CSCI468 - Compilers

Trent Baker, Ethan Malo, Marie Morin, Mike Pollard

February 10, 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

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

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

(min.2 pages)

3.d   Symbol Table

(min. 2 pages)

3.e   Code Generation

(min. 2 pages)

3.f   Full-fledged compiler

4   Conclusion and Future Work