Programming in the Past

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

January 22, 2023

1 Log

1.1 Predicton

I am predicting that it will take me around **20 hours (average of 4 hours per programming language)** for me to learn Fortran, COBOL, BASIC, Pascal, and Scala and build a Caesar cipher in each language. This is an extremely rough estimate as I have never used any of these programming languages before and will have to learn each of them starting with "hello world." I am sure that I will have moments of staring at my computer screen for extended periods of time due to a weird nuance or gimmick in at least one of these programming languages that is not common in more modern languages. However, despite my lack of familiarity with these languages, I am hoping that there will be some nice similarities between them, so my approach to writing the Caesar cipher does not drastically change between them, and each implementation can easily be compared to each other on an even playing field.

1.2 Progress Log

Date	Hours Spent	Tasks / Accomplishments / Issues / Thoughts
January 18	3 hours	I built the entire Caesar cipher in Fortran, which was a bit frustrating but not the worst thing in the world.
January 19	3 hours	I started working on the Caesar cipher in COBOL. This has been exponentially more difficult than Fortran as some of the language's tiny details made me stare at a computer screen for hours without any resources because there were no error messages or warnings.
January 20	0.25 hours	I modified my Fortran implementation to account for negative shift amounts and shift amounts greater than 26.
January 20	1.5 hours	I finished writing the Caesar cipher in COBOL and dealt with more of COBOL's small nuances that did not provide error messages or warnings. Luckily, it did not take me nearly as long to find the issues today as it did for me yesterday.
January 21	1.5 hours	I wrote Caesar cipher in BASIC without much trouble.

1.3 Final Results and Analysis

2 Commentary

2.1 Fortran

2.1.1 My Thoughts

Building the Caesar cipher in Fortran was a challenging experience due to its strict rules regarding code organization and how functions and subroutines work. First, regarding code organization, I did not like having to declare all used variables at the top of a subroutine. In many other languages, the purpose of variables can oftentimes be inferred by the location in which they are declared. For instance, a variable declared by a loop often means it will have an important role in the loop, usually as a loop increment variable. However, in Fortran, since the variables were declared at the top of each subroutine and program, the readability of the language is hurt because the variables are all clumped together, and the language is tougher to write because you have to be careful in making sure that comments are written to explain what each variable will be used for. In the end, this organization did clean up the rest of the subroutine, but I would have rather declared the variables in more logical spots to go along with the flow of the program.

Another issue that I had with Fortran was with functions and returning values. I initially wanted to use functions to take in the original string and return the output string for encrypt and decrypt. However, functions and character objects were really not working nice as I received a bunch of errors, for which Google searches were not able to help me as the language has not been widely used since the inception of the internet. Some of these errors included but were not limited to an "entity with assumed character length at (1) must be a dummy argument or a PARAMETER" and functions that were explicitly marked to return a character (string) were returning a REAL and caused type mismatches at compile time. In the end, the best resource on the internet was the quick start guide on Fortran's official website, whose advanced examples provided me with some inspiration to use a subroutine and pass in the variable that would be modified and used as an output. Additionally, I wanted to make the function/subroutine support characters of an unknown length to work with all possible inputs. However, despite a lot of forums online having examples with this as the situation, the code just never compiled or worked for me. Therefore, I chose to have one of the parameters of my subroutines to be the length of the string, so the variable declarations at the top of the subroutine would be able to provide a known length.

Overall, aside from the variables being declared up top, Fortran is quite a clean programming language in terms of readability as a lot of the control structures are similar to those in more modern languages. However, the lack of clarity for how functions work with strings is extremely frustrating and significantly hurts the writability of Fortran as I had to find a way to finesse a solution using subroutines and pass in the output variable, even though I would have preferred to have better organized code that returns a result from a function.

2.1.2 Google Search History

- fortran hello world
- function in fortran
- string type in fortran
- get length of string in fortran
- Entity at (1) has a deferred type parameter and requires either the POINTER or ALLOCATABLE attribute
- pass in character to a function fortran

- Entity with assumed character length at (1) must be a dummy argument or a PARAMETER
- return string from function fortran
- iterate through characters in a string fortran
- fortran print string to include value of variable
- fortran mod

2.2 COBOL

2.2.1 My Thoughts

Based on my experience with writing the Caesar cipher in COBOL, I have determined that it is not a user-friendly programming language in the slightest of ways. COBOL is extremely readable and self-documenting as every operation and section is explicitly defined by the programmer. This is the only decent thing going for it as the rest of my experience was dreadful due to its poor implicit type conversions and its lack of stack frames for functions.

First, the implicit type conversions for passing in parameters to functions was a nightmare to deal with by itself. In each function, all variables are defined with their types, and numeric types need to state the number of digits the variable will take up. However, oftentimes as a programmer you want to pass in a raw number to a function as the number will not be used anywhere else, so storing it in a variable would not be very useful. However, COBOL does not interpret these values well against defined types. For instance, an unsigned integer parameter that needs to store 2 digits must take in a number like "08" and not just "8." The main problem here is that the compiler does not say anything is wrong with the input. Instead, it can print out that it has an "8," but all mathematical operations with the value will be completely off. The same goes with signed numbers as well, but they basically do not even work as "-001" in a 3-digit signed integer comes back as "-00" and the "1" is missing, not to mention wild math results as well. Unfortunately, the only way to really deal with parameters is to pass in a variable that has the value you want to pass in. This ensures type safety as an "8" will be the same as "08," which contradicts the problems stated before. It almost reminds me of truthy and falsy values in JavaScript as you sometimes cannot tell what the output will be because the compiler just does its own thing without telling you what it is going to do.

Another serious issue that I faced was the lack of stack frames with function calls. In my experiences with many other programming languages, they all seem to use stack frames for function calls. That is, any variable defined in a function exists for no longer than the duration of the function. This means that the next time the function is called, the variable will be reinitialized with its starting value every time. Unfortunately, this is not the case for COBOL. Based on what I am able to tell, all variables defined in data divisions exist for the entirety of the program's lifespan. Therefore, the state of the variables for a function will be preserved across multiple function calls. COBOL lets you initialize variables in the data division with a value, but it is basically useless for functions that need to make sure that the variables get reset at the start of the function call anyway. This is another example of COBOL making itself really difficult to use because it provides functionality for features that do not make sense, at least relative to today's standards and paradigms.

Writing my Caesar cipher in COBOL was by far one of the most challenging things I have done as a programmer. In fact, COBOL's poor writability made Fortran look good, which demonstrates how bad my COBOL experience was because I did not originally think highly of Fortran. Although COBOL's code is self-documenting, it was nothing that could not be done with comments in Fortran and could not make up for the excruciatingly difficult task of writing COBOL code.

2.2.2 Google Search History

- cobol hello world
- cobol column rules
- functions in cobol
- user-defined functions in cobol
- data types in cobol
- iterate through characters in a string cobol
- string copy cobol
- cobol function with both linkage section and working storage
- convert character to ascii code cobol
- get character from ordinal value cobol
- cobol addition is not right
- negative numbers in cobol
- string length cobol
- working storage section variables are not wiped after a function call and their state persists between function calls

2.3 BASIC

2.3.1 My Thoughts

BASIC, as implied by its name, is a very simple programming language that, at times, felt too orthogonal due to its lack of key features that are included in almost every other programming language. Despite these flaws, my experience writing the Caesar cipher in BASIC was much better than that for Fortran and COBOL.

User-defined program structure in BASIC was nice and flexible, but prevented any form of documentation. In BASIC, every line of code is preceded with a number that represents which line in the program the code should be inserted into. Although it was annoying to micromanage the code and make sure every line number was consistent, I did not mind this too much because I enjoy seeing these lower-level details. I was also thankful for a text editor because I know I would have struggled had I manually entered each line into an Apple II computer and rewrote the lines if I ran out of space or wanted to clean up the code organization. For subroutines, I used the practice of every 1000 lines was a new subroutine (i.e., encrypt started on line 1000, decrypt started on line 2000, and solve started on line 3000). I personally thought this was really cool because I was able to explicitly define how I wanted my code to be organized. However, calling these subroutines was through the line number, so the code was not self-documenting and comments were needed to label which subroutine was being called in each instance. This was really unfortunate as the rest of BASIC's syntax is extremely readable and self-documenting.

Similar to the code organization, I had mixed feelings about the simplicity of variables in BASIC. I really liked how all identifiers had to end with either a '%' to show it is a number or a '$' to show that it is a string. This was really nice because variable declaration was simple as the data type is implied with the name and there was never a question of a variable's type when writing the program because it is self-documented in its name. Despite this really nice syntax, BASIC provides no variable checking at runtime. For instance, if

you omit the '%' or '$' or misspell a variable, the BASIC interpreter assumes that you are trying to reference a new variable. Thus, it creates a new variable and initializes it to 0 without telling you. When writing my Caesar cipher, this "feature" screwed up program execution, and it was very difficult to debug since there were no warnings or error messages.

Aside from some of these minor issues I had with BASIC, there were a few more major issues that deterred from the language's readability and writability. First, since each line of code starts with the line number in the program, there is no formatting in BASIC to line up when different blocks of code start and end. This became very challenging to debug when I forgot an endif as the code was not formatted, which hurt the language's readability and writability as well. The other serious issue with BASIC was its lack of scope for variables. All variables are global and can be accessed from anywhere in the program. This is very dangerous when dealing with subroutines and made the code quite messy when trying to reuse input and output variables that I dedicated to the various subroutines. Considering that the program is a Caesar cipher and was messy because of variable reuse, I cannot imagine writing more complex programs in BASIC and micromanaging all the variables and how and when they are used.

Overall, the nice syntax of BASIC made it much easier to both read and write code compared to Fortran and COBOL, the lack of formatting and variable scope makes the language much more suited for smaller programs with few moving parts as it does not scale up well with increased complexity.

2.3.2 Google Search History

- hello world basic programming language
- chipmunk basic
- https://www.youtube.com/watch?v=7r83N3c2kPw
- mid$ basic

2.4 Pascal

2.5 Procedural Scala

3 Conclusion