Technical Document

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

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

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

ACM Reference Format:
Jessica Jorgenson, William Roberts, and J. Beckett Sweeney. 2020. Technical Document. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ever since Grace Hopper wrote the first compiler in 1952, compiler creation has been a focus of computer scientists across the globe. Compilers are important to modern day computing, as the need to have machine-independent programs has grown with the amount of computers in our world. This project is a chance for our team to experience the complexity, difficulty, and importance of creating a compiler, and offers a great opportunity to learn what happens within computers when you hit the "compile source code" button for your programs.

Compilers are complicated translators. They translate from a high-level language (like Java) into machine code (like Assembly) in a matter the computer understands. They allow for programmers to write code readable to the human eye, and at the same time understandable to a computer. This communication between the readable code and machine code is important, as the readability of a program decides how efficiently the program can be debugged or further extended. Compilers also do remarkably more, including the ability to compile programs that are able to operate on different machines, and even different operating systems. See the Background section for more information.

With the importance of compilers in the modern world, our team felt it was important to get an understanding of how they work, what they do, etc. We were motivated to construct our own compiler, albeit with the tools of the modern day like ANTLR and already-defined high level languages. These tools made the process more intelligible, while offering the same learning experience we would achieve without them.

We began our project with the goal of learning how to construct a compiler, and through the following sections of this technical document, you will see we achieved our goal. We built a compiler using the Java language as our driver, and can use it to compile Little code into Tiny assembly code. In the following sections you will understand how we achieved this, first through a short background about compilers and what they are, then through the different sections on the stages of compiler creation. This paper ends with a conclusion about our process and a small acknowledgement of possible future work.

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/nnnnnnn.nnnnnnn

2 BACKGROUND

What is a compiler? What is its purpose? What are its components?. How do these components work together?

As mentioned above, a compiler is only a complicated translator for high-level languages into a lower-level machine language. Compilers were first created when computers first gained enough memory to allow for higher-level languages to be invented. Before the compiler, all programs were written in nigh-unreadable machine code, and were generally machine-specific. This caused a broad number of problems during that time period which had to deal with running programs on different central processing units, or CPUs. Since the 1950s, computer architectures, along with compilers, became more and more complex - today, we program upon some of the most complicated compilers ever created. Most compilers created are made to compile source code to machine code, though today there are several different kinds of compilers that fulfill different roles. A cross-compiler is an example of a compiler that is explicitly created to compile code for different CPUs and/or operating systems. There are also several examples of decompilers, which are compilers which work the opposite way; from low to high level.

Almost all of the different forms of compilers share the same general components, each of which are detailed in the different following subsections. Below is an outline of the component processes, and how they fit together.

Any typical compiler begins with a Lexer, or a Token Recognizer. This component is perhaps the simplest part of the compiler, and its entire job is to run through every character of the program and convert them into a list of tokens. The job we had was to create a grammar by which a powerful parser generator called ANTLR (further detailed below) could generate the source code for the lexer. At the end of the Scanner process, our compiler would produce a list of tokens, which is used with our Parser.

The next step of the compilation process is the Parser. At this point in the process, the compiler has no understanding of the syntax of the program, and cannot determine if it is indeed a valid program. This is where the list of tokens comes in to play. A parser takes in as input the list of tokens and parses them with a series of context-free grammars, or CFGs. The CFG breaks down the list of tokens into valid program pieces, making a tree filled with different nodes defined by rules of the CFG. This stage in the compilation process is where the program would fail if shown to be invalid. If the parser is successful, it will create a parse tree filled with the different logically-placed tokens from the scanner, which will be used to generate our code.

The third component of our created compiler is the Symbol Table. This stage of compilation works on the output of the two prior stages to create a powerful data structure that will help us in Code Generation. This symbol table is a large collection containing the non-keyword symbols within a Little program. The symbols could be variables, function names, and other pieces of user-defined code the computer will utilize later. The general idea of a symbol table is to store the symbols in different scopes, ensuring that a correct scope symbol is always used at the correct location in the program. This is where the Listener class from ANTLR is used, and our use of the Observer pattern. More information for this pattern can be found in section 3.4. With this step complete, the compiler is ready to generate Tiny machine code.

Code generation is by far the most complicated component, as it is comprised of different inputs and sets of internal controls. Code generation will use every piece of information that has been thus far created, including the parse tree, the symbol table, and a new intermediary data structure, called the Abstract Syntax Tree. During this last step of the compilation process, we will create a tree of Intermediary Representation (IR), Nodes of 3 Address Code (3AC), and use that tree to generate Tiny machine code. At the end of this component, the compiler is complete, and we have new Tiny code for our Little program. We went from Little to Tiny, and this is considered a success.

Ultimately, all of these components work equivalently to an assembly line, with one component after another in the compilation process. Over time, the compiler builds up input and output lines, with the next component digesting what the prior component outputted. The entire process in a high-level is simple; we move on to delve into the individual parts and pieces that make up our compiler.

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 obstacles, we successfully got it functioning.

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

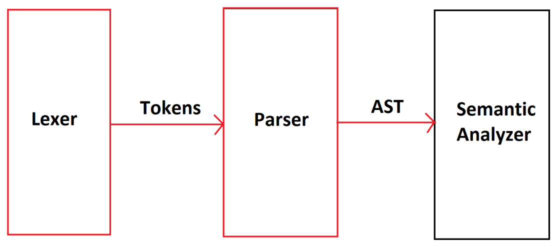


Figure 1: The Simplified process of language recognition.

computers can understand begins with the scanner, making it one of the most important parts, even in its simplicity.

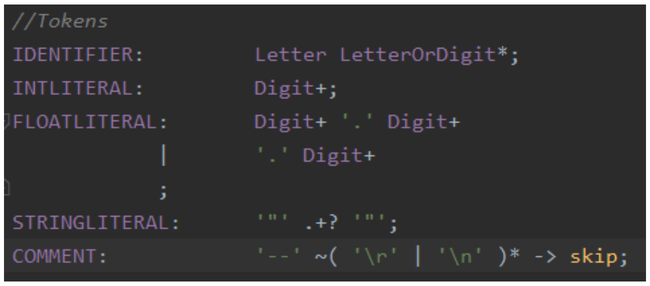
Scanners are token recognizers (or tokenizers, lexers). They sift through the entire file and convert every piece of the language by their characters, such as "(", "+", or "A", into a token. Once converted, the scanner outputs the list of tokens. The process is simple, and utilizes simple regular expressions in logic. There are complications in this is, however. One complication is every possible input must be defined. Another is 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. Developing an understanding 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 ANTLR also can generate parsers. We found it simpler to define the parser promptly, so we can more sufficiently test the first stages of our compiler (see Parser below).

However, a list of tokens is not 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, with most of these challenges resulting 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



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

Figure 2: An Example Regular Expression.

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 "(example)" be included as "(example)", or as "LPAREN example 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 code 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, the parser builds upon those expressions in its own context-free grammar - a grammar that defines the entire parser.

There are many different ways to write a parser, and while each parsing theory might yield the same result, they are written and created in different ways. During our time in our compilers course,

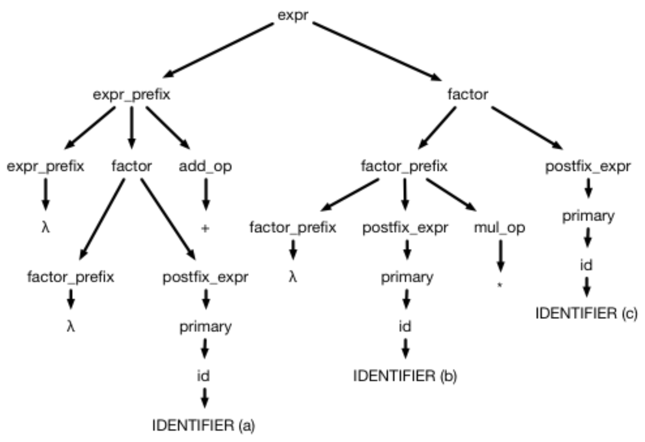


Figure 3: An Example Parse Tree.

we learned there at least two 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 using top-down, where the parser starts at the "root" node, known as the beginning of the program, and building downward by predicting the next production to use, or using bottom-up parsing, starting with its leaves, known as tokens, and building up by reducing the input program to its defined parts. Top-down parsing uses Left-most Derivation (LMD), usually defaulting to Left-to-right, Leftmost Derivation Lookahead(1), or LL(1), which means the parser 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 Left-to-right Rightmost Derivation lookahead(0), or LR(0), which means it starts on the left of the input and goes right like an LL parser; however, it uses the rightmost 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(*), or 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 fail over to backtracking depending on the complexity of the parsing decision," (Fisher et. Parr). This process makes LL(*) 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

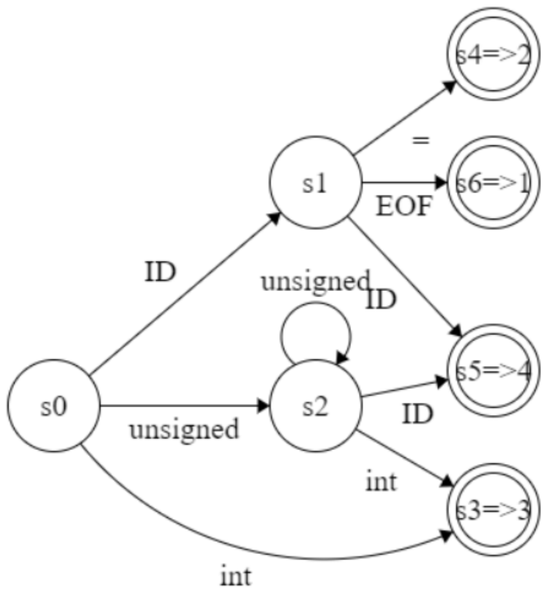


Figure 4: The Resulting DFA. Notation sn => i means "predict the i'th production."

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 DFA in figure 4. 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 immensely 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, as it only knows for sure if it is matching an expression, and that inhibits many of its uses and generality. Prediction is also 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 many difficulties in defining the context-free grammar for the Little language, and the Parser was completed during our step 1, alongside the scanner. The primary challenge we had was developing the Java Driver to interpret "accept" and "failure" and creating the bash scripts to run our tests.

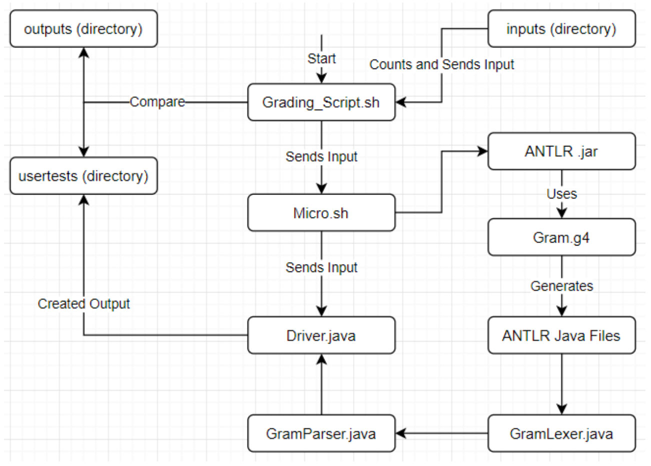
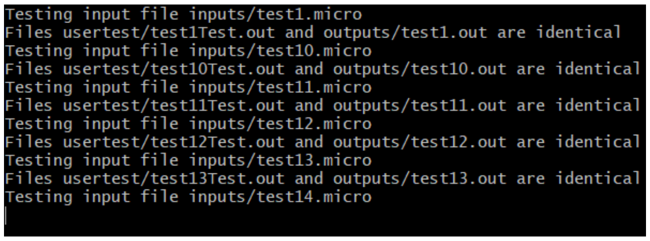


Figure 5: Workflow of our implementation

Figure 3 shows the workflow of our implementation. The workflow begins with the provided Grading_Script.sh, which first initializes an empty "usertests" directory to keep program output. After that, the script 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 to 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. Once everything is generated and compiled, Driver.java is run with the current input file redirected in as input. Driver.java will read the input file, create a Lexer from the file, create a Parser from that Lexer, then run the parser to check if the input program is valid. If it is valid, it outputs "Accepted". If not, it outputs "Not accepted". On its own, Driver.java outputs to the console, but the grading script instead sends its output to a specified output file with the same name as the input file into the "usertests" directory. Once the output file is created, the grading script calls the "diff" function to compare the file to the corresponding file in the "outputs" directory. If these two files are identical, the grading script will return that they are identical. If not, it returns 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 issues. The first was that our Micro.sh script would not run 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 different. The next issue we ran into was our Driver.java file rejecting all input files on the Linux machines. We discovered the way we handled reading the input file was the problem between the different machines. On our personal machines, we read the input file using Scanner.next(), which pulls the next token from a file until there



```
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.

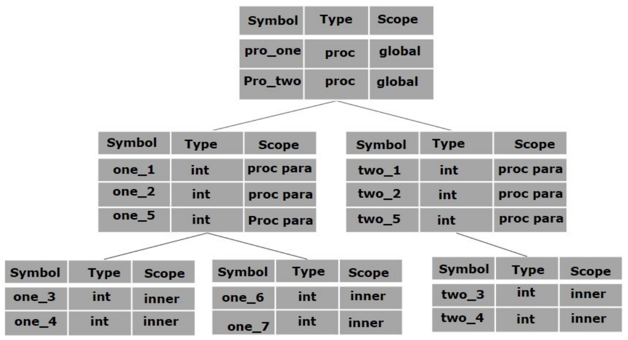
are no more tokens. Along with this, we used several variations of System.lineSeparator(), which worked fine on our machines but did not work in the lab. Some adjustments made a few input files work properly, but any file with a comment in it still did not work. After many more iterations of trial and error, we found that we should use 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 does not recognize this as a typical exception, so implemented the class suggested by Mouagip, which takes that specific cancellation error and turns it into a throwable exception that can be caught by Java. Once this is properly implemented, a simple try-catch statement will grab the cancellation error and return "Not accepted" to the console. If this error is not caught, the parser successfully parsed its input and we return "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 is 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 compile. Up until now, our compiler has simply been reading in, also known as scanning, the input, turning the input 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 and understood the program, and now we must utilize the compiler's interpretation. However, using this interpretation 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.



Symbol	Type	Scope
pro_one	proc	global
Pro_two	proc	global

Symbol	Type	Scope
one_1	int	proc para
one_2	int	proc para
one_5	int	Proc para

Symbol	Type	Scope
two_1	int	proc para
two_2	int	proc para
two_5	int	proc para

Symbol	Type	Scope
one_3	int	inner
one_4	int	inner

Symbol	Type	Scope
one_6	int	inner
one_7	int	inner

Symbol	Type	Scope
two_3	int	inner
two_4	int	inner

Figure 7: A basic example of a Symbol Table.

Before we jump into how we implemented our Symbol Table, we will describe what a such a structure actually is, and what it allows us to do. The process is not 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 and 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.

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 its components. We chose possibly one of the more simplistic

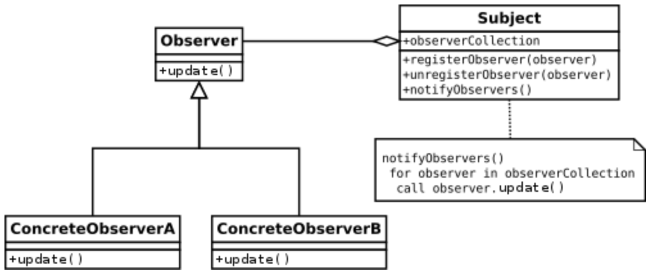


Figure 8: The Observer Pattern.

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 can be. 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 impressive, and greatly appreciated by his group members.

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 choosing which methods we would need to implement. This process took an abundance of testing and many print statements to see when each method was called and what data the methods had access to. Assessing the order in which to perform actions within each method also took time.

Once this was all settled, 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 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 being 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 /todoWHAT DOES THIS IT MEAN? 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 returning 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 returns an error with the name of the duplicate variable. If no duplicates are found, each symbol table is printed out in the order that they were declared. Each print begins with

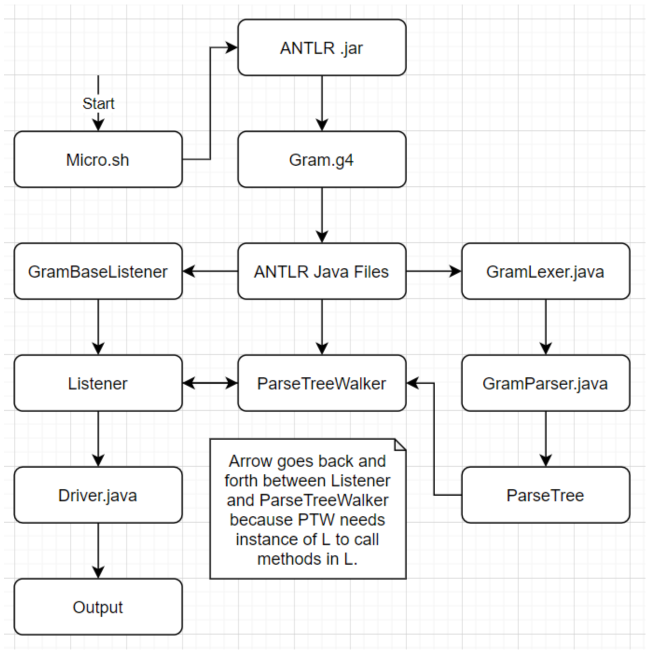


Figure 9: Workflow of our Symbol Table

the title of the block, such as "Symbol table GLOBAL", followed by each declaration within it, such as "name n type FLOAT" or "name example type STRING value 'abcde'". This set of print statements is compared to our given test cases to ensure 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 finds 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 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 sends 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

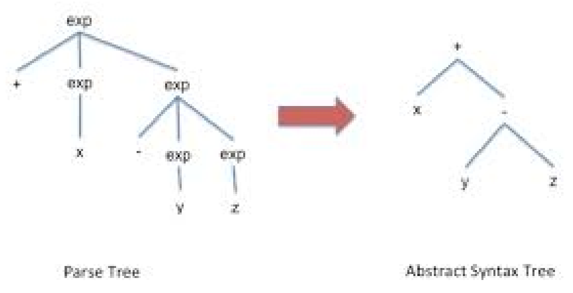


Figure 10: Transformation of Parse Tree to AST

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

3.5 Code Generation

The last step in the compilation process is the most important component. Up until now we have only built the structures and base information required to fulfill code generation; this step is the most complicated.

Code generation is the process of actually converting the information stored within our parse tree into machine code. It takes as input the Parse tree and the Symbol Table and outputs machine code (Tiny code in this case). Even though it is not necessary to explain what code generation is, there is reason to explain the new data structures we use for Intermediary Representations, specifically the Abstract Syntax Tree. With that in mind, code generation was completed in three steps; first, we generated this Abstract Syntax Tree, we then converted the abstract syntax tree into a sequence of Intermediary Representation Nodes using Three Address Code, and finally we traversed the intermediary representation nodes and generated our Tiny code.

The first step is to generate the Abstract Syntax Tree, or AST. It is an important disclaimer that generating an AST is completely optional within Compiler creation, but its purpose is entirely to make the process easier. Recalling the Parse Tree in Figure 3, the data structure contains a lot of information; we need only a fraction of it to generate code. For example, the parse tree usually contains information like ";" and "(" and other means of orders of operations. ASTs greatly reduce the amount of tokens to the essentials we require to generate code (see Figure 10). The AST is greatly simplified over the parse tree, and offers the information we need to generate Tiny assembly. One of the best ways to think about ASTs is that they are the Parse Tree but with all derivation information deleted. This means that they are convenient and meaningful, a great intermediary representation for use in our compiler.

While most of the information on implementation will be discussed further on, it is important to note that we build our AST the same way we build our symbol tables; by using Semantic Actions. Essentially as we walk/parse the parse tree, we are building this additional data structure.

ASTs aren't the only intermediary representation we are going to use in this compiler. While an AST makes the process simpler, we still have to take one extensive step to get from an AST to Tiny code.

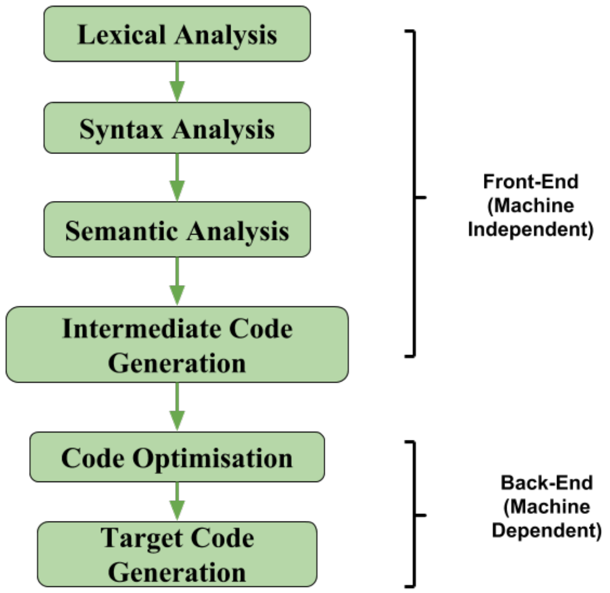

Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimisation
Target Code Generation
Front-End (Machine Independent)
Back-End (Machine Dependent)

Figure 11: The Full Compilation Process

That step is bridged by converting the nodes within our AST into Three Address Code (3AC), another intermediary representation, or IR. 3AC is characterized by its simple structure; each instruction has at most two source operands and one destination operand. It is similar to assembly code, though eliminates the register allocation, making the process of generating the code simpler. Instead of registers, 3AC uses *temporaries*, variables that hold results of computations for later use as we navigate through our program. Below is an example of 3AC converted from the AST in figure 10:

```
|
MULTI y z T1
ADD x T1 T2
|
```

Typically, one would store the value of T2 into a value after the final call within that 3AC; this quick example shows just how easily an AST can be converted into 3AC.

The next and final sub step within Code Generation is converting the 3AC to Tiny assembly code. Tiny is just an assembly instruction set that comes with the Tiny simulator; a program that acts like a simplified machine in order to execute assembly instructions. We used Tiny as it allowed the program to be generally graded by our professor. As was mentioned in the background section of this paper, compilers write machine code that is specific to their type of CPU, unless they are cross-compilers. Tiny is a machine simulator, so it allows any local machine to test compilation upon it; then any computer could do the same as long as they have the simulator. Figure 11 shows how most of our compiler, up to the generation of 3AC, is machine independent. Once we start generating and optimizing code, we are no longer being general to the machine. This step is fairly straightforward, though not simple. Thankfully, in the last step, we generated a long list of IR nodes containing 3AC, which is similar to assembly instructions.

We move onto implementation. As mentioned before, the first step was creating and filling an Abstract Syntax Tree. Due to the order in which the Listener class calls methods, this process involved creating nodes and partially filling them on our way down the tree, then completing their information on the way back up the tree.

Instead of having one massive tree where the root node pointed to each expression, we kept an ArrayList that held each root node of each AST. This meant that each tree could be a simple binary tree. The root node's information was slightly different from all of its children because the root node's behavior would end up different from all others when it came time for the IR.

As the listener travelled into each individual expression, we created a new root node within our ArrayList of AST Nodes. When the listener ran into a new expression, it filled this root node with what type of expression it would be (i.e. assign, write, or read). From this point, the program was considered to be in state a, state w, or state r. This allowed us to create the tree based on its needs, because write and read trees are structured differently from assign trees. Write and read trees always added "id" nodes to their left and "op" nodes to their right. In this case, the "op" nodes only meant that there would be another variable that needs to be written.

In the case of assign nodes, the left child of the root node was the first "id" initialized. After this, empty "op" nodes were added with left-preference until a new "id" was reached. We considered this to be the act of climbing down the tree, which did not fill all necessary information. It wasn't until an "op" node's children were both "id" or "lit" (literal) that its actual operation would be filled. This meant that the operations were filled on the climb back up to the root node.

This process leaves us with an ArrayList of complete ASTs for each expression in the program. The next step was creating the IR for each AST, which would provide the 3AC we need for the final step.

After much consideration, we decided that the easiest way to get our IR code would be to modify our AST Nodes to hold the necessary info. Our generate() method is what retrieves the complete IR code for each expression. This method does a recursive post-order search of the tree, creating IR Nodes as it runs. The method keeps everything in that post order and stores the nodes in a new ArrayList of IR Nodes, which it returns. This process is where we generate a new temporary whenever we might possibly need it. This is nowhere near the most efficient process, but our goal wasn't efficiency, it was completion. We now have an ArrayList of ArrayLists of IR Nodes (list of expressions, each of which is a list of lines of code necessary for that expression) /todoWHAT DO YOU MEAN BY THIS?.

It was at this point we realized we neglected to specify if each expression was for an integer, floating point number, or string. We were able to use a similar technique as before, where we used a post-order traversal to fill each IR Node with their type as well, based on the information we passed it from the Symbol Table step of our project. We now have a type attributed to each line of code, which means we are finally ready for tiny code generation.

Translating IR to Tiny was an interesting task, mostly because our IR used 3AC whereas Tiny only uses 2 addresses. In the end,

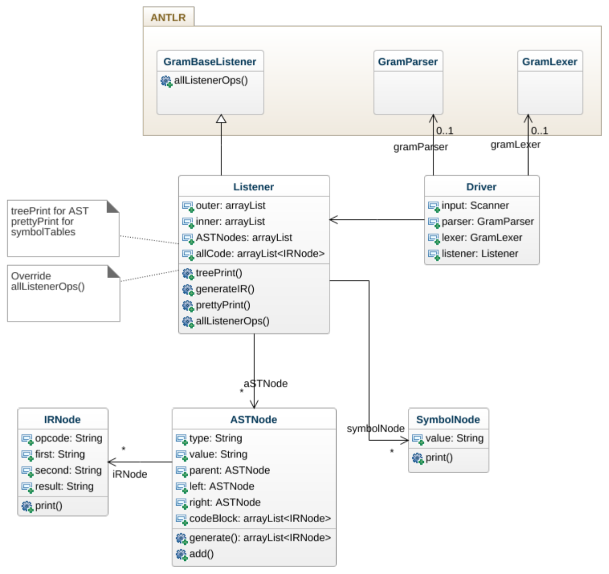


Figure 12: UML Diagram for our Compiler

this only means that any mathematical operation that needed 3 addresses would instead become two separate lines. For example:

c = a + b

would become:

c = a

followed by:

c = c + b

In literal terms, this would instead take the form of:

addi a b c

becoming:

move a c

followed by:

addi b c

3AC's general format is <operation> <operand1> <operand2> <result> where Tiny's 2AC format is <operation> <operand> <result>.

To get our Tiny code, we used a large switch statement with each case containing several if statements. To make the code readable and understandable to humans, our output is grouped by expression, then again by IR code. Each line of IR code, which is commented out using a leading semicolon, is followed by its corresponding Tiny code.

With our variable declarations printed first, then our IR and tiny code, we finally have our complete compiled output.

3.6 Full-Fledged Compiler

The compiler as a whole is composed of many different working pieces, with the bedrock of the program being the ANTLR lexer and parser generated from our gram.g4. All of the driving code was written in the Java programming language, a design choice made by our team simply in favor in using a language we were all familiar with. The Java language also comes with powerful built-in structures, like the Java array list that we could use to cover most of our data structure needs.

The Driver class, where our main method is, contains the references required to every one of our classes, as you can see in the UML diagram in Figure 12. From the Driver class we control and call the methods defined in Listener, and through Listener control the data structures constructed for this compiler like the Symbol Table and the AST.

We also had several different node classes for each different point in the program. Node was used for variable declarations, ASTNode for our abstract syntax tree, and IRNode for our IR code.

Although our final code is unorganized, we regularly cleaned out aspects of the program that were no longer necessary throughout the semester. With the modifications to the project scope, we ended up scrapping a large amount of the code used for the Symbol Table since we did not need to worry about nested scopes, only the main scope.

The only design pattern we explicitly planned was the State Pattern. This was used for large states, such as assign, write, or read statements, as well as small states, such as the type of "id" being read by the Listener.

When given the choice between the Observer pattern versus the Visitor pattern for our Listener, we chose the Observer pattern. This decision was made because the Observer pattern seemed more intuitive to use.

Most of our communication was done through Discord. We regularly checked in on each other and did our planning here, before and after switching to remote learning. As for version control, we used a GitHub repository to hold all of our necessary files.

4 CONCLUSION AND FUTURE WORK

As stated before, this implementation is by no means the most efficient possible implementation. We made no attempt at optimizing our Tiny code, but we agreed as a group that optimization was a low to nonexistent priority from the beginning.

As for future work, we would finish implementing all of the originally intended features, such as multiple scopes, if, for, and while blocks and true optimization.

This project gave great insight into how compilers work and we have a much greater appreciation now for large scale programming languages such as C and Java.

5 REFERENCES

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