compiled, I could run *cmp* or *diff* to immediately smoke test any immediate mistakes.

Within the **GET_NEW_LINE** paragraph (formerly **S1**), there was a **go to EXECUTE_BABYLON** (**B1**) call, which I embedded directly into the **GET_NEW_LINE** (**S1**) paragraph. This was the easiest paragraph integration—later ones were tricky.

What really annoyed me about this program was how much fallthrough there was. Usually in a program, a successful operation and an unsuccessful operation would be separated by **if-else**, but the logic in the **GET_NEW_LINE** (**S1**) paragraph didn't permit this, as it transferred control explicitly:

if the input radicand was greater than zero, go to B1. Print error message.

Meaning that if the number was valid for sqrt(), it would **go to B1** and *never come back*. If it didn't **go to B1**, it would just **fallthrough** to printing the error message, making the user go through mental gymnastics to realize it was an implied **else**. In most other languages, we would use **if-else** to convey meaning to the reader that it is an alternate path. But this wasn't the case. I removed the fallthrough here and did so for later occurrences as well.

Before and After:

```
GET_NEW_LINE.
    read inputFile into lineStruct at end go to finish.
    if radicandIn is greater than zero go to EXECUTE_BABYLON.
    move radicandIn to otZ.
    write outLine from errorMessage after advancing 1 line.
    go to GET_NEW_LINE.

EXECUTE_BABYLON.
    move epsilonIn to epsilon.
    move radicandIn to radicand.
    divide 2 into radicand giving guess rounded.
    perform MAKE_NEXT_GUESS thru NEXT_ITERATION varying i from 1 by 1
       until i is greater than 1000.
    move radicandIn to outpZ.
    write outLine from abortMessage after advancing 1 line.
    go to GET_NEW_LINE.
```

```
GET_NEW_LINE.
    read inputFile into lineStruct
        at end move zero to eofSwitch
    end-read.

    if eofSwitch is = 0 go to finish.

    if radicandIn is greater than zero
        move epsilonIn to epsilon
        move radicandIn to radicand
        divide 2 into radicand giving guess rounded
        perform MAKE_NEXT_GUESS thru NEXT_ITERATION varying i from 1 by 1
            until i is greater than 1000
        move radicandIn to outpZ
        write outLine from abortMessage after advancing 1 line
        go to GET_NEW_LINE
    else
        move radicandIn to otZ
        write outLine from errorMessage after advancing 1 line
    end-if.
    go to GET_NEW_LINE.
```

4. More Paragraph Restructuring

First, I moved the **FINISH** paragraph into the **GET_NEW_LINE** (**S1**) paragraph. Then, I moved the **NEXT_ITERATION** (**E2**) paragraph into the **MAKE_NEXT_GUESS** (**S2**) paragraph, as all it did was transfer the current guess value into the previous guess value (in order to make way for the *new* current guess). This proved to be very difficult due to the way the control flow worked with the **PERFORM VARYING** loop in **B1**; the way that control transferred back to the loop ended up putting me down a rut for a few hours. Eventually I managed to get it working.

```
if eofSwitch is = 0 go to finish.
```

```
if eofSwitch is = 0
    close inputFile, standardOutput
    stop run
end-if.
```

```
MAKE_NEXT_GUESS.
    compute nextGuess rounded = 0.5 * (guess + radicand / guess).
    if (function abs(guess - nextGuess)) / (nextGuess + guess)
        is greater than epsilon go to NEXT_ITERATION.
    move radicandIn to outZ.
    move nextGuess to outY.
    write outLine from printLine after advancing 1 line.
    go to GET_NEW_LINE.

NEXT_ITERATION.
    move nextGuess to guess.
```

```
MAKE_NEXT_GUESS.
    compute nextGuess rounded = 0.5 * (guess + radicand / guess).
    if (function abs(guess - nextGuess)) / (nextGuess + guess)
        is greater than epsilon
            move nextGuess to guess
            exit paragraph
        end-if.
    move radicandIn to outZ.
    move nextGuess to outY.
    write outLine from printLine after advancing 1 line.
    go to GET_NEW_LINE.
```

5. The Crux of this Assignment—Embedding S2 into S1

This was the hardest part of the assignment for some reason. Looking at it now it looks braindead, but as we all know, hindsight is 20/20. Both **GET_NEW_LINE** (**S1**) and **MAKE_NEXT_GUESS** (**S2**) had **go to GET_NEW_LINE** (**S1**) at the end of the paragraph. Tweaking the unconditional jump so that control stopped at the right place was annoying.

After many tries, I managed to integrate **MAKE_NEXT_GUESS** (**S2**) into **GET_NEW_LINE** (**S1**). In doing so, I managed to remove another fallthrough. Originally, it would tell you to

perform MAKE_NEXT_GUESS until i is greater than 1000. Print error message.

which had the same problem as earlier. Success in going to **MAKE_NEXT_GUESS** (**S2**) meant that we would never end up on the other side of this **perform** loop—control would explicitly be thrown back to the **GET_NEW_LINE** (**S1**) paragraph—tracing through enough iterations for the reader to reach that conclusion was infeasible. As such, if we never ended up back in **GET_NEW_LINE** (**S1**), we would have to *fallthrough* yet again and print the error message as an implied **else**. I made a shoddy **else** statement here (to be fixed later). But eventually I had it down and successfully put **S2** into **S1**. I think I got McDonald's after.

```
GET_NEW_LINE.
    read inputFile into lineStruct
        at end move 0 to eofSwitch
        exit paragraph
    end-read.

    if radicandIn is greater than zero
        move epsilonIn to epsilon
        move radicandIn to radicand
        divide 2 into radicand giving guess rounded

        perform varying i from 1 by 1 until i is greater than 1000
            compute nextGuess rounded = 0.5 * (guess + radicand / guess)
            if (function abs(guess - nextGuess)) / (nextGuess + guess)
                is not greater than epsilon
                    exit perform
            else
                move nextGuess to guess
            end-if
        end-perform
        if i is not greater than 1000
            move radicandIn to outZ
            move nextGuess to outY
            write outLine from printLine after advancing 1 line
        else
            move radicandIn to outpZ
            write outLine from abortMessage after advancing 1 line
        end-if
    else
        move radicandIn to otZ
        write outLine from errorMessage after advancing 1 line
    end-if.
```

6. Timing Out the Timeout

I completely removed the 1000 time out. It was useless. I created a regression using Microsoft Excel when I found out that the Babylonian square root of 10-digit numbers could be within 5 – 6 decimal places of accuracy in less than 25 iterations. After a bunch of data sampling in the COBOL program (as well as a C program I made on the side), it turned out the number of iterations needed grew *linearly* with the number of digits. This made sense, as adding another digit was exponential, whereas the dividing step of the Babylonian algorithm was logarithmic. They cancelled each other out, and it turns out—you would need a whopping 600 digits to reach the 1000 iterations required by the timeout. So, I removed it.

At first, after I made this discovery, I made the **perform** loop unconditionally iterate 1000 times. I figured it would be enough accuracy, and yes it was. But it seemed somewhat excessive. I know today's computers are lightning fast, but to suit the spirit of COBOL, I ended up hardcoding an epsilon accuracy check of 0.000001 so it wouldn't go 1000 times.

```
perform varying i from 1 by 1 until i is greater than 1000
    compute nextGuess rounded = 0.5 * (guess + radicand / guess)
    move nextGuess to guess
end-perform
```

```
*> Iterate until we are below a threshold for absolute error
perform with test after until function abs(guess - prevGuess) < 0.000001
    move guess to prevGuess
    compute guess rounded = (prevGuess + radicand / prevGuess) / 2
end-perform
```

7. User Input and Polishing

The program was working—and was a huge step up from the older dialect version. But at this point the main part of the program called **GET_NEW_LINE** until it reached the end of the file. I ended up overhauling the program such that input would be taken as such:

- The user could enter as many numbers as they wanted
- The program would continually prompt the user for numbers one at a time
- A sentinel value of "0" or "q" would be entered if the user wanted to exit

In addition to removing file input and adding user-only input, I isolated a lot of the stuff in **GET_NEW_LINE** (**S1**) and moved it to the main portion of the program, making sure only the Babylon-related calculations remained. I renamed the resultant paragraph to **babylon**. Now, the "main" of the procedure division would handle the user input, error handling, etc. so that calling the **babylon** paragraph would serve only a single responsibility: performing the actual calculation. Later on I encapsulated the input/error handing as **calcSqrt**.

```
*> 1. Ask for user input
display "Enter number ('q' or '0' to exit): " with no advancing.
accept userInput end-accept.
```

8. Modularization

In an alternate universe, *sqrtbabyex.cob* is a modularized version of *sqrtbaby.cob*. There were two major changes:

- Instead of **perform babylon**, the calculation subroutine became: **call "squareroot" using radicand, answer**.
- **guess** and **prevGuess** became implementation details embedded within the external function file. My program defaults to halving the input number (radicand) as the first guess—the user was never given an option to seed a guess themselves. As a result, by moving the actual **babylon** paragraph logic to a separate file, it also ended up halving the number of variables in the main program.

userInput and **answer** were treated as the input and output variables of the **squareroot** external function, respectively. This was much cleaner than having to deal with iterator variables, guess variables, and difference variables (from the original program).

The result of this step is realized in *sqrtbabyex.cob* and *squareroot.cob*.

```
else
    *> 6. Calculate
    perform babylon
```

```
else
    *> 6. Calculate
    call "squareroot" using radicand, answer end-call
```

That wasn't necessarily the end though. I made a bunch of small changes (which is why these screenshots don't match up entirely with the finished program) and danced back and forth between design decisions that likely didn't make a big difference in the end. The biggest decision I made was to accept the user's input as a string rather than a number—this allowed me to have a lot more control with error handling, which is probably the biggest difference you'll notice between these screenshots and the final product.

My Thoughts on COBOL

I liked COBOL to be honest. It isn't conventional, but it does have its uses. I think the answer is obvious as to how I fared in learning the language—COBOL threw a lot of my existing paradigms out the window and teaching myself the language made me re-think a lot of how programming works.

I thought Ada's strictness was annoying, but COBOL takes the cake as the most annoying language of this semester. Having all variables be inherently global threw me off and having to describe each variable in the form of Z, V, X, 9 was annoying. The former meant that encapsulating function calls parameters didn't really exist (as everything was global) and the latter meant I had to define (and test) my own precision metrics for each numeric variable I used. I guess it gives the end-user the power to be very specific with the type of data involved, but it was hard to learn at first.

What annoyed me about control flow in COBOL was the fact that **if** statements were binary—they were just **if**-**else**—there was no concept of an **else if**—you'd have to nest *another* **if** statement within the **else** clause. It made indenting very annoying, but I got over it. In the end, I didn't really need to nest many statements in my re-engineering, so I am thankful for that.

I liked a few things about COBOL. First, I really like how file I/O is streamlined in this language. You can define a line to have any representation of data and then immediately read it into a file. The original COBOL program made heavy use of implied decimal points (using **V** in the variable declaration) because the numbers in the input file were always fixed precision:

+123456789812345600100

- the first character of each line was the sign of **Z**
- the next ten were the characteristic (the value before the decimal) of **Z**
- the next six were the mantissa (the value after the decimal) of **Z**
- the last five characters were the epsilon (separate variable, **DIFF**)

Being able to instantly read all of this with a single **READ INPUT-FILE INTO IN-CARD** statement was a very powerful aspect of COBOL that I really liked. In my research of this project, I made an equivalent program in C that read in files the same way. It was terrible and the function required to emulate the line above (**READ INPUT-FILE INTO IN-CARD**) took an excruciating amount of time to emulate in C—mostly due to the implied decimals!

Another thing I liked about COBOL is how English-like it is. You could write a statement like:

```
if x > 5
   compute x = x + 5
end-if
```

but you could also write it like this:

```
if x is > 5 then
   add 5 to x
end-if
```

thus increasing readability and allowing laypeople to trace through your code.

Overall, COBOL was fun to learn—even if it was super frustrating at times.