Spring 2020

LAPS Development Experience Report

Creating a language processor

Author: Geoffrey Casper

Advisor: Dr. James Heliotis

Rochester Institute of Technology DEPARTMENT of Computer Science

I took the Programming Language Concepts course (PLC) 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 that works together with a main program. 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 in the JavaDoc style.

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 the language 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 file 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.

Once I stopped making progress on the class loader, I started coding up some of my ideas for my parser and storage for an abstract syntax tree. At this point, I didn’t really know exactly what I was doing. So, I just wanted to get all my ideas into code. I quickly realized that the storage for the abstract syntax wasn’t necessary since that would be handle by each LAPS language specification.

In the meantime, I created what I thought would be an example language for LAPS to process. In this draft language specification, I used my limited knowledge of annotations to annotate classes in what seemed like a reasonable manor for the use case. That is, the annotations’ arguments defined grammar rules and tokens. Since I hadn’t compiled LAPS at this stage, I didn’t realize that annotations can only receive Strings and primitive types. That made the only way to pass in valuable arguments to the 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, in lieu of using annotations to define grammar rules, I ended up moving to a different idea of using class constructors instead. Although that had to change, the token annotation worked flawlessly under these restrictions and ended up in the current version.

While the example language was in the works, I started creation of a recursive descent parser implementation. I started out thinking of what operations I wanted to implement based on concepts of regular expressions and context-free grammars. I started out with the most basic type of parser I could think of, a token parser. This accepts a single token and returns its string value. The next parser that made the most sense was a parser which accepts a sequence of parsable types, i.e. tokens. The next most simple parser I could think of was a parser which would accept any of one of a set of parsable types, what I referred to as an OR grammar rule. This one really confused and frustated me, because I knew it could lead to ambiguity of which item in the set should be parsed first. I didn’t know it yet, but this would be the basic idea for the list of constructor grammar rules defined in @GrammarRule types.

As a biproduct of the architecture I have chosen, I need to create a class to help manage storage of tokens and parsers. This seems to be a common architecture for compiling, since tokens need to be accessible by both the parsers and the tokenizer. I called my implementation Resources for no reason other than I couldn’t come up with anything better at the time and it stuck. Since everything was stored so concisely, I ended up adding a feature to be able to save and load languages from a file, making it easier to pass around compiled language specifications.

At this point, I was still a bit frustrated with the struggle I was having with OR grammar rule, so I decided to take a step back and work on another portion of the project, the scanner, or tokenizer. This portion was much easier for me to develop since I’ve written a tokenizer before in PLC and I’ve analyzed the Scan class, which has a similar function, in the PLCC source code. All that had to be done was convert a stream of characters to a stream of tokens. I did this by retaining a list of token types and then looping over those tokens to hopefully find the longest match.

Meanwhile, I went back to working on the parsers, which were still missing a crucial feature, the ability to convert a @GrammarRule type into a parser. To achieve this, I looked at the documentation for Reflection in Java. To my surprise, it was much simpler to understand. Java Class objects provide interfaces for getting their fields, methods, and constructors. All I had to do regarding the parser was collect all the constructors and convert their parameters into grammar rules using the previously made sequential and token parsers, as well as this parser recursively.

Most of the project was done by this point. Things that still needed to be done were adding error messages, making a command-line utility to use LAPS, fixing bugs within the modules, adding some features to catch erroneous things which could be in a LAPS language specification, and refactoring of files to increase modularity.

One of the major bugs I ran into was that token strings weren’t given back to the scanner by the parser when parsing failed on a grammar rule. I ended up needing to add features of buffering the input to the scanner and being able to push unparsed characters back to the front of that buffer. In addition, I had to store all of the parsed token values in a stack for each parser.

Making the command-line utility was relatively straight forward. The only involved part was making the command-line argument parser. I ended up using a HashMap of characters to lambda functions for each parameter. The arguments are used to construct the environment for the LAPS modules. Then, it opens the stdin as the language input, makes a new scanner and parser, and executes the language.

There are still a few things I’d like to work on regarding LAPS. There exist better parser designs that I wish to implement. Some of the Java reflection code is intertwined with the parser code which goes against the modularity aspect of my goals. The interfaces I made for the parser and the scanner seem to be useful in the modularity aspect. Since LAPS is eventually meant to be used as a teaching tool, more example language specifications and assignments should be created. Overall, I’m very happy with what I created. LAPS is very easy to modify and grasp how it functions.