# Programming in the Past

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

January 28, 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 the Caesar cipher in BASIC without much trouble. |
| January 26 | 0.75 hours | I wrote the Caesar cipher in Pascal with even less trouble than BASIC. |
| January 27 | 0.75 hours | I wrote the Caesar cipher in Scala, which was unsurprisingly almost identical to how it would have been written in Java. |

### 1.3 Final Results and Analysis

Overall, this assignment took me 10.75 hours to complete, which is about half the time I initially expected. Although I was on pace for 20 hours after Fortran and COBOL, the time required for me to write the Caesar cipher in BASIC, Pascal, and Scala was much shorter than originally anticipated. I initially predicted about 20 hours because I did not know what to expect from the older programming languages and was unsure how long it would take for me to learn enough of each language to put together the respective programs. Furthermore, the results from Google searches were much more detailed and useful for the last 3 languages because they are not as old and have some good online resources that demonstrate the basics of the languages. Also, since they are newer, BASIC, Pascal, and Scala all have grammars that are much closer to other popular programming languages today, so more similarities were able to be recognized and taken advantage of. On the other hand, since there is no language today like COBOL, it is in a league of its own and had to be figured out in its entirety with no assumptions of similarities to other languages. Lastly, as described in my prediction, I wanted the solutions to be as close to each other as possible. Thus, I did not try to do anything crazy with the solutions and kept it simple throughout. Thus, once I wrote the Caesar cipher in Fortran, the assignment became a task of trying to translate the Fortran code into other languages rather than trying to create a new solution from scratch 5 times.

## 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

### 2.4.1 My Thoughts

My experience in Pascal was overall really good and I genuinely have very little to complain about. I found the code to be a great example of elegant simplicity. Pascal contained all of the basic programming constructs, including some of the more modern ones, while being less verbose than the other programming languages so far.

First, Pascal's clean and simple grammar made it both really easy to read and write the code. Some of the issues in languages like Fortran and COBOL was that they required all variables to be declared at the top of functions. Fortran did not have anything to separate these declarations and the actual program, and COBOL was extremely detailed and also did not exactly line up with the actual execution of the program. Although Pascal also requires variables to be declared at the top of functions, it has 2 very clean sections to do so, "const" for constants and "var" for everything else. The inclusion of these sections makes the code more readable as it is self-documenting within the sections as well as extremely writable as I was able to write a lot less code to do the same thing as COBOL. Another part of the language's grammar that is unique relative to the languages used so far is the walrus assignment operator (:=). Rather than using the single equal sign (=) for assignment, Pascal uses the walrus operator, which increases readability as there is no confusion between assignments and comparisons and only comes at the cost of one additional character being typed, which is a miniscule cost relative to ease of which one can understand the code.

Another feature that I really appreciated was how functions deal with return variables. In most other programming languages, you can return from any point in a function as well as have multiple return points. In Pascal, however, the returned variable takes the same name as the function and automatically gets returned at the end of the function. This design pattern upholds the idea of functions being one-way in and one-way out and makes the code extremely easy to understand and follow. I do, however, wish that the returned variable had a built-in name that was not the same name as the function as recursive functions may be hard to read because you can have multiple instances of the same name in a single line of code that both mean completely different things. Despite this small complaint from me, the improved design pattern carries a much larger weight than the name of a variable and, therefore, functions are much more readable and writable than most other programming languages today.

Overall, my experience using Pascal to write a Caesar cipher was really enjoyable due to its simple, yet feature-filled, grammar that made the code really easy to both read and write. I wish the language was more popular today because I found my experience to be better than some more modern languages that are widely used throughout the world.

### 2.4.2 Google Search History

- pascal hello world
- pascal function multiple parameters
- and operator pascal boolean expression

## 2.5 Procedural Scala

Scala, although very similar to Java, was a bit awkward to write in a procedural manner because its identity lies closer to a functional programming language than a procedural one. Although it tries to be a better version of Java, its increased simplicity makes the syntax feel a bit constrained and challenging to write at times.

First, Scala's simple syntax makes it a bit harder to both read and write in certain situations. Looping in Scala is similar to the other languages used in this assignment as it condenses the variable declaration, comparison, and post-iteration adjustment into a few words. However, the syntax is inconsistent with declaring variables everywhere else in the program, which I do not really understand. Speaking of variable declaration, Scala's syntax for declaring variables prevents one from understanding if a variable is a constant or not. Variable declaration for constants begin with "val" and variable declarations for mutable variables begin with "var." This one letter difference does not make it very readable as one can easily glance over this error (I did several times) and is not very writable because the change is so minor that I do not feel like I am expressing the purpose of a variable with a single letter.

Fortunately, since Scala is a newer programming language, it provides developers with a wide range of tools out of the box. More specifically, its compiler is designed to increase understanding of the problems that it finds rather than just building programs. For instance, when I was trying to assign a new value to a variable declared with "val," the compiler initially threw an unhelpful error of the problem and suggested that I use the "-explain" flag for more details. I then tried to recompile the project with the "-explain" flag and the compiler not only showed me the line of code that had the error, but also explained why it was throwing the error and how to fix it. This was extremely useful as it helped my fix the problem as well as understand what I did wrong so I could learn from my mistakes. This feature is not uncommon with compilers of newer programming languages (Rust is a great example) as it helps boost the writability of the language by providing the better tools for developers to quickly get the answer to their problem rather than having to perform a Google search on the error message and hope that someone else has previously had the error.

Overall, Scala was not a bad programming language for the Caesar cipher as it has a lot of similar features and syntax patterns as a lot of other programming languages. However, its oversimplified syntax hurts it at times and is quickly made up for by its compiler that was built to increase developer productivity.

#### 2.5.1 Google Search History

- scala hello world
- scala loop through a string
- scala string object
- absolute value scala
- scala convert int to char

## 3 Conclusion

Here is my final rankings for the 5 programming languages:

1. Pascal
2. Scala
3. BASIC
4. Fortran
5. COBOL

Starting from the bottom, COBOL was a near nightmare to work with because it was extremely difficult to write and, despite its verbose code, did not follow the code that I wrote and, instead, decided to do its own thing at times. Next, Fortran was not a bad programming language by any means, but it was very difficult to write and had a compiler that produced confusing errors to understand. In second and third place, I have Scala and BASIC, respectively. Although I enjoyed writing BASIC more than Scala, the unsafe program structure of BASIC was tricky to work with at times, while the Scala compiler provided me with a great experience despite me not entirely liking its syntax. Lastly, Pascal was my favorite programming language in this assignment because it demonstrates the perfect balance of simplicity in its code while also being detailed enough for the code to document itself without questioning what the code does.