

Compilers Portfolio

William Roberts
risause4@gmail.com
Montana State University
Bozeman, Montana

Jessica Jorgenson
jorgenson.jess@gmail.com
Montana State University
Bozeman, Montana

J. Beckett Sweeney
jbeckettsweeney@gmail.com
Montana State University
Bozeman, Montana

ACM Reference Format:

William Roberts, Jessica Jorgenson, and J. Beckett Sweeney. 2020. Compilers Portfolio. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 2 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

Blank Description.

2 BACKGROUND

Blank Description.

3 METHODS AND DISCUSSION

3.1 Tools and Setup

To begin our project, we used the "Getting Started with ANTLR v4" guide provided by antlr.org.

We followed the necessary steps to set up ANTLR for Java on Windows and after working through a few minor bumps, we successfully got it working.

For version control, we have set up a GitHub repository so that the three of us can work remotely and safely have access to previous versions of our project.

As for team management, we have decided to use Trello. Its simple interface and powerful tools paired with us already being familiar with it makes it a perfect tool for our team.

Communication will take place on Discord, where we can easily send messages and files back and forth.

Both Trello and Discord have mobile apps as well so planning and communication can be done even when we are away from our computers.

3.2 Scanner

Scanners are powerful yet simple tools that constitute the first step and process in program compilation. The multi-step process of turning high-level languages like Java, Python, etc., into code that computers can understand begins with the scanner, possibly making it one of the most important parts, even in its simplicity.

Scanners are token recognizers (or tokenizers, lexers). They sift through the entire file and convert every piece of the language by their characters, such as "(" or "+" or "A", into a token. Once

converted, the scanner outputs the list of tokens. The process is quite simple, and utilizes simple regular expressions in logic. There are a few complications in this is, however. One complication is that every possible input must be defined. Another is that the tokens outputted may not be legal expressions in the given language. Scanners only recognize the tokens - they do not have an understanding of the program, or output that much. That job is up to the parser, which uses the tokens as input.

ANTLR (Another Tool for Language Recognition), is the tool we used to generate the source code for the lexer, rather than coding it all ourselves. ANTLR takes in as input a .g4 file, which contains the regular expressions that define the tokens within a language (in our case Little), and creates a program that will scan source files in that language. It is important to note that ANTLR also can generate parsers. We found it simpler to define the parser immediately, so we can more adequately test the first stages of our compiler (see Parser below).

However, a list of tokens isn't especially useful for our schooling assignment. We used the IntelliJ IDE and wrote a simple program using the high-level language Java to translate the list of tokens into a more readable format. We had to match this format exactly to how our Grading teaching assistant wanted it. There were several small challenges in completing this, mainly of which came down to outputting the tokens correctly.

IntelliJ was also useful for creating the .g4 file. With the ANTLR tool plugin installed, the IDE had an output tool that showed us the process of creating the scanner source code from the .g4. It also has a great auto-complete process that helped greatly in building our .g4. We also decided to create most of the Parser in this step, and IntelliJ has a powerful output tool for quickly generating Parse Trees and output from inputted files, which was incredibly helpful.

The process of preparing our Scanner began with the example grammar file we were provided by our instructor. This file, named "grammar.txt", contained all of the rules that we would need to implement for both the Scanner and the Parser, though they were not quite in the format we needed. The grammar file was carefully translated into the appropriate format needed for a .g4 file, which is what ANTLR uses for its Scanner and Parser grammars. We named our file "g.g4".

This process of translating took some time and was prone to error. At certain points we were confused regarding whether certain symbols were to be input directly into our g.g4 file, or if we were supposed to type their named counterpart. For example, should "(thing)" be included as "(thing)", or as "LPAREN thing RPAREN"? The answer seemed to consistently be the latter, which made this process less ambiguous as time went on.

In addition to translating the text from its given format into an acceptable .g4 format, we also needed to translate some English into

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

regular expressions for terms such as "IDENTIFIER" and "INTLITERAL". Some of these translations were simpler than others, though in the end we overcame these challenges.

The final step was to convert the list of tokens into a more readable format for grading purposes. Creating a Java Driver program turned out to be slightly more complicated than we had first imagined. Once the output was fine-tuned to the exact specifications outlined by our Grading Teacher's Assistant, we used the "diff" command-line prompt to determine if there were any differences between our outputs and the ones provided to us for guidance. This introduced another step of fine-tuning that took some time, and once we got it just right, we went on to programming a simple bash script to run all of the process in a single command.

The step 2 assignment detailed that we needed a script we could call with the name of a file and it will run the scanning and outputting process. We needed to be able call ./Micro and have this process completed, so a simple bash script would do. Ultimately

not many of us understood bash scripts, but it only took a little learning to be able to program such a simple program.

3.3 Parser

Blank Description.

3.4 Symbol Table

Blank Description.

3.5 Code Generation

Blank Description.

3.6 Full-Fledged Compiler

Blank Description.

4 CONCLUSION AND FUTURE WORK

Blank Description.