Building a Compiler using ANTLR

Tiana Smith, Kincade Pavich

Table of Contents

INTRODUCTION

In the beginning, code was written in machine language. Overtime, tools were built on top of these native frameworks to allow programmers to write code more intuitively, which lead to more efficient implementations and increased productivity. These layers of abstraction lowered the barrier for scientists and programmers to do their work, as much of the heavy lifting was being taken care of by underlying tools. One of these tools is called a compiler. It allows a programmer to write code in a high level language, to be later translated into low level instructions that a machine can understand. With compilers, programmers are able to avoid complex architectures and streamline their work. Compilers also combatted a resource bottleneck, allowing programs to be run on any machine, not just Common Instruction Set Architecture (ISA) supported machines[1,2,3]. Previously, if the ISA was outdated on the local machine, a program would have to be rewritten. Compilers allowed programmers to recompile programs rather than rewrite them. Overall, compilers allowed programmers to be more efficient and focused in their work.

In this project, our goal was to build a simple compiler for a grammar to learn how a compiler works and to practice implementing one on our own. Because in industry compiler generators are often used over compilers built from scratch, we chose to implement our compiler with java using the compiler scanner and parse tree generator tool ANTLR. We show how we were able to build a scanner, parser, symbol table, and perform semantic routines for a grammar of the LITTLE language, a language created for this compilers capstone course. We will describe the challenges we faced in using a tool such as ANTLR, and how we implemented our scanner and parser to work with ANTLR. We will examine how all of the components of a compiler work independently, how we chose to implement them in code, and how we linked all of the parts together to create one compiler that successfully compiles programs of the LITTLE language. We will explain our reasoning behind design choices and share our next steps for optimizing our code and present further projects we would like to tackle in this space.

BACKGROUND

We need to understand how a compiler works and how it can be generated using a tool like ANTLR. The parse tree generator ANTLR is used by Oracle, for their SQL Developer IDE, and by the languages Hive and Pig, which are building blocks for Hadoop, which is used for many large scale data science projects [3]. Understanding how a compiler works is important because it solidifies the big picture understanding of computer science. It strings together computer architecture, programming, programming languages, computer science theory, and many more facets of computer science.

A compiler is composed of four main parts: a scanner, a parser, a symbol table, and semantic routines. First, code written in a specific grammar will be passed as input to the scanner. The scanner will pass this input to the parser, which will generate a syntax tree representing the expressions and statements of the program. A symbol table is created to store variable names, their assignments and their scope. Next, semantic routines are performed to create an intermediate representation (IR) before the code can be translated into a low level language such as machine language or assembly. This low level language is generated by the IR, and the output is a set of executable instructions for the machine to perform. We wrote a shell script called Micro to test our compiler. We then ran our compiler through the TINY simulator, a program that simulates a machine that supports the TINY instruction set to test our compiler.

Let us first look at how a scanner works. A scanner is a tool that reads input from a file and breaks the input into smaller pieces that can be fed as input to the parser. Because this stream of statements, functions, or expressions are broken into characters or smaller pieces, called tokens, a scanner is sometimes referred to as a lexer or tokenizer. To write a scanner, we must use regular expressions to recognize patterns in the input stream of characters. The tool we used, ANTLR, has the ability to generate a scanner using regular expressions. The regular expressions that we wrote to capture our grammar are the token definitions, and additionally, ANTLR uses a character class to allow fragmentation. ANTLR takes these regular expressions that we wrote and can generate code which recognizes when a string in the input matches our regular expression.

After the scanner breaks a stream of input into individual tokens, the data is passed to the parser. The job of the parser is to determine whether or not a string is valid. To make this distinction, we must understand the definition of the language. While regular languages are represented by regular expressions, they are limited in their ability to decipher an entire program in a language. Instead, we must use context free grammars (CFGs), which can handle infinite strings, and recursive languages (which have nesting). In CFGs, a string can be derived by rewriting characters based on a set of rules. To derive a string or token from a language, we must pass through constructs in a language, which are nonterminals, to get to the tokens which are terminals. Nonterminals are rewritten until all characters or tokens are terminals. A useful visualization tool for understanding how a string is produced by a language is a parse tree. The inner nodes of the tree are the nonterminals and the leaf nodes are terminals. A parse tree can then be expanded by top-down or bottom-up parsers, which use either pre-order or post-order traversals to derive the string. Parsers can be divided into separate classes based on how they derive a token. The LL(1) parsers use top down derivation and one symbol lookahead, while LR(1) parsers use bottom up derivation and one symbol lookahead.

So now we have chopped up a stream of characters into individual tokens, and we have built a parse tree made up of those tokens. Next, we must keep track of the variable names and scopes so we can understand what lines of codes need to be executed first inside nested programs. To solve this problem, we will build a symbol table, which is a simple table that keeps track of declarations associated with each scope. A symbol table stores one entry for each variable declaration and its attributes, such as its name and type. Symbol tables may be implemented as a linear list, binary search tree, or hash table. A list of symbol tables is maintained and a stack can be used to keep track of the current symbol table. When a program block allows declarations, a new symbol table is created and is pushed onto the stack as the current symbol table. When a program block is exited, the current symbol table can be popped off the stack. Then semantic actions can retrieve the type and name of the identifier, scan the symbol table to see if it is already there, add it if it is not, and throw an error if it is. Abstract syntax trees (AST) are used to represent the construct of the program. Identifiers and literals can then be referenced by creating a new AST node with a pointer to the table entry. In this step, our goal was to create symbol tables and process variable declarations. This required our parser to get token values such as identifier names and string literals from our scanner. We also added semantic actions to create symbol table entries.

Now that we have passed a grammar into the scanner to break it into tokens, created an abstract syntax tree from the parser, and created a list of symbol tables of declarations which keep track of scope, we need to create an intermediate representation for the code generation step. We can generate the three address code (3AC) directly or after building the AST by a post-order traversal. Code is generated for the child nodes before the parent nodes. Next, we use data objects, which can store information of the current expression, or flags for temporary values such as constant, L-value, or R-value. When a function is called, the frame pointer is set, space should be allocated for local variables, and the callee may save registers the caller is using. Either callee or caller saves can be used for saving registers. Since we have the 3AC, we now need to convert to assembly. We will perform instruction selection, peephole optimizations, local common subexpression elimination, local register allocation, and instruction scheduling.

METHODS AND DISCUSSION

Scanner

To implement the scanner using ANTLR, we only had to write code to handle the scanner's output. In this step, we printed the tokens (type and value) in the standard output, which was later modified to feed the input to the parser, passing tokens as parameters with the help of the ANTLR user manual. Our goal in this step was that our scanner program could open and read a LITTLE source file and print all the valid tokens. To build our scanner, we opted to utilize the tools that ANTLR offers to assist us. We did this for two main reasons. First of all, utilizing ANTLR allowed us to focus our efforts on the most important part of the scanner: the grammar itself. Secondly, although we had a basic understanding of how the creation of a scanner and an entire compiler works, we did not have enough understanding of the topic to build a scanner from scratch without the assistance of ANTLR.

Moving forward, we wrote and tested our scanner using the Linux operating system. After attempting to create our scanner using a Windows operating system, we quickly realized that Windows was not the ideal way to complete this part of the project. Windows throws seemingly random errors that we would have had to found time consuming solutions to, whereas swapping to Linux with the same correct grammar worked flawlessly almost immediately. This taught us that ensuring we use the correct tools (programs, operating systems, etc.) for a given task is incredibly important. Rather than wasting time determining how to use a tool that was not designed for our task, we switched to a tool that worked effectively and efficiently. Our difficulties in building the scanner involved realizing we should switch to Linux, basic syntax and ordering errors in our grammar, and a general lack of understanding in some areas. Each of these issues was solved by referencing online sources, discussing the problems with peers, and obtaining assistance from the lab assistant.

We were getting an error with the EOF at the end of our output files, which did not match the output provided. We decided that changing the do-while loop to a while loop would execute the code only if the condition was true; not before checking if the condition was true. This solved the issue and got rid of the extra EOF in our output files. We were having an issue with the command prompt not showing up after execution of our Micro script. We were having to press ^C to call the command prompt because the way we had written the grammar, it was waiting for more input. We also had issues for a long time trying to figure out why our grammar was not working, but learned that the order mattered in the listing of regular expression for languages. We put the shortest ones first, then the longer ones. After successfully troubleshooting each of these issues, our scanner was fully functional.

Parser

Our goal in this step was to generate a parse tree and then print “Accepted” if the content of each test input was correct according to the grammar, or “Not Accepted” if it was not. The task of creating our parser was almost identical to that of creating our scanner, but went a little more smoothly. Learning from our mistakes in creating the scanner, we started right off the bat using Linux and ANTLR. After developing the rules for our parser, we had to make quite a few syntax changes by adding ‘...’ around rules for specific matching, along with a few other minor changes. Once we were ready to test our parser, we had to update our driver. Our main issue with this was finding a way to compare our output to the sample outputs effectively. Initially, we tried to check if the parse tree was incomplete. After reading further documentation and realizing the method supplied by ANTLR was useless, we were forced to come up with another way. After referencing further documentation, we found a method that allows us to check how many syntax errors occurred in the files we were testing. We utilized this method, and knew that if the method showed any errors then the file was not a valid program. Similar to our scanner, each of our issues were solved by searching through online sources and discussing with peers and the lab assistant.

Symbol Table

We knew it would be important to keep track of each variable’s status - whether they were local (variables declared as part of a function’s parameter list, beginning of a function body, then block, else block) or global (variables declared before any functions). We wanted to construct symbol tables for each scope in our program, so we knew it was important to keep track of nesting. For this step, we output “DECLARATION ERROR <var_name> if there are two declarations with the same name in the same scope. We planned to use a hashtable, stack, tree, or list to implement our symbol tables, and we planned to modify the driver to traverse the tree.

The most challenging aspect of constructing a set of symbol tables for a specific program of the LITTLE language was understanding the ANTLR API and which code was generated by walking the parse tree and recording action events by the listener and which aspects we had to implement ourselves. We spent approximately six hours just figuring out where we should start, working together to figure out how exactly we wanted to implement the symbol table, answering questions such as what data structure we wanted to implement, reading through documentation, and asking the lab assistant for guidance on where to start. We also received some helpful pointers from Stuart Dilts on creating a scope class and a symbol table class. Once we figured out that we just needed the ANTLR file to generate the LITTLE files, we were able to start building some code.

At this point, we still needed to implement the print method and grab a couple of functions and declarations to store in the symbol table. Initially, we thought to implement a hash table, but since we had not implemented that data structure in a while, we decided to implement a LinkedList of ArrayLists of symbol tables. Our next challenge was figuring out how to grab all of our data to print. Due to the nested structure of our program and code, we ended up having to write code which went inside all these structures. A challenge in writing this aspect of the program is that our only debugging tool was print statements. It was difficult to test our code before the entire step was finished without using print statements.

We also had issue with generating an extra block for tests 20 and 7. It took us a while to figure out but eventually we learned that we did not need to increment scope or create another block for “else” statements, because “else” statements are always paired with “if” statements and the “else” part was being called twice (inside the “if” block, and also in its own “else” block). We also had to rewrite the code such that it worked to throw a declaration error when a variable was already defined. We realized that we needed to clone our stack and work with the stack’s clone rather than the original. We were not able to simply make a copy of the original Stack, as they would both reference the same object. Cloning our Stack gave us the functionality to work with the Stack as needed, while leaving the original scoping Stack unaltered.

We were also printing some blocks in reverse. We thought of solving this with a List or an Array, but ultimately decided to implement another Stack and reverse it to keep the symbol tables in the correct order. This allowed us to abandon the LinkedList implementation, and simply print our symbol tables as we popped entries off of the reversed scoping Stack.

Semantic Routines

Our goal in this step was to generate executable code for a simple program written in the LITTLE language, which can be run on the TINY simulator. Every time a grammar rule is recognized, we added semantic routines to be executed by the parser. Completing this step involved creating an intermediate representation from a grammar and then converting it into assembly language readable by the machine. As we worked through input files, we generated relevant IR code for each action being performed. When variables were used, we simply worked our way through the symbol tables to find their declaration and determine their type. From this intermediate representation, we could then generate assembly code.

First, we thought about how we might access our data and how we would like to store that data. We found the ctx.getText() method, which allowed us to grab strings from the listener class as it responded to parse tree walking. We were able to get strings when we entered a read statement, or exited an assignment statement, just to name a few. We wanted to format this data and store it as an IR code. We decided to use a LinkedList to store our IR values. Once we had identified how to get the values and where we wanted to store them, we referenced the IR guide, which showed us the target format of the IR data. Next, we looked through input and output files to understand which type of operations would yield what type of IR structure. Looking at real test cases helped us to understand exactly what operations we needed to perform on our unpolished string to transform it into IR.

We discussed options for how to go about this translation. We immediately thought to use a regular expression library, because we could cutout erroneous text with one line of code. While we were very familiar with the regex library in python, neither of us had used it in java before, so we decided on a simpler method, using String.replaceAll() to clean our string. Next, we had to decide which methods to knock out first. We started with READI, READF, WRITEI, WRITEF, STOREI, STOREF because we knew we just had to distinguish between integer and float values. By printing our ctx.getText() string, we were able to pull apart the variable and type and insert a string, such as "READ" for the READI instruction. We created a string array of all the variables in the read statement, and for each variable, we created a new IR entry, retrieved the type of the current variable, and based on the type, labeled as READS, READI, or READF. Then we added the IR entry to the list of IRs. Since the WRITE and STORE IRs were created similar to this method, we will next talk about tackling the expression statements.

We thought that ADDI, ADDF, SUBI, SUBF, MULTI, MULTF, DIVI, DIVF would be somewhat simple at first, but discovered the difficult in dealing with order of operations. We spent some time discussing the best way to move string values into their corresponding IRs. One of us thought the best way to implement this was a stack, and searching the string for the highest order, '(', then exponents, then "*" then "/" etc, and pushing those values onto the stack in reverse order, so that "-" and "+" operations would be performed last, and would sit at the bottom of the stack. We ended up implementing a list, and shifting values to the left and deleting values as they were performed on, so we ended up with one register at the end with the result of the expression. Often when it came to our implementation choices, we thought one way might be better, but we felt more confident in our ability to write code for the alternative, which is why we chose a list in this case. One of the challenges in this step was trying to figure out how to write code to handle all cases for all input files. For example, in a typical assignment statement we would have a + b, and that was simple enough to store in a register. But when we had a+b/(value - k) we had to rework some of our previous solutions to handle an expression inside of an expression.

Last, we tackled the GT, GE, LT, LE, NE, EQ, LABEL, and JUMP IRs. Once again we jumped back to the sample input and output files to understand how LABEL and JUMP worked and how they were processed in a previous implementation of IRs. We created a global variable to keep track of the label value and spent some time trying to figure out where we should update this value. It was unclear at first if we could increment it after the ENDIF statement, or in the IF statement. Ultimately, we made a choice, tested it, and refined our solution based on our output. In the enter condition function, we just split on the operator and inserted the corresponding label for GT, etc. Then IRs were added to the IR list at the end. We ran our code through the TINY simulator and the final grading script. After taking care of a few last syntax errors and minor mistakes, our code ran flawlessly through the script.

Full Fledged Compiler

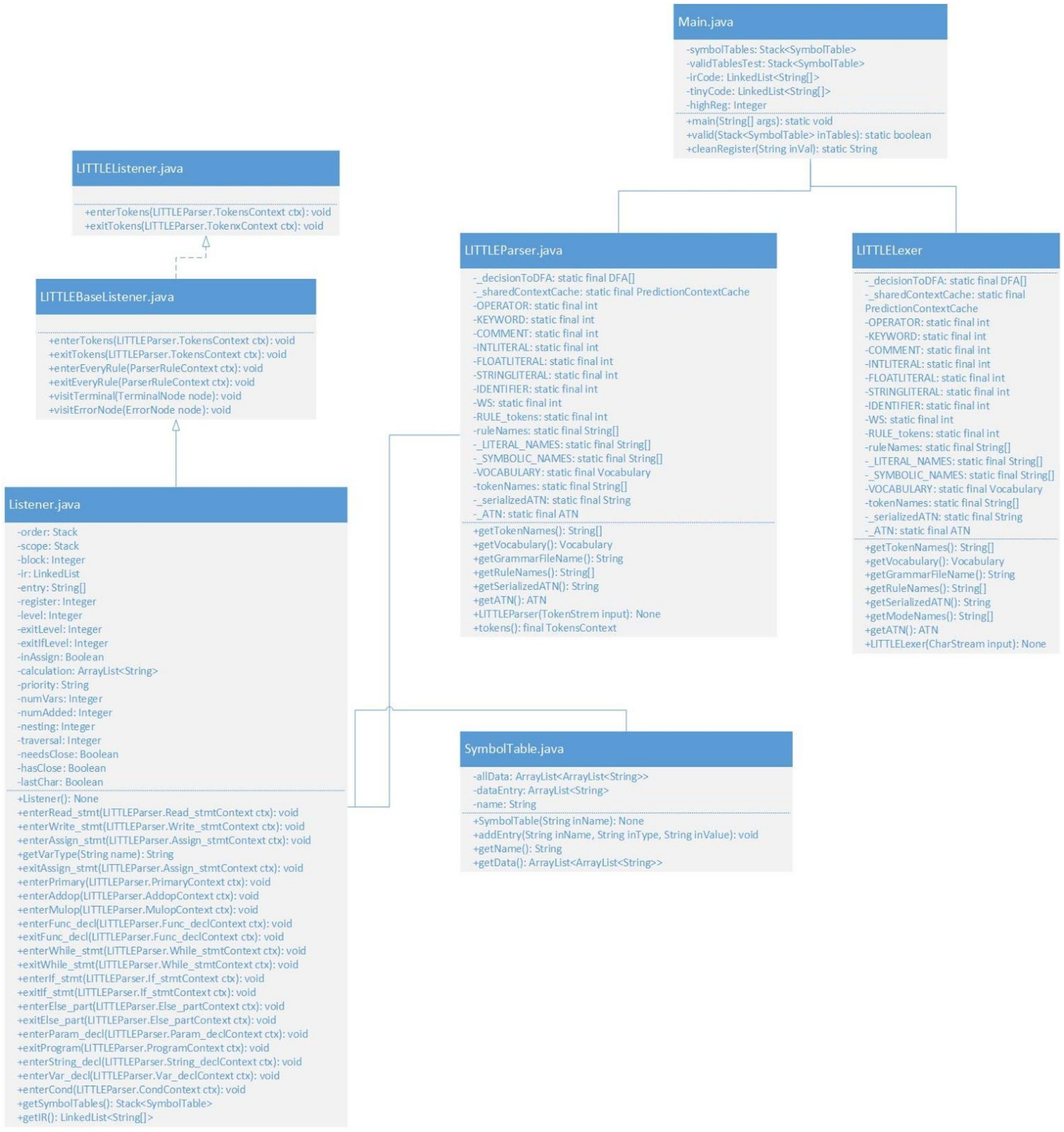
Main.java
-symbolTables: Stack<SymbolTable>
-validTablesTest: Stack<SymbolTable>
-irCode: LinkedList<String[]>
-tinyCode: LinkedList<String[]>
-highReg: Integer
+main(String[] args): static void
+valid(Stack<SymbolTable> inTables): static boolean
+cleanRegister(String inVal): static String
LITTLEListener.java
+enterTokens(LITTLEParser.TokensContext ctx): void
+exitTokens(LITTLEParser.TokenxContext ctx): void
LITTLEBaseListener.java
+enterTokens(LITTLEParser.TokensContext ctx): void
+exitTokens(LITTLEParser.TokensContext ctx): void
+enterEveryRule(ParserRuleContext ctx): void
+exitEveryRule(ParserRuleContext ctx): void
+visitTerminal(TerminalNode node): void
+visitErrorNode(ErrorNode node): void
Listener.java
-order: Stack
-scope: Stack
-block: Integer
-ir: LinkedList
-entry: String[]
-register: Integer
-level: Integer
-exitLevel: Integer
-exitIfLevel: Integer
-inAssign: Boolean
-calculation: ArrayList<String>
-priority: String
-numVars: Integer
-numAdded: Integer
-nesting: Integer
-traversal: Integer
-needsClose: Boolean
-hasClose: Boolean
-lastChar: Boolean
+Listener(): None
+enterRead_stmt(LITTLEParser.Read_stmtContext ctx): void
+enterWrite_stmt(LITTLEParser.Write_stmtContext ctx): void
+enterAssign_stmt(LITTLEParser.Assign_stmtContext ctx): void
+getVarType(String name): String
+exitAssign_stmt(LITTLEParser.Assign_stmtContext ctx): void
+enterPrimary(LITTLEParser.PrimaryContext ctx): void
+enterAddop(LITTLEParser.AddopContext ctx): void
+enterMulop(LITTLEParser.MulopContext ctx): void
+enterFunc_decl(LITTLEParser.Func_declContext ctx): void
+exitFunc_decl(LITTLEParser.Func_declContext ctx): void
+enterWhile_stmt(LITTLEParser.While_stmtContext ctx): void
+exitWhile_stmt(LITTLEParser.While_stmtContext ctx): void
+enterIf_stmt(LITTLEParser.If_stmtContext ctx): void
+exitIf_stmt(LITTLEParser.If_stmtContext ctx): void
+enterElse_part(LITTLEParser.Else_partContext ctx): void
+exitElse_part(LITTLEParser.Else_partContext ctx): void
+enterParam_decl(LITTLEParser.Param_declContext ctx): void
+exitProgram(LITTLEParser.ProgramContext ctx): void
+enterString_decl(LITTLEParser.String_declContext ctx): void
+enterVar_decl(LITTLEParser.Var_declContext ctx): void
+enterCond(LITTLEParser.CondContext ctx): void
+getSymbolTables(): Stack<SymbolTable>
+getIR(): LinkedList<String[]>
LITTLEParser.java
-_decisionToDFA: static final DFA[]
-_sharedContextCache: static final PredictionContextCache
-OPERATOR: static final int
-KEYWORD: static final int
-COMMENT: static final int
-INTLITERAL: static final int
-FLOATLITERAL: static final int
-STRINGLITERAL: static final int
-IDENTIFIER: static final int
-WS: static final int
-RULE_tokens: static final int
-ruleNames: static final String[]
-_LITERAL_NAMES: static final String[]
-_SYMBOLIC_NAMES: static final String[]
-VOCABULARY: static final Vocabulary
-tokenNames: static final String[]
-_serializedATN: static final String
-_ATN: static final ATN
+getTokenNames(): String[]
+getVocabulary(): Vocabulary
+getGrammarFileName(): String
+getRuleNames(): String[]
+getSerializedATN(): String
+getATN(): ATN
+LITTLEParser(TokenStrem input): None
+tokens(): final TokensContext
LITTLELexer
-_decisionToDFA: static final DFA[]
-_sharedContextCache: static final PredictionContextCache
-OPERATOR: static final int
-KEYWORD: static final int
-COMMENT: static final int
-INTLITERAL: static final int
-FLOATLITERAL: static final int
-STRINGLITERAL: static final int
-IDENTIFIER: static final int
-WS: static final int
-RULE_tokens: static final int
-ruleNames: static final String[]
-_LITERAL_NAMES: static final String[]
-_SYMBOLIC_NAMES: static final String[]
-VOCABULARY: static final Vocabulary
-tokenNames: static final String[]
-_serializedATN: static final String
-_ATN: static final ATN
+getTokenNames(): String[]
+getVocabulary(): Vocabulary
+getGrammarFileName(): String
+getRuleNames(): String[]
+getSerializedATN(): String
+getModeNames(): String[]
+getATN(): ATN
+LITTLELexer(CharStream input): None
SymbolTable.java
-allData: ArrayList<ArrayList<String>>
-dataEntry: ArrayList<String>
-name: String
+SymbolTable(String inName): None
+addEntry(String inName, String inType, String inValue): void
+getName(): String
+getData(): ArrayList<ArrayList<String>>

Figure 1. UML diagram of our Compiler implementation.

The best way for us to build our compiler was to follow the waterfall software development model. The project was broken up into four steps by our course instructor. This made tackling each aspect a bit easier. Since we both have never built a project this large before, it was nice to be able to break it up into steps and just focus on one of the four steps at a time. In general, we would read through course materials and figure out what we were trying to do. Then we would look at input and output files to see what we were starting with and what we were trying to generate. Next, we would do some research. For example, in step 3 when we built our symbol tables, we did research to understand how the listener events worked and how the observer design pattern worked. Then we would jump into our code. We never went back and re-iterated something that we had completed before. In step 4 when we were building our IR, we did start implementing some of the IR but paused and switched to some simpler transitions when we got stuck. In this case, we simply tried to figure out what was happening in the simpler examples before tackling harder problems (like the expressions with multiple orders of operations). In the end, we feel the waterfall model was still the best way to implement this project.

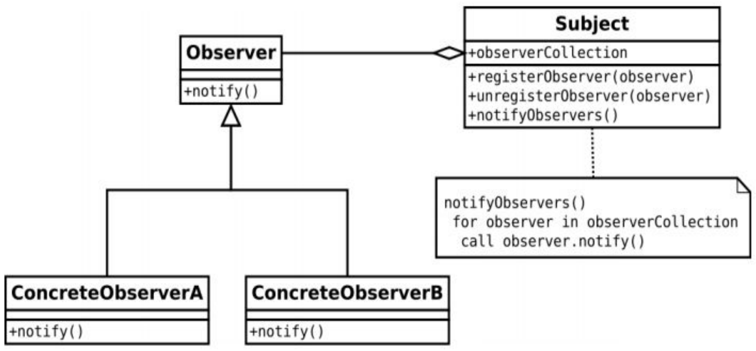

Figure 2. Observer Design Pattern [1].

We implemented the observer design pattern in our compiler using ANTLR Listener. This method allowed us to take actions on parse tree traversals [1]. In this design pattern, listener actions do not return anything, and state or intermediate structures are handled internally. Using this design pattern allowed us to avoid implementing a manual tree traversal, or pattern matching. Instead, our listener class would be automatically notified of events that occurred while walking the parse tree. We first implemented this pattern in step 3 of our compiler, when we were building the symbol tables. We extended the base listener to write our own listener.java class. We continued to use it through the completion of our compiler when we generated intermediate representations (IR) in step 4 of our compiler.

We made many design decisions during the process of building our compiler. One design decision we made in this project occurred as we were deciding how to handle the conversion of complex operations into IR code. We had two logical design options to choose from in this section of the compiler development. Our first option entailed reworking the ANTLR parse tree to allow us to build a Stack of operations to narrow a result down into one register. Our second option involved creating a LinkedList of variables and operators, and then processing the calculations with regards to operator and parentheses priority. Ultimately, we decided to move forward with option two. The first option would have undoubtedly resulted in less code and cleaner code. With this said, it may be actually consumed more time, as we would have had to rework the parse tree to cater to our needs. Furthermore, pushing onto the Stack in the correct order would have been complex, and updating registers as calculations occurred may have resulted in further design issues. Our implementation required lots of redundant code since java does not allow methods within methods and we did not want to pass a large number of parameters into a method, as it would have been more confusing than it was worth. With all of this being said, both methods should actually have a similar runtime. While our method required a significant amount of code, each assignment case generally only used a few of the segments.Overall, it was somewhat frustrating to end up with an implementation that used so much of the same code over and over again. With this being said, when we take into consideration ease of implementation, runtime, and functionality, the method we chose to implement was better given our understanding. With a bit more thorough of an understanding, we may have chosen to utilize a Stack or Tree of operations for this section of development.

We used Google Drive and GitHub for version control.

CONCLUSION AND FUTURE WORK

Building a compiler with the help of the scanner and parser generator ANTLR was a tremendous learning experience and solidified our knowledge of computer science. For each step, (building the scanner, parser, symbol table, and semantic routines) we really had to understand conceptually what was going on and forcing ourselves to learn how to get data from abstract sources was incredibly rewarding. For both of us, this was our first time building a project in Linux, our first time writing a shell script to compile and run a program, and our first time using a generator tool in a project. Because this was a learning experience, and our first time implementing a compiler, we have a long list of things we could improve, as well as long list of things we gained and learned from this project.

In general, our code was inefficient. We often defaulted to using methods we felt comfortable implementing, like lists instead of hash-tables or stacks, because we felt overwhelmed already by trying to understand ANTLR, and how it interfaced with the code we were writing. We were often confused as to which part we were writing, and which part ANTLR was generating for us. Next time, we would like to implement the Symbol tables using a hash table, and handle assignments statements within the IR using a stack or tree. We liked the idea of using the parse tree, but were farther from understanding how to implement that in code than the solution we already had. In the future, we will also have a better understanding of how a generator works. Even if we were to choose a different compiler generator tool, which supported a different language, like C, we would feel more comfortable branching out and making design choices that would make our code more efficient. We would also like to try implementing a compiler in a different language. We are both familiar with python, in which you can write subroutines, and we found it frustrating that we could not define or call a function easily inside another function for this project. Instead, we ended up copy and pasting a lot of code and using flags to allow our program to have a desired functionality.

If we had more time or resources, we would have liked to spend more time learning and understanding ANTLR and spending more time in the design process understanding how different parts of our program would work together. We would also like to try to optimize our compiler for a more complex and perform code optimization, which we did not have the opportunity to do in this course. Another interesting area to explore would be to see if we could learn how to do better testing on our compiler and compilers in general. Often, we were not sure what was happening unless we used print statements, so we would like to explore alternative routes of debugging and testing our code. Overall, this was a great learning experience and we learned a lot about how compilers work and how high level languages can be translated into lower level languages.

REFERENCES

[1] I. Kahanda, K. Smith, Montana State University, Compilers. [Accessed: 30-Apr-2017].
[2] "ANTLR," *ANTLR*. [Online]. Available: http://www.antlr.org/. [Accessed: 30-Apr-2017].
[3] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc, *Crafting a compiler*. Boston: Addison-Wesley, 2010.