I took the Programming Language Concepts course in 2019 at RIT. At the time, we used a piece of software called PLCC written by Dr. Timothy Fossum. This software was great in regard the teaching resources it provided, but it had some features lacking for students and contributors. My initial intention was to modify PLCC to make it less of a black box and more user-friendly, but I quickly learned that this modification would have a similar amount of work as making something new, which ended up being called LAnguage Processor & Synthesizer, otherwise known as LAPS.

Making LAPS gave me complete control of the architecture, feature set, and packaging. This allowed me to adhere to my goals of software modularity, comprehensible, and support for debugging and development in IDE’s. LAPS isn’t written as a typical piece of software; it’s written as a library which works together in a main file. So, anyone may come along and use a different scanner or parsing implementation. They can even remove the main and directly use LAPS integrated into their code. To enable this, the LAPS source code is heavily documented and is JavaDoc compatible.

Before starting development, I knew LAPS needed to be written in a language many undergraduate students and I were both comfortable with. PLCC was originally written in Python, but since Python can enable and even encourage bad coding practices, I figured Java was the next best thing.

Once that was settled, I began thinking about the design structure of what LAPS eventually turned into. I started by writing a custom Java class loader which would use annotations in a package-info.java to identify which classes would be used by LAPS. That ran me into a big problem which I was unsure how to solve. Packages weren’t considered “loaded” until a class from that package has been loaded. That meant that a user of LAPS needed to pass in the path of their package, making the class loading mechanism obsolete. I ended up replacing this with using the current working directory as part of the classpath and having the user pass in the fully qualified class name (e.g. path.to.classes.ClassName) of their top-level grammar rule. This took away a lot of overhead in file I/O and potential of insecure algorithms.

In the meantime, I added what I thought would be an example language which LAPS would process. In this example, I used my limited knowledge of annotations annotate classes in what seemed like a reasonable manor for the use case, defining grammar rules. Since I was waiting to compile until I had something useful to compile, I didn’t realize that annotations can only receive Strings and primitive types. That made the only way to pass in valuable arguments to annotations is to pass them by Strings, which have no compilation checks, other than having quotes surround them, making that not very IDE friendly and hard to debug. So, I ended up scrapping the idea of using annotations to define grammar rules in lieu of class constructors.

While that was in the works, I started creation of the recursive descent parser.