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, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

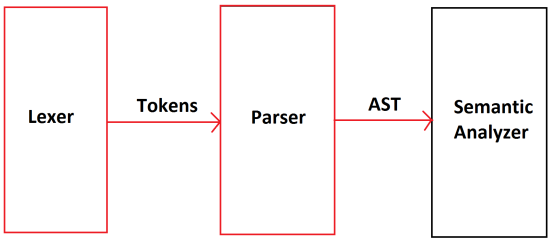

Figure 1: The Simplified process of language recognition.

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.

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
//Tokens
IDENTIFIER:         Letter LetterOrDigit*;
INTLITERAL:         Digit+;
FLOATLITERAL:       Digit+ '.' Digit+
            |       '.' Digit+
            ;
STRINGLITERAL:      '"' .+? '"';
COMMENT:            '--' ~( '\r' | '\n' )* -> skip;
```

Figure 2: An Example Regular Expression.

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 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

Parsers fulfill the second step of the compiling process, the step where actual understanding of the syntax and the program begins. Parsers are another great example of a simple yet powerful tool that follows simple logical processes to complete its job. Rather than using regular expressions like the Scanner, it builds upon

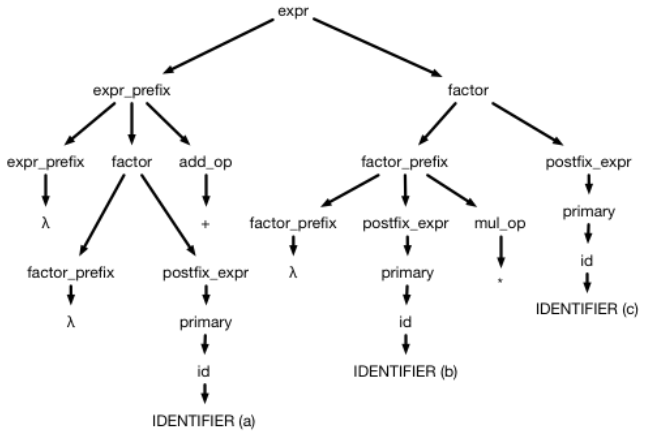

Figure 3: An Example Parse Tree.

those expressions in its own Context Free Grammar-a grammar that defines the entire parser.

It probably doesn't come as a surprise that there are many different ways to write a parser, and while each different parsing theory might yield the same result, they are written and created in very different ways. There are two (at least we learned) standard ways of writing a parser, either top-down or bottom-up. These methods mostly describe how the parsing process constructs a parse-tree, either starting at the "root" node (the beginning of the program) and building downward by predicting the next production to use, or starting with its leaves (the tokens) and building up by reducing the input program to its defined parts. Top-down parsing uses Left-most Derivation(LMD), usually defaulting to LL(1) or Left-to-right Left most Derivation Lookahead(1), which means it works from the left of the input to the right, looking 1 token ahead to determine which production to use. Bottom-up parsing uses Right-most Derivation (RMD), and with the process we learned in class usually defaults to LR(0) or Left-to-right Right-most Derivation lookahead(0), which just means it starts on the left of the input and goes right like LL, but it uses the right-most Non-terminal to break down the language, without needing to look ahead.

Since writing your own parser by hand is a costly and error prone process, we used a parser generation tool to help design our parser; the same tool we used to generate our Scanner: ANTLR. ANTLR uses the LL(*) (LL star) strategy in its definition for parser generation. LL(*) is a complicated process that uses the LL method for creating parse trees called Recursive Descent, along with multiple different strategies that define its "*". "The key idea behind LL(*) parsers is to use regular-expressions rather than a fixed constant or backtracking with a full parser to do lookahead,"(Fisher et. Parr). LR(*) combines prominent strategies like backtracking and REGEX, within parser generation to ensure that the generation process is as quick and powerful as needed. "The analysis constructs a deterministic finite automata (DFA) for each non-terminal in the grammar to distinguish between alternative productions. If the analysis cannot find a suitable DFA for a non-terminal, it fails over to backtracking. As a result, LL(*) parsers gracefully throttle up from conventional fixed $k >= 1$ lookahead to arbitrary lookahead and, finally, fail

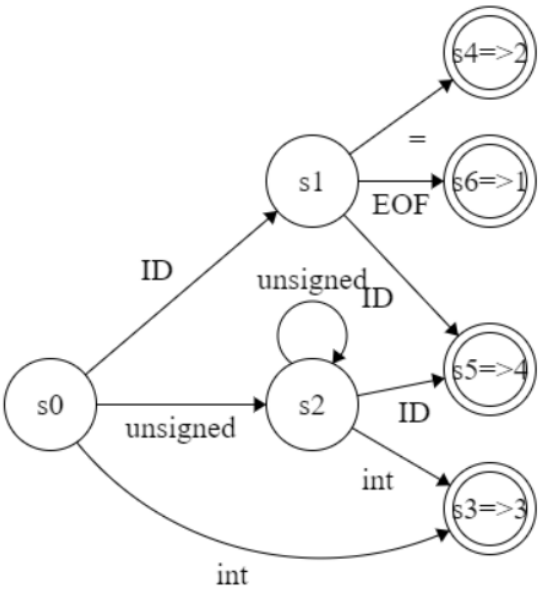

Figure 4: The Resulting DFA. Notation sn => i means "predict the i'th production."

over to backtracking depending on the complexity of the parsing decision,"(Fisher et. Parr). This process makes LL(*) very flexible and powerful, and ensures that the generator can create a parser for any given CFG. Fisher and Parr gave a great example of this process in action; Consider the non-terminal *s*, which uses the (omitted) non-terminal *expr* to match arithmetic expressions.

```
s : ID
| ID '=' expr
| 'unsigned'* 'int' ID
| 'unsigned'* ID ID
;
```

There is a need for arbitrary lookahead once the parser reads in an 'unsigned', meaning it needs to scan ahead until it finds a value that distinguishes between production 3 and 4. However, this lookahead "language" is regular, so the writers describe that LL(*) will create the following DFA in figure 2. If the lookahead language was not regular, the process would fail and use backtracking to continue the parsing step.

As you can see, the process for using LL(*) is in theory not too complicated, but far more powerful than any other LL method and stands up to many LR methods in strength. The advantage of LL(*) is that it achieves the functionality of LR methods while keeping the same error-handling capabilities of LL. LR is known for being terrible with erroneous input (because it only knows for sure if it is matching an expression), and that inhibits many of its uses and generality. Prediction is also just much easier to understand.

Ultimately our method of creating the Context Free Grammar for the Little language followed closely to the structure shown during class and found in The Definitive ANTLR4 Reference by Terence Parr. There were not very many difficulties in defining the Context Free Grammar for the Little language, and in fact the Parser was

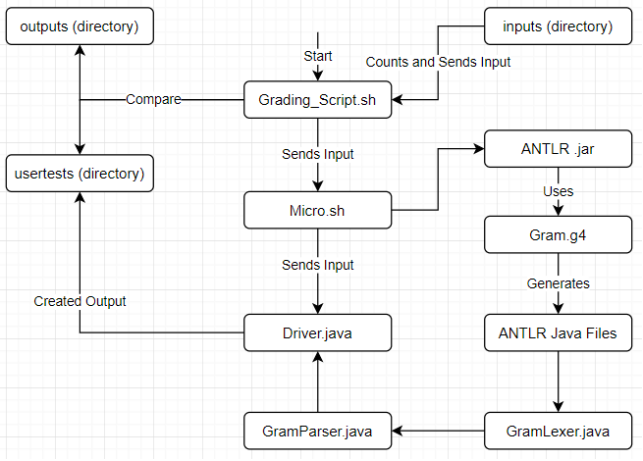

Figure 5: Workflow of our implementation

completed during our step 1, alongside the scanner. The primary challenge we had was developing the Java Driver to interpret accept and failure and bash scripts to run our tests.

Figure 3 shows the workflow of our implementation. It begins with the provided Grading_Script.sh which first initializes an empty "usertests" directory to keep program output. After that, it begins a for-loop that runs for each input found in the "inputs" directory. The grading script then calls our Micro.sh script, making sure redirect in the current input file. Micro then calls upon the ANTLR program to generate all of the necessary Java files using our Gram.g4 file. Once these files are generated, Micro compiles them, then moves on to compile Driver.java as well. Once everything is generated and compiled, Driver is run with the current input file redirected in as input. Driver will read the input file, create a Lexer from it, create a Parser from that Lexer, then run the parser to check if the input program is valid. If it's valid, it outputs "Accepted". If not, it outputs "Not accepted". On its own, Driver outputs to the console, but the grading script instead sends its output to an output file with the same name as the input file into the "usertests" directory. Once the output file is there, the grading script calls the "diff" function to compare it to the corresponding file in the "outputs" directory. If these two files are identical, the grading script prints out that they're identical. If not, it prints out what their differences are. This portion of the project is deemed successful if the outputs are identical for all 21 input files.

One issue we ran into when testing was that the project worked on our personal machines, but not the lab machines that we would be graded with. After several hours of working through this, we discovered there were a couple different things at play. The first was that our Micro.sh script would not run at all on the Linux machines in the lab. The solution to this problem, although strange, was to rewrite the script character for character in the Linux machine. This worked because Windows saves scripts differently than Linux, so even though they looked identical, they were functionally slightly different. The next issue we ran into was our Driver.java file rejecting all input files on the Linux machines. It turned out the way we handled reading the input file was what didn't work

```
Testing input file inputs/test1.micro
Files usertest/test1Test.out and outputs/test1.out are identical
Testing input file inputs/test10.micro
Files usertest/test10Test.out and outputs/test10.out are identical
Testing input file inputs/test11.micro
Files usertest/test11Test.out and outputs/test11.out are identical
Testing input file inputs/test12.micro
Files usertest/test12Test.out and outputs/test12.out are identical
Testing input file inputs/test13.micro
Files usertest/test13Test.out and outputs/test13.out are identical
Testing input file inputs/test14.micro
```

Figure 6: A sample output with all the tests being successful.

between machines. On our personal machines, we read the input file using Scanner.next(), which pulls the next token from a file until there are no more tokens. Along with this, we used several variations of System.lineSeparator(), which worked fine on our machines but did not work in the lab. Some adjustments made a few input files work properly, but any file with a comment in it still didn't work. After many more iterations, we finally landed on using Scanner.nextLine() instead of Scanner.next(), as well as manually adding "" to the end of each line while concatenating our input file to our string variable. With all of these steps complete, the entire process finally worked regardless of what machine we tested on.

Error handling took some significant work and several different strategies to get just right. We ended up using the strategy proposed by user Mouagip on stackoverflow. Their work can be found at *"https://stackoverflow.com/questions/18132078/handling-errors-in-antlr4"*. This strategy involves recognizing when an ANTLR parser cancels its parsing process. Java doesn't recognize this as a typical exception, so implemented the class suggested by Mouagip which takes that specific cancellation error and turns it into a throw-able exception, which can be caught by Java. Once this is properly implemented, a simply try-catch statement will grab the cancellation error and print out "Not accepted" to the console. If this error is not caught, the parser successfully parsed its input and we print "Accepted" to the console.

The entire process ended once our tests were successful, having a finished parser and thus ending the second step of our compiling process. The next step in program compilation is the creation of the Semantic actions and the Symbol Tables. This step takes as input the Parse Tree, and more specifically its leaves, in order to determine variable (and function) declarations. It's important that we utilize this handy data structure (the parse tree) in the future applications of this parser, rather than embedding application-specific code into our grammar.

3.4 Symbol Table

The creation of the Symbol Table fulfills the third step of the Compiling process, the step where our compiler actually begins to DO something. Up until now, our Compiler has simply been reading in (scanning) the input, turning it into usable tokens, parsing over those tokens to validate the program, and creating a parse tree. At this point, the compiler has truly only read in, and understood the program, and now we must do something with that understanding. However, using this understanding will most likely come in the next step, first we must take one more step in creating the data structures that will help us in code generation.

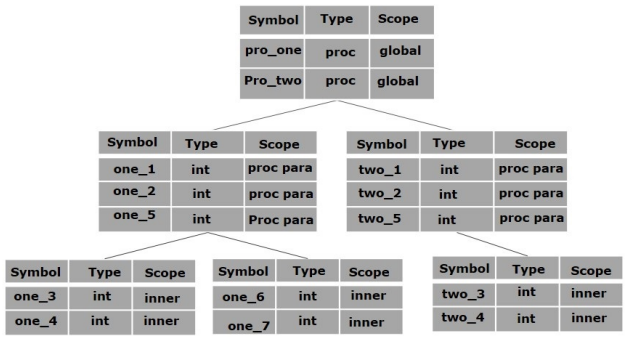

Figure 7: A basic example of a Symbol Table.

Before we jump into how we implemented our Symbol Table, we better describe what a such a structure actually is, and what it allows us to do. The process is not very complicated, nor is understanding what a Symbol Table is.

A Symbol Table is a data structure that contains/stores information about the non-keyword symbols within our program. These symbols are variables, and function names, and many other pieces of code that contain meaning to the computer. These symbols are to be used in further steps of our compiler (when we actually generate code). In fact, that is the purpose of our symbol table, to contain the different levels of scope within our program, at each level holding information the symbols defined/used there. See figure 7 for an example on how Symbol Tables keep track of scope. Each level (from top to bottom) defines a new block of scope within a program, and each scope may have different initialization for variables that it utilizes first.

The next step in understanding the Symbol Table is how we populate the symbol table with the constructs of our program. In order to fill the table, we must tell the program when to create nodes out of the tokens and parsed input. In order to do this, we create Semantic Actions, steps that our compiler takes as the Parser recognizes constructs in the code. Essentially, all the Parser rules that define a construct within code should call a method that creates a node of the construct and stores it within the symbol table.

This would be an incredibly lengthy and complicated process were it not for ANTLR automatically supplying us with the Listener class for our respective (parser) grammar. This Listener class automatically follows along with the Parser as it reads a input program and calls the respective methods as it comes across constructs within code. This may sound familiar, because this utilizes the well known Observer design pattern. You can see this simple design pattern in Figure 8. For example, if the Parser were to interpret a construct as a STRINGLITERAL, then the Listener would automatically call the method to create a StringLiteral node and store it in the current scope's symbol table.

Of course, the actual implementation of these methods was not filled by ANTLR, so we had to create our own methods to create nodes, and populate them into our Symbol Table. We had to create the Symbol Table Data structure, as well as ways to easily navigate the table and it's components. We chose possibly one of the more

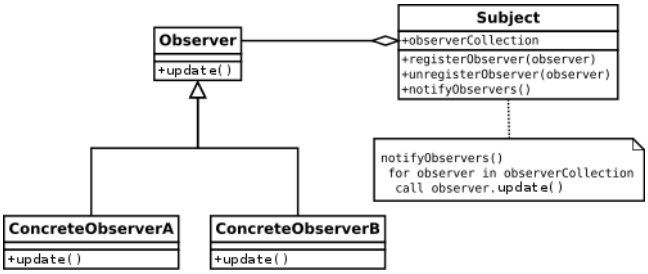

Figure 8: The Observer Pattern.

simplistic ways to do this, but that just means that our program is more easily readable, as well as writable.

The concept of creating and filling symbol tables is not complicated, but implementing it is of course, more-so. It's more easier said than done, so next we will describe exactly how we created our Data Structure and filled it with the semantic records created by our semantic actions. These steps were completed almost exclusively by J. Beckett Sweeney, and his initiative here was very appreciated and impressive.

Our implementation of this step began with the creation of our own Listener class. Because our grammar file is called "Gram.g4", Listener extended the previously mentioned ANTLR generated listener class called "GramBaseListener". After a while of exploring this class and seeing the structure of how our parse tree would be walked, we began picking out which methods we would need to implement. This process took a ton of testing and many print statements to see when each method was called and what data they had access to. It also took a while to figure out the order in which to do things within each method.

Once this was all squared away, we needed to set up our data structure. We elected to take advantage of Java's dynamic ArrayLists. We used an ArrayList of ArrayLists to hold all of our individual symbol tables and their data. This is not the most efficient method of storage, but it is very simple to implement and its dynamic nature makes it more friendly than using just Java Arrays. The outer "shell" of the data structure held each symbol table, and the inner "shell" was a list of each data point within each symbol table. Those data points were made up of our own Node class, which held a Title, Name, Type and Value, all of which were Strings. Title is used only when declaring a new symbol table. For example, the first title called is "GLOBAL". This is the initial symbol table for the program. Any new scope that is entered creates a new symbol table that either takes the name of the scope (eg. "main"), or if it is a While, If or Else block, it declares the new scope as BLOCK X, where X is the number of block scopes entered so far. The other three fields within Node hold the necessary data for corresponding variable declarations. Value is only used for String variables, while String, Float, and Int variables all use Name and Type.

The final steps for our Listener class are checking for duplicate variables and then printing out each symbol table. For the duplicate check, we travel through each scope and compare each variable. If the two compared variables have both the same name AND the same type, the program prints an error with the name of the duplicate variable. If no duplicates are found, each symbol table is printed

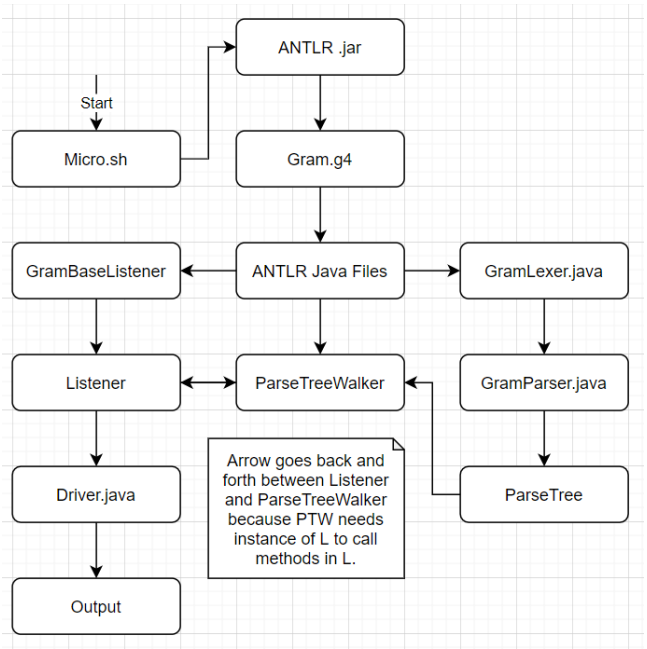

Figure 9: Workflow of our Symbol Table

out in the order that they were declared. Each print begins with the title of the block, such as "Symbol table GLOBAL", followed by each declaration within it, such as "name n type FLOAT" or "name dummy type STRING value "abcde"". This set of prints is compared to our given test cases to make sure we appropriately grab all values. Once all of the tests succeed, we are done!

The last piece of the puzzle is ANTLR's own ParseTreeWalker class. This class takes in a Listener and a Parse Tree as its parameters, then it walks through each element of the tree. This is how we call the methods within our Listener class. ParseTreeWalker calls each method for its given Listener as it runs into them, making for effortless traversal of the tree for us.

Following along with Figure 8, we begin with our Micro.sh file. This workflow is very similar to Figure 5's workflow, so certain aspects have been removed to avoid redundancy. Micro.sh calls the ANTLR .jar file using Gram.g4 to generate a host of ANTLR Java Files. These files use the input that Micro.sh also sent along to generate the Parse Tree for the given input. Listener extends GramBaseListener, then sends itself to the ParseTreeWalker, along with the previously acquired Parse Tree. Using these two things, the ParseTreeWalker calls methods within Listener in order to build the Symbol Table, which once complete, Listener sends to Driver.java. The Driver then prints the table to output and the rest of the test cases are compared from there.

With this step completed, we are ready to move on to the most complicated and difficult process of creating a Compiler; the step of Code Generation. We will need all the information within our Parse Tree, all the information within the symbol tables, and a significant amount of understanding of the theory of program compilation to complete this next task. At the beginning of this next step we hope to implement an Abstract Syntax Tree to aid us in turning our Parse

Tree into a data structure that is easier to work with, and then we move onto finally creating an executable of the Little language.

3.5 Code Generation

Blank Description.

3.6 Full-Fledged Compiler

Blank Description.

4 CONCLUSION AND FUTURE WORK

Blank Description.

5 REFERENCES

Fisher and Parr. *LL(*): The Foundation of the ANTLR Parser Generator.*