LITTLE Compiler

Kevin Hainsworth, Jonathan Johnson, and Josh McCleary

May 4, 2016

1 Introduction

This semester's capstone project for the Compilers class was to develop a compiler to compile code written in LITTLE into assembly code the assembler Tiny can understand and process. LITTLE is a toy programming language designed for use in the Compilers course at Montana State University. LITTLE is based off of the programming language MICRO defined in the textbook *Crafting a Compiler in C*, by Charles N. Fischer and Dee-Ann LeBlanc. The goal of this project was to implement a simple compiler for the LITTLE language. All members of our group were the most comfortable coding in the Java language, therefore we decided to code our compiler in Java, using the Eclipse integrated development environment and using a popular parser generator, ANTLR. Our motivation for starting and completeing this project was driven by the desire to pass our capstone course. For this project, we developed a working scanner and parser, created a symbol table with multiple scopes, and designed semantic actions to create our working LITTLE compiler. The following sections delve into the background of a compiler, how we developed our compiler, as well as the future improvements we can implement.

2 Background

A compiler is a program that translates some higher level language into a lower level language. In most cases, a compiler will translate this high level language

into a form of assembly code that can be executed by computer hardware. Compilers can also optimize this assembly code to run faster and more efficiently. They can also allow for new code to be ported to other machines without needing the program to be rewritten for another machine's architecture. The basic components of a compiler are the scanner, parser, symbol table generation, semantic routine generation, and code generation. Compilers can also include optimizations, but these will not be covered in any great detail in this report.

A scanner performs the first step of the compiler. Its function is to read in an input stream of characters and break it up into a set of tokens. These tokens are defined based on the grammar of the language being compiled. A grammar is a set of rules, productions, and keywords that define a valid string of characters. The scanner is responsible for determining if a string of characters can form valid "words" for this grammar. This set of tokens is then sent to the parser. The parser's job is to determine whether a program's tokens are in a valid order according to the language's grammar and to build the overall structure of a program. Once the parser has done this, the symbol table can be created. The symbol table is a data structure that stores information about non-keyword symbols that appear in programs. These non-keyword symbols are objects such as variable, function or procedure names. The symbol table stores information about program scopes and which of these previously mentioned objects are visible for each scope. With this information, we can determine whether variables and funtions have been defined and used in appropriate places. After the symbol table is created, semantic routines can be designed. Semantic routines are routines that are called as productions are recognized by the parser. These routines are responsible for generating the lower lever language for certain operations that the machine can read. If the compiler includes optimizations, they would be performed after the semantic routines.

3 Methods and Discussion

Each step of the compiler from Step 1: Scanner to Step 4: Tiny Code will be discussed in details which would include the strategy used to build each sections, design implementation, and dificulties and issues that we encountered during the process and solutions that were implemented to solve these.

3.1 Scanner

Since Java is the programming language we chose to program in, our decision on how to implement the scanner for our compiler was to use ANTLR. ANTLR (ANother Tool for Language Recognition) is a lexer/parser tool for Java that generates a scanner and parser from a grammar file. ANTLR is one of the most popular parser generation options for Java. After reading through ANTLR's documentation, and installing ANTLR, we created a grammar file that ANTLR was able to read and generate lexer and parser files. To implement our scanner using ANTLR, we defined the grammar LITTLE using ANTLR's syntax as well as regular expressions to match any tokens that are defined in the programming language LITTLE. This grammar file consists of grammar rules, tokens, and keywords. A grammar rule takes the form of: **ruleName : alternative1 | ... | alternativeN ;** where ruleName is the name of a rule that the grammar can match and identify, and alternative1...N can either be any combination of a set of other rules, tokens or keywords. The following examples display keyword and token matching for the keyword "FUNCTION" and an integer literal token, respectively: **FUNCTION : 'FUNCTION' ;** and **INTLITERAL : [0-9]+ ;** . The second example uses a regular expression to match one or more integers in the range 0-9 to identify the integer literal. When this grammar file is compiled, ANTLR generates the Java files that work as our scanner.

Main Difficulties:

1. Our main difficulty in executing this part of the project was figuring out how to install and use ANTLR with Eclipse. We ran into many issues

trying to get ANTLR to be recognized by our machines and Eclipse. ANTLR's documentation was, while expansive, fairly difficult to read and understand because it is not very accommodating of beginners. We found the documentation to be very good for understanding the general structure of how ANTLR can be used and what functionality it offers, but it was not as beneficial for use as example material. We decided to use the Eclipse ANTLR plugin for our project, which caused some confusion because the documentation sometimes refered to running ANTLR at the command line, rather than in Eclipse. After several days of troubleshooting and reading through documentation, we were successful in installing the ANTLR plugin and could move forward with the project.

3.2 Parser

In order for us to verify that the syntax of a program was being identified as designed, we used a built in function called getNumberSyntaxError() from the parser that ANTLR generated to check if there was an error while it was parsing the input file. This error report returned the number of syntax errors during the parsing routine. This function was called after the parsing routine and the error result was stored in a variable called *error* with a data type of integer. This integer would count each instance of an error in the parse. After getting the error result we wrote a number of conditional statements that allows us to identify if there were errors from the parsed file. If the error variable was equal to zero, the string "Accept" would be written to the output file and if the error variable value was greater than zero, "Not accept" would be written to the output file.

Main Difficulties:

1. We ran into problems with Eclipse MARS 2 updates. After step one, our computers started having multiple errors within the generated files from ANTLR. The next move that we initiated was to update the Eclipse IDE. After the big updates, it showed more errors from the generated files for

all of the computers being used. After additional update attempts, the next logical move was to re-install or update all packages and proper proprietary updates for the ANTLR plug-in for Eclipse, xtext 2.7.3, Java SE 8u73/8u74 Java Development Kits and Java Runtime Environments, and ANTLR v4.5.1 on each of the computers. It should be noted that xtext 2.7.3 was a co-requirement for installing ANTLR in Eclipse. After this point, only one computer seemed to have updated everything successfully and had it working as desired. After managing to get one computer working, we were able to perform the same steps on the other machines to get them functioning once again.

2. Our biggest problem we encountered over the course of the entire project occurred during this implementation of the parser. All test cases we ran on our parser worked except for test cases 1,7,14, and 20 out of a total of 21 test cases. In these four test cases, the parser rejected the input programs when they should have been accepted. Cases 1 and 14 gave a "no viable alternative for END" error, which meant that the parser was expecting something else to show up before the END keyword. We were not sure what it expected to see, but we found it very odd that it all seemed to parse fine until the end, at which point it threw this error. Cases 7 and 20 gave a "mismatched input..." error at the same statement in both cases: RETURN F(n-1)+F(n-2);. We understood this error to mean the parser was expecting to see a right parenthesis directly after the left parenthesis in the first function call: F(n-1), but since it saw n, the parser determined the program was invalid. We struggled for several days trying to resolve these two problems, but could not find a solution. Before falling back to rewriting our grammar, we got in contact with a member from another group who had success with their compiler and also used Java and ANTLR. After some deliberation, we identified the problem in our grammar. We had a grammar rule named *empty* that went to an empty production. This rule was called whenever a statement could potentially end, rather than identify another rule. ANTLR did handle this rule they

way we expected and was causing our strange behavior. We removed that grammar rule, and wherever the empty rule was called, we replaced it with the option to produce an empty production. That change fixed these two errors with the four test cases. We were also calling our whitespace token wherever we needed to ignore whitespace. Since our whitespace token was using the "-¿skip" command in ANTLR, these extra calls were unneccessary, so we removed them. We did the same for our comment token. We give a big thanks to Michael Shihrer for his help in moving our project along.

3.3 Symbol Table

For step three, we implemented the symbol table for the compiler. To do this, we created a number of variable ANTLR objects that allow us to iteratively walk through the parse tree. As we walked through the parse tree, the appropriate symbol tables were printed out to the console. The variable objects for our main file were a parse tree walker object called "walker" and a variable called "extractor" which has an object type of ExtractInterfaceListener. Once we declared those variables, we ran a function called *walk* from the "walker" object and passed in arguments: "extractor" and a parse tree from the parser. This allowed us to incrementally walk through the parse tree and identify the variable names and associated types that were declared within the limits of a specific scope of the function declarations, type declarations, if statements, loop declarations, and the program class declaration.

The class ExtractInterfaceListener was created to extend the LittleBaseListener class from the ANTLR generated files. The LittleBaseListener.java class contains the methods that trigger when the parser enters and exits grammar rules from our LITTLE grammar. ExtractInterfaceListener overrides selected methods to allow us to perform various instructions when a grammar rule is entered/exited. We used a Java HashMap data type to store our information for the symbol tables because look-up in a hash map is algorithmically O(1), or constant, access time and allows us to check for a duplicate variable easily. The

hash map consists of a variable or a parameter name as the key while the value is the type of the variable/parameter. Both the key and value in the hash map are declared as strings. When we enter or exit an associated grammar rule, for instance, exiting a Variable declaration, we check to see if the id of the variable is already in the symbol table (the key). If it is, we mark that there has been an error because LITTLE does not allow multiple variable declaration with the same name. Otherwise, we add the value of the id to the hash map at that key. After the parser finishes, we check if there have been any errors recorded. If there were any errors, we print an error statement. Otherwise, we print out the symbol tables as normal.

In a program, there is usually more than one scope that variables exist in. For instance, some could be declared in the global scope, but others inside an IF statement that the variables in the global scope cannot see. Therefore, we had to account for multiple scopes that each have their own symbol table. To do this we implemented a stack. When we enter a new scope, a hash map is created and pushed onto the stack. Any variables that are created will be placed in the hash map that is on top of the stack because that is the current scope. Once we leave a scope, the top of the stack is popped off. As an example, a new hash map is pushed onto the stack when the parser sees a while statement (the enter while statement method from ExtractInterfaceListener) and is popped off when the while statement is exited (the exit while statement method from ExtractInterfaceListener). This implementation maintains their own declared variables and it forms the basic symbol table collections for later use. With this implementation, we can determine all variables which are visible at each point in the code.

Main Difficulties:

1. The major problem for the extract listener class was figuring out how we could extract the various scope identifiers (which is the identifier for program, class, and function declarations), variable types, and variable values. Through trial and error, we discovered that we could create a parse tree object with a specific type and set the grammar rule object

tree's parent node to the parse tree object. This allows us to iterate though the specific parse tree extracted parent node and check its children for relevant information. The parent node consists of numerous objects that have the values from the grammar rule. We can then iterate through the parent node's children to find the values that we want to extract. Once we extracted the appropriate value we set the children as the parent. We did this so we can delve deeper into other nodes to see what variables we can extract. This is similar to implementing a depth first search algorithm on the parse tree.

2. For the error handling section, the error should only print the very first error encountered from walking through the parse tree. Our initial implementation printed out the last found error rather than the first. The issue was the error variable was being overwritten every single time it encountered another error. Our implemented solution was to change the data structure of the error variable from a String to an ArrayList. This allowed us to insert as many errors that the walker encounters. After the walk through the parse tree completed, we could then pull the very first error encountered. This also helped us see how many errors the parser found and where they were if we were experiencing errors when we should not have.

3.4 Semantic Routines

For step four, we implemented the semantic routines for the compiler. These semantic actions generated the code necessary to execute the program in the Tiny assembly code simulation. Implementing the semantic routines involves overriding more methods from the LittleBaseListener to generate code off of the parser entering or exiting various grammar rules that were being recognized. We needed to generate code was an intermediate representation (IR) between the LITTLE code and Tiny code, which are the assembly instructions that are run on the Tiny simulator. The IR code takes the form of three address code (3AC), similar to other assembly code. As an example, to add two integers together,

the 3AC takes the form of **ADDI OP1 OP2 RESULT**. To generate this code, we created a class called IRNode. This class contained string variables for the operator code (opcode) e.g. ADDI, the first operand, the second operand, and the result. When the parser exited an assign statement, the compiler checked if it was a direct assignment statement (e.g. a := b). If it was, the compiler then checked what type value was going to be assigned. A new IRNode was created with either "STOREI" or "STOREF" assigned to the opcode depending on what type value it was. The node class then stored the value of what was being assigned to an operand and places a numbered register as the result. This IRNode is then added onto an ArrayList to be stored as a sequential list. The register number is then incremented by 1 so we can use as many registers as desired to perform these functions. If it is not a direct assignment, then the right side of the operation gets split into each individual digit or float or operator and gets stored into an array (e.g. 1 is stored by itself, 1.0 gets stored together, + gets stored by itself etc.). Then the compiler checks through the array, looking for specific operators based on order of operations, i.e. parenthesis first, then multiplication/division, then addition/subtraction. When one of these operators are found and processed, the corresponding IR code is generated and added to the list. Similar instructions occur with READ/WRITE expressions and IF/WHILE statements. Due to time constraints, and our IR code not working exactly as designed, we resorted to generating our tiny code by using the parser's syntax tree.

Our tiny code is generated by similar operations as performed for the IR code generation, but generating tiny code instead. To generate code for assignment statements, we implemented an in-order traversal of our syntax tree to allow for any size of right hand expression. Again, we check if the assignment is direct, and print out tiny code accordingly (e.g. var a). Otherwise, we break down the expression in order of operations. When an expression is completed, it gets stored to a register, the strings involved in that expression are removed from the array and that register is stored in their place in the array. For example, if the expression looked like "a := (b+c*d)-e," the first operation would be to resolve

the parenteses. Our program would find instances of parentheses and evaluate that expression to a register value. For this step, our program would look only at "b+c*d." It would then look through this expression for instances of multiplication or division. Our program would then analyze "c*d" and evaluate that to a register. The expression would then be replaced to look like the following: "a := (b+R0)-e." By following order of operations, the program would make the following changes to the above example expression: "a := R1 - e" -¿ "a := R2." The register number was incremented after each register use. Eventually there will be no other operators in the expression left, and all of the tiny code for that expression will have been printed out. Generation of tiny code for read/write expressions and if/while expressions was more simple than assign expressions. If and while statements needed to keep track of their correct labels, creating one for when an if or while is called and creating one when it exits if or else. Once we dealt with these situations, we found the expected tiny code genereated had differences between similar operations. However, after analysis, it was not clear to us what the difference between i and r was on the operations, i.e. what the difference was between divi and divr were for the generated Tiny code.

Main Difficulties:

1. enterAssignstmt – For the assignment expressions in the input files, our code utilized the base listener to walk through the IR and tiny code generation. The most difficult part of the program we faced was how to program the correct arithmetic order of operations in the assignment expressions. The simple expression such as a := digit and a := b * c proved to be easy enough to code. The problem was that when the program was testing the input files for step 4, more complex assignment statements displayed "array out of bounds" errors to the console after the program execution. To fix the complex assignment expressions we went back to the assignment expressions' Abstract Syntax Tree (AST) because, in theory, the AST should display the correct arithmetic operations: parenthesis first, multiplication and division, and then addition and subtraction. Once the program sees

the assignment expression AST, the AST is ran through an inorder tree traversal algorithm. The inorder traversal allowed the program to create a list of the expression by taking the assignment operation, floating point numbers, integers, and variables to their own respective cells in the array list. After inserting each node of the AST tree to the array list, the program can process the assignment expression by iterating through the list. We created mulDiv and addSub functions which allow us to iterate through the array list and generate the appropriate Tiny code for each multiplication, division, addition, and subtraction operation in the expression.

2. Tree traversals – We faced an issue with the original implementation of the in-order traversal function. It seemed to only accounts for some of the ASTs' children and never bothered to look correctly beyond the current node. This problem provided a list structure were some identifiers, assignment expression, floating point numbers, integer values, and parenthesis are inserted in their resprective cells while some entire expressions were saved in a single cell. Saving the expression in a single cell casues the program to view it as one identifier, floating point number, and so on. To solve this dillema, we went back to the in-order traversal function and added fixed this buggy funtionality so that it would properly utilize a depth-first search approach. This method allowed the program to look for the current nodes' children and ensure that the curent node is processed after its left child and before its right child in the tree. After mending the problem, the in-order traversal is now able to traverse the tree and place each child individually to the list.

3. Identifiers' Type – After making the tiny program seperate from the Intermediate Representation (IR) program, some additional functionality was required for the Tiny program to run properly that wasnot working properly in the IR code. One of these functions was the recreation of the symbol table so we could access a variable's type and use the correct Tiny

instruction. The solution for this problem was to copy all the nessesary functions that the IR generation used to know the identifier's type. Unfortunately, while we successfully implemented the symbol table for this step, we were not able to distinguish between when to use instructions such as addi versus addr. Therfore, we left our Tiny code generation as it was with the ability to reference variable type should that be required.

3.5 Full-Fledged Compiler

The full-fledged compiler program project was compiled and merged together from the very start of the project. Each section of the process was built on top of one another. After each program step was finished and debugged, we did our necessary testing and submitted the required files for each step. When we were about to move to the next step of the compiler, we performed minor changes from the current version of the code by commenting the unnecessary code such as print statements and step specific functions like creating files to save the console prints. This process allowed the program to be coded for the next step quite easily since we were building on previously written code for the next step of the project. For example, the ANLTR parser methods are needed for steps 2-4 of the project. By using other classes to override these methods, we were able to see multiple stages of the program work together simultaneously, while maintaining visual code separation. As an example, the base listener used for step 3 symbol table was used to walk through the abstract syntax tree to identify the semantic routines. The same base identifier was also used to walk through the abstract syntax tree for step 4 semantic routines.

Throughout the program design, the "builder" pattern is used to help the program organize the complex functionality and organization of the various compiler phases. The builder pattern was used primarily in the generation of the step three symbol table and the step four semantic actions components of the compiler. Both used classes from the base listener generated by the ANTLR parser generator. This design allowed the program to share the same resources while maintaining its own functionality for each step. In this case, the step 3

symbol table can walk through the syntax tree and generate the necessary print statements while the step 4 semantic action can see and walk the same syntax tree while deciphering the required node of the tree for further analysis and computation.

Our compiler implementation also consisted of an iterator and observer behavioral design pattern. We needed to traverse through sections of data constantly, whether it was traversing through parse trees, through lists and arrays, or through strings. Iterating through our data and objects was a key part to implementing our compiler. The observer design pattern was used to identify when an error was found by the parser. This observer method observed the parser moving through the syntax tree and notified the main class when an error was found. This was how we knew when to print out an error message or when a program was valid or invalid.

During the design process, the team decided to build each project part on top of the last while maintaining the integrity and functionality of the previously written code. Earlier in the semester, the program had different versions or iterations we saved on our computers so that if there was any major setback in the process, the team can fall back to the previous code and build again from that point. This form of code production gives us multiple instances of the functioning code but a potential problem could arrise if the team does not collaborate and ensure they are working on the same copy of code. For example, if two members are working on fixing a bug in the same block of code, they may come up with different solutions and two issues will have occurred. First, the same work has just been performed twice, wasting time. Secondly, the members now have to work on merging their code and ensuring nothing else breaks in the process. This also wastes time and energy that could be better used working on other parts of the project. The change that we implemented was as follows: If one team member works on the program and has made functioning changes, he would send it to the rest of the team so that each team has the latest version of the program to work with. Our UML diagram of our compiler displays the combined parts and is shown in Figure 1.

The tool that the team used for version control was electronic mail. Electronic mail was used extensively for sending and distributing the latest code version. The team had discussed using a code repository program, GitHub, which was primarily in consideration during step 4 of the program phase handling semantic routines, but this was not formally executed because most of the team members were not comfortable using GitHub due to some learning curve required to use the tool.

4 Conclusion and Future Work

With the completion of our compiler comes future plans to improve upon it. With more time and resources, we would rework our intermediate representation code and tiny code implementations. We want to have our tiny code generated from IR code, rather than straight from the source code. We would like to optimize our compiler in several ways. Currently, we have few code optimizations and no run-time memory management in our compiler. We would review our grammar to see if any of the rules or tokens could be modified to improve the efficiency of our parser, especially since it is using ANTLR. We could see if there were any optimizations we could make that ANTLR could take advantage of. We would like to implement optimizations to use fewer registers in the Tiny code to save on memory. Given more time and resources, we would look through our code looking for places where our implementation is slow, or uses up a significant portion of memory. We would take another look at those implementations and try to devise different ways that would hopefully be more efficient. Currently, the vast majority of our global variables and methods are publicly available. This is a major security concern that was not addressed over the course of this project. To eliminate this issue, we would search through our code and change several of these variables to be private. Any variables that would need to be accessed or changed would then receive methods that return or set the variable. Security was not a primary concern during the implementation of the compiler, but in the future we would want to ensure that nobody

could access any of the compiler source data without our intention. Though the compiler is working as intended, it is far from perfect and we would like to improve these imperfections in future work on this project.

5 References

1. Fischer, C. N., LeBlanc. RJ Jr. (1991). *Crafting a Compiler with C.*
2. Parr, Terence. *The Definitive ANTLR4 Reference.* 2nd ed. N.p.: Pragmatic Bookshelf, n.d. Print
3. Dan Pilone, and Neil Pitman. *UML 2.0 in a Nutshell.* Sebastopol, CA: O'Reilly, 2005. Print.

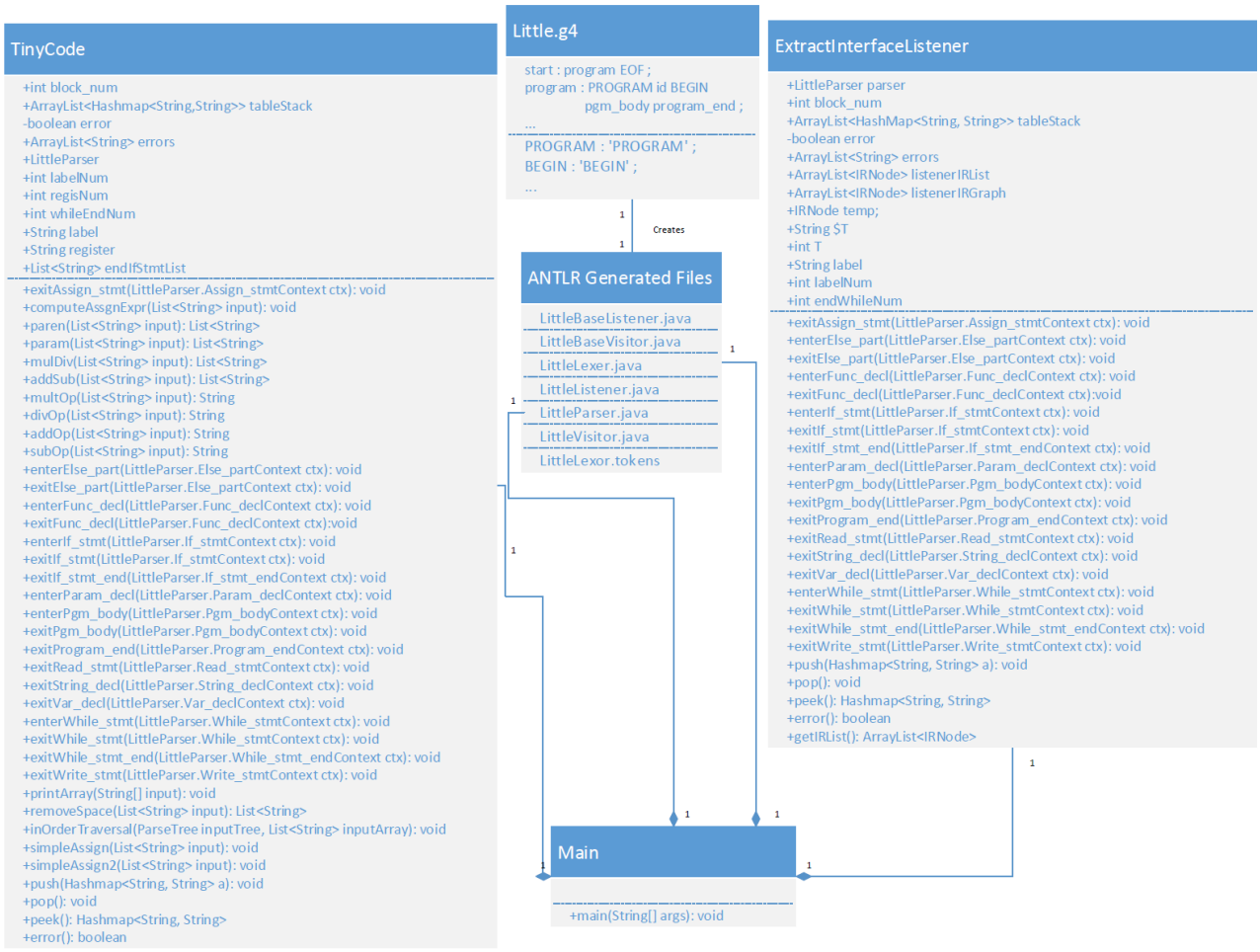

Figure 1: This is the UML class diagram for this compiler.