CSCI468 - Compilers

Trent Baker, Ethan Malo, Marie Morin, Mike Pollard

March 2, 2019

1 Introduction

This project is about creating a small compiler to convert code from the LITTLE programming language into instructions that a computer can execute. It is the focus of the Montana State University Computer Science capstone course, and serves to give us a better understanding of the compilers we use daily. We will implement each step of the compilation pipeline and describe the concepts behind each step.

2 Background

A compiler is a program that converts a program written in one programming language to another language. It converts the instructions the first program provides to instructions in the second language. Machine code isn't human-readable, but instructions must be in machine code for computers to execute the instructions. The function of a compiler is to take human-readable code written in a high-level language, and translate it into machine language at the lowest level to be interpreted by the computer.

The main parts of a compiler are as follows:

- Scanner - Converts program instructions into a string of tokens.
- Parser - Converts string of tokens into an abstract syntax tree or a parse tree.
- Optimizer - Optimize translated code, making it more efficient.

An example of a compiler is GCC for C, C++, Objective-C, Java, and many more languages. GCC takes C code and converts it to assembly language, and then from assembly language the code is assembled to machine language.

3 Methods and Discussion

3.a ANTLR Setup

For the development of the LITTLE compiler, we are employing the use of the tool, ANTLR v4, as a parser generator for reading, processing, executing and translating structured text or binary files. This tool was chosen as it is considered a powerful industry standard with the ability to integrate seamlessly with the Java programming language and consequently, our LITTLE compiler.

In order to setup ANTLR v4 for development, we had to install the Java 1.6 or higher dependency and then download, install, and configure ANTLR appropriately with the following commands.

```
$ cd /usr/local/lib
$ curl -O https://www.antlr.org/download/antlr-4.7.2-complete.jar
```

Then we added `antlr-4.7.2-complete.jar` to our `CLASSPATH` by inserting the following line into our `.bash_profile` or `.bashrc` file depending on configuration

```
export CLASSPATH=".:/usr/local/lib/antlr-4.7.2-complete.jar:\$CLASSPATH"
```

Lastly, we created aliases in our `.bash_profile` or `.bashrc` file for both the ANTLR Tool and TestRig with the following insertions.

```
$ alias antlr4='java -Xmx500M -cp"/usr/local/lib/antlr-4.7.2-complete.jar:\$CLASSPATH"
org.antlr.v4.Tool'
$ alias grun='java -Xmx500M -cp "/usr/local/lib/antlr-4.7.2-complete.jar:\$CLASSPATH"
org.antlr.v4.gui.TestRig'
```

We decided to use Java for this project, and therefore did not have to install any runtime tools for other languages.

3.b Scanner

The purpose of a scanner in this project is to convert a given file into machine recognizable tokens. The first step is to reading in a stream of characters from a given source, often a file of the appropriate type. Then collections of characters are grouped and matched to token types from a given grammar specification. It then returns each token in a stream, and gives them to the parser to continue compiling. If invalid tokens are found, this process will instead result in an error and the program will not compile.

Scanning is the first step in the compilation process and therefore takes human-readable source code as input. This input may contain faults, and depending on the type of fault, the scanner may or may not catch them. For this step, we assume that our input is valid and does not contain faults that a scanner should catch such as invalid or missing symbols.

In order to create the scanner for the LITTLE compiler, we first had to look at the specification for the language provided for us in `grammar.txt`. This helped us decipher what we would need to include in the `.g4` file for the lexer to recognize the required tokens. The symbolic names for the components we had to include were:

- `KEYWORD`
- `IDENTIFIER`
- `INTLITERAL`
- `FLOATLITERAL`
- `STRINGLITERAL`
- `COMMENT`
- `OPERATOR`
- `WS`

Each of these was represented as a regular expression that would capture the specific type of token that is present within the LITTLE programming language.

Specific regular expressions were necessary to match the expected assignment of token types. The regular expression for `STRINGLITERAL` needed to implement a lazy matching strategy with `?` so as to only assign the string type to characters until the next `"` was reached. Similarly, the `COMMENT` regular expression implemented the lazy matching strategy to prevent it from interpreting most or all of the proceeding inputs as one large comment.

As the lexer functions with a top down, or first to last, approach to identifying token types we had to consider the order in which we defined the lexer rules. First we search for tokens of type `KEYWORD` which needs to take precedent over the `STRINGLITERAL` type specifically, otherwise we would not be able to properly identify `KEYWORD` tokens at all. In a similar way we determined that the `FLOATLITERAL` token type should precede `INTLITERAL`, lest the initial whole number value of every `FLOATLITERAL` be interpreted to be an `INTLITERAL` with a separate trailing `FLOATLITERAL`. See figure 1

input	`INTLITERAL` then `FLOATLITERAL`	`FLOATLITERAL` then `INTLITERAL`
1.23	1, 0.23	1.23
4.0	4, .0	4.0

Figure 1: Scanning order

Additionally, the `COMMENT` identifier was placed after the `STRINGLITERAL` so we would not run into issues with strings containing the comment indicator "–" as this would cause the scanner to run into potential errors with string handling as there could then be single unhandled quotes through the code.

The next step was to generate all of the java files for the grammar that we wrote called `little_grammar.g4`. Since ANTLR is being used to help with the lexing process, we executed the command `antlr4 little_grammar.g4`. This created a host of different `.java` files and `.token` files in the working directory. From here, we were able to

```
theNomad:step_1 trent$ ./Micro inputs/fibonacci.micro
Token Type: KEYWORD
Value: PROGRAM
Token Type: IDENTIFIER
Value: fibonacci
Token Type: KEYWORD
Value: BEGIN
Token Type: KEYWORD
Value: STRING
Token Type: IDENTIFIER
Value: input
Token Type: OPERATOR
Value: :=
Token Type: STRINGLITERAL
Value: "Please input an integer number: "
```

Figure 2: Micro script with input file (output truncated)

```
theNomad:step_1 trent$ ./Micro
```

```
theNomad:step_1 trent$ ./Micro
232a233
> intentional difference for demonstration
\ No newline at end of file
```

Figure 3: Micro script without input file (full output)

compile all of the java files with `javac little_grammar*.java` which compiles the generated Java source code into the necessary `.class` files. This enabled us to start writing the driver program to print out the tokens correctly.

In regards to the printing program, the only import required for the lexer to work was `org.antlr.v4.runtime.*` Then, in the Java class we created called `printTokens`, we run the entire program solely in the main function. Firstly, we created a new lexer object from the lexer that we generated using ANTLR called `little_grammarLexer`. In the same line we also use the `CharStreams` ANTLR class to read in the input file from the first command line argument. Finally, we loop through all of the tokens to print them out using both the `Token` and `Vocabulary` objects.

To compile the program, we use `javac printTokens.java` and then we can run our program on some input file using `java printTokens \$1` where `$1` is the first command line argument representing the path to the input file.

To expedite the entire process of compiling and running everything in the correct order and with the correct inputs, we created a bash script called `Micro`. We can then run the script with `inputs/sqrt.micro` by executing `./Micro inputs/sqrt.micro`. This script correctly compiles and runs the program printing the output to `STDOUT` as well as removing any of the generated files afterwards. Our build process can be summarized into the following steps:

1. Use ANTLR to generate a lexer and other relevant files
2. Compile the generated source code
3. Compile our driver program
4. Run our driver on some input file
5. Delete generated files

The `Micro` script has two modes of operation. If executed as described above with an input file that contains LITTLE source code, the token type and value is printed for each detected token as in figure 2 Alternatively, if no file is given as input, `Micro` will run our scanner on each of the following inputs and only print out the differences between our output and the expected output using `diff -b` as shown in figure 3

3.c Parser

The purpose of the parser in the project is to convert a stream of tokens into a parse tree. The parse tree is built using rules provided by a given grammar, and if the parse tree is accepted by that grammar then the program is syntactically valid. To read a stream of tokens and determine if they are accepted the parser must be able predict the productions that could be used to generate the tokens from non-terminals in the grammar, as well as productions to produce these non-terminals. This process is then repeated until there are no valid predictions remaining, which will be the top of the parse tree. If the program is accepted by the grammar, then the tree will begin with the `program` non-terminal. Any other end to the tree characterizes an invalid program.

The parser is the second major component of our compiler and it is designed to pick up where the scanner left off. It is built using the same grammar as the lexer, the LITTLE language grammar, specified in `little_grammar.g4`. It is a left-to-right right-most derivation parser with zero look-ahead, or LR(0) grammar. This indicates it's bottom to top nature, which is necessary to efficiently determine if the input stream constitutes a program accepted by the grammar.

The right most derivation also indicates the capability of the parser to represent recursion. The parser can handle recursion, but only when the recursive call is the left most term in the production. This is due to the fact that when analyzing potential productions to use, it will first check if the recursive call has ended, and only once it has determined that it has not, will it check the next depth. If this were not the case, that is if the parser were to check for another level of recursion, it would result in an infinite depth check which would cause the parser to fail even if the program is otherwise functional and acceptable by the grammar.

Having a zero look-ahead indicates that once a specific non-terminal symbol has been identified, the compiler is able to identify the production used with that symbol simply by looking at the next symbol to process. In other words, once the first symbol from a production has been read, the parser will know the unique production used. Even for non-terminals with complex productions with many specific symbols, such as the `func_decl` production, or non-terminals with numerous different productions, like the `base_stmt` productions, after the parser reads in the first symbol the entire production used will be unique(see Figure 4).

```
/* Function Declarations */
func_declarations : func_decl func_declarations |;
func_decl : 'FUNCTION' any_type id '('param_decl_list')' 'BEGIN' func_body 'END';
func_body : decl stmt_list;

/* Statement List */
stmt_list : stmt stmt_list |;
stmt : base_stmt | if_stmt | while_stmt;
base_stmt : assign_stmt | read_stmt | write_stmt | return_stmt;
```

Figure 4: A sample of productions in the LITTLE grammar

The LITTLE grammar consists of 49 non-terminal symbols, and a total of 110 productions including those used in the lexer. These symbols and productions make up the entire grammar necessary to represent the LITTLE programming language. To better differentiate between productions, they are also broken up into the following categories:

- `Program`: The production rules used to define the program being run.
- `Global String Declaration`: Productions to declare string variables
- `Variable Declaration`: Production declarations of floats, integers, and consecutive variables.
- `Function Parameter List`: Productions for lists of parameters used in a function.
- `Function Declarations`: Productions of the main function body and operations within the function.
- `Statement List`: Production rules used to parse a series of statements, including the base statements, if, and while statements.
- `Basic Statements`: The production rules used to parse assignments, read, write, and return statements within the program.
- `Expressions`: The productions to define general expressions, such as addition or multiplication.
- `Complex Statements/Conditions`: If statements and conditional expression productions.
- `While Statement`: A specific declaration for while loops.
- `Lexer Rules`: Productions used in specific terminals.

ANTLR also includes tools for generating `.dot` files that represent the grammar in the form of augmented transition networks (ATN), which are a type of NFA. These files can then be converted into images with a program also named `dot` to see how the compiler would parse various symbols in a graphical form. Figure 5 shows a rule from the grammar called `primary`. You can see that this network makes heavy use of non-determinism and is labeled with arbitrary numbers, which can make it more difficult to link each of the ATNs that were generated. These diagrams are most useful when confirming that the rules written in the grammar are producing correctly.

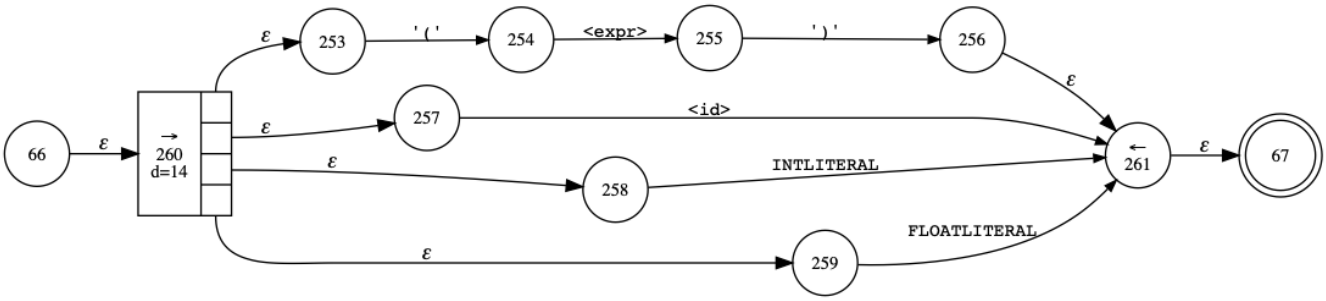

Figure 5: Expressions: primary

These symbols and productions are also what allows for the parser to perform error checking.

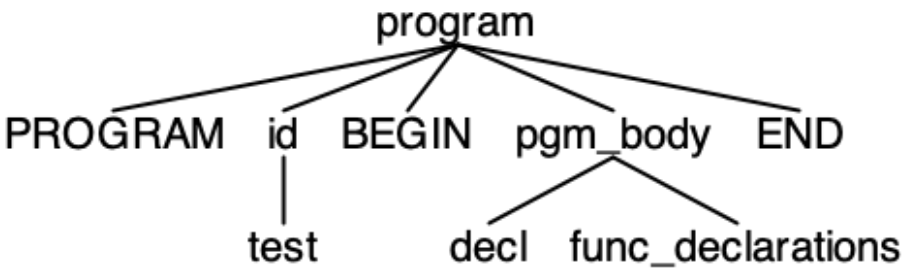

Figure 6: Correct and complete parse tree for test1.micro

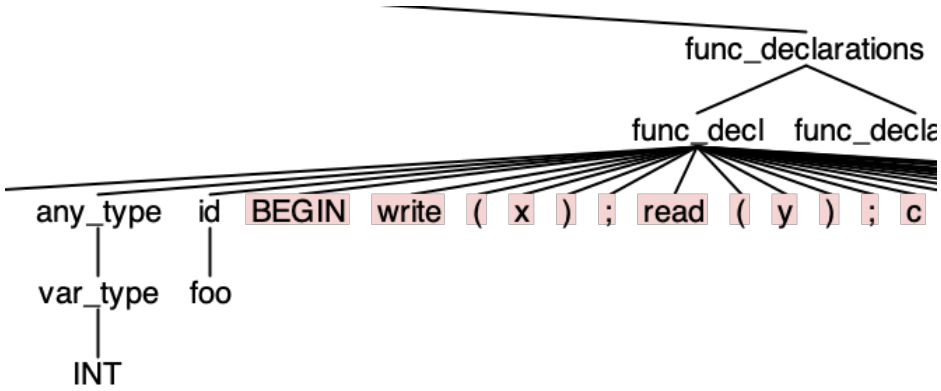

Figure 7: Excerpt from parse tree with parsing errors for test4.micro

The Micro script for part two checks for the `antlr-4.7.2-complete.jar` in the current working directory and downloads it using `curl` if it is not present. Then, the script executes the jar file on `little_grammar.g4` to generate the antlr java files for use in parsing and lexing. All of the java files are then compiled using the `javac` command and the program can now be run using the grading script to determine if there are any parse issues within a given LITTLE input file. At the end of the script, the generated files are deleted and the full output is presented.

3.d Symbol Table

(min. 2 pages)

3.e Code Generation

(min. 2 pages)

3.f Full-fledged compiler

4 Conclusion and Future Work