Compiler's Project Report

Trevor Gahl & Skylar Tamke

April 30, 2019

1 Introduction and Background

This project pertains to the creation of a compiler geared towards the Tiny language. The methods and discussion portion of this document will cover the process of creating said compiler throughout multiple stages. First covered is implementing the required toolchains. While there are multiple different toolchains used to create compilers, this document focuses on using one in particular, ANTLR. After setting up ANTLR, we move on to the creation of the scanner, used to read in a micro file and use regular expressions to break the text down into the relevant literals. From there, the project is expanded to create a parser. This stage is used to read in the tokens provided from the scanner system and create a parse tree, describing the program and determining program validity. Once the parser was created, further expansion was done to create the symbol table. A symbol table breaks down the provided code and creates a structure that is used to display how a compiler can look at a code segment and prepare it for code generation. Which leads into the last expansion step performed, code generation. Code generation takes all of the previous components and creates the machine level assembly code required for the program to run on the targeted hardware.

All of these stages together create what is commonly known as a compiler. A compiler is simply a program that breaks down a programming language into machine code. A very common example of this is gcc, which is a C compiler. GCC takes a program written in C and breaks it down into a format that is understandable to a variety of different computer architectures. As outlined in the previous paragraph, these compilers commonly consist of four main components; the scanner, the parser, the symbol table, and code generation. Each of these components is built off of the previous allowing for the final code generation to be performed. The process that our group took to actualize the required compiler can be explored throughout the rest of this paper, culminating in the ability to compile Tiny code to the corresponding assembly instruction set.

2 Methods and Discussion

2.1 ANTLR Setup

Before building the components of a compiler it's necessary to configure the tools that will be used. There are two powerful tools that can be used to help build a compiler, Flex and ANTLR. Flex is a commonly used UNIX scanner that is used to generate the tokens for the parser. However, ANTLR takes it a step farther and is capable of producing the parser in addition to the scanner. This is done by converting regular expressions into code that will recognize when the desired regular expressions are matched by a string.

The process of setting up ANTLR was straightforward and painless. Since ANTLR was developed as a java tool it requires java to be installed on the host machine. After java is installed, the ANTLR jar file is downloaded and added to the machine's classpath. From there it's possible to use ANTLR but the creation of a few batch files makes it less painful. Two batch files are created, one to run ANTLR and the other to run TestRig. After creating the batch files they are added to the system path. A quick test case of the Hello grammar file provided by ANTLR demonstrates the system is configured and ready for use.

2.2 Scanner

After the ANTLR toolchain is functioning, creation of the compiler can begin. The first stage to any compiler is the scanner, which takes the input stream and breaks it into a set of tokens. These tokens can consist of such things as identifiers, reserved words, literals, operators, etc.. In order to begin crafting the scanner, the grammar of the language must first be known. Without the grammar definition the keywords and other token values will be unknown, making it impossible to create a scanner. In the case of this project the grammar was given out with the project requirements in the grammar.txt file. More information on the grammar of this language can be found in the required text book "Crafting a Compiler with C".

The grammar text file provided includes the entirety of the language grammar, much of which is unneeded for the scanner portion of the compiler. Required parts of the grammar text file are the literals (INTLITERAL, FLOATLITERAL, STRINGLITERAL and COMMENT), the keywords, and the operators. Knowing the format of these token definitions allowed us to define the operators and keywords in our ANTLR file, but more importantly it allowed us to craft the regular expressions required to capture the defined literals. It's also known that whitespace is ignored, this was placed and defined at the end of our ANTLR grammar file.

The defined literals (as found in the supplied grammar.txt) are as follows:

- INTLITERAL: integer number
  - ex) 0, 123, 678
- FLOATLITERAL: Floating point number available in two different formats (yy.xx or .xx)
  - ex) 3.14, .14
- STRINGLITERAL: Any sequence of characters except ’"’ between ’"’ and ’"’.
  - ex) "Hello World"
- COMMENT: Anything that starts with "−" and lasts till the end of line.
  - ex) – This is a comment

Before actually crafting the regular expressions, fragments were defined to make them easier to develop. These fragments defined our DIGIT0 (0-9), DIGIT1 (1-9), and NL (new line). Using these fragments allowed us to craft the following regular expressions:

```
STRINGLITERAL :'"'~('"')*'"';
INTLITERAL : DIGIT1*DIGIT0+;
FLOATLITERAL : INTLITERAL'.'DIGIT0*|'.'DIGIT0+;
COMMENT : '--' ~[\r\n]* NL -> skip
```

Once the file was composed it was saved as "Little.g" and the ANTLR tool was used to create the lexer that was required. The last thing required for the scanner component was a method to drive it and to test it using the provided test cases. Each test case is written in the Little language comes with the token output collected from a working compiler.

The language chosen to implement the driver was Python 3.6. While the ANTLR toolchain is developed in Java, both members of our team are more familiar using the python language. To create a driver in python, both ANTLR needed to be installed for python, and the lexer needed to be built in python. The former was achieved by simply running the first pip command, the latter was achieved using the second command.

```
pip install antlr4-python3-runtime
antlr4 -Dlanguage=Python3 Little.g
```

During initial planning and testing of the grammar file, an attempt was made to use C++ as the driving language. This proved difficult in practice however as neither team member is well versed in the use of C++. While it was possible to generate the lexer using ANTLR to work with C++, the inclusion of those files into the driver program were difficult to implement and it was decided to move to a more comfortable language.

After the ANTLR lexer and parser were generated for python, a short python program was created that would read in a test file, input that stream to the lexer and output the results in the desired format. For this to work a few imports were required; antlr4, LittleLexer, LittleListener, and LittleParser. The latter three imported modules were generated by ANTLR from the Little.g file outlined earlier. The created function is as

follows:

```python
def main(filename):
    # Create a lexer stream by passing filestream to generated lexer
    lexer = LittleLexer(FileStream(filename))
    # Read the first token
    token = lexer.nextToken()
    # If the token isn't the end of file token continue
    while token.type != Token.EOF:
        # create name variable that hold relevant token information
        name = lexer.symbolicNames[token.type]
        # Output token type and token value on separate lines
        print("Token Type: {}".format(name))
        print("Value: {}".format(token.text))
        # Update token to reflect next token in stream
token = lexer.nextToken()
```

Using the defined function and above outlined grammar file, the test cases were supplied to the driver and the outputs were compared using the diff command. Results from the diff command indicated a 100% match with the output test cases provided, indicating that the scanner portion of the compiler is functioning correctly. This allows the project to continue on to the parser stage. **Problems - Scanner** With the exception of attempting to use C++ for this stage of the project, no difficulties were encountered. It was straightforward to create the ANTLR grammar file with a little extra time dedicated to creating the regular expressions outlined above. Once the decision was made to use python the implementation of the driver program was actualized rapidly with little pain.

2.3 Parser

The next stage after the scanner is the parser. The goal of the parser is to take the tokens provided by the lexer and create a parse tree which will then determine the validity of the code for the defined grammar set. While the lexer is able to use a simple regular expression format to create the grammar rules required, this is insufficient for the production of a parser. This is due to the standard programming languages implementation of nested components, whether that's nested parentheses or nested structures a regular expression cannot be crafted that fits all types of nested systems.

A solution to this problem is context free grammars (CFGs). A CFG is a grammar defined by a set of terminals, a set of non-terminals a start symbol and a set of productions. This is generally seen in the format G = (Vt,Vn,S,P) where:

- Vt is the set of terminals
  - Terminals are characters that can be input by the user.
- Vn is the set of non-terminals
  - Non-terminals are characters that cannot be input by the user and are used by the parser.
- S is the start symbol
  - The start symbol indicates the start of the program.
- P is the set of productions
  - Productions are the rules that define the functionality of the language.

This section of the project was reliant on the implementation of the production rules defined for the Little language. These production rules were provided to us in the grammar.txt file referenced in the previous section. Information was taken from the grammar.txt file and rebuilt using the syntax and requirements of the ANTLR language and saved into the Little.g file also used in the previous section.

Finally the python driver file also needed to be modified to check if the provided programs were either acceptable programs for the defined grammar or not acceptable. This was done by leveraging the existing ANTLR method parser._syntaxErrors. If the value returned is equal to zero then the program is accepted, if it returns anything else, the program is not accepted. The driver program can be seen below.

```python
def main(filename):
    # Create a lexer stream by passing filestream to generated lexer
    lexer = LittleLexer(FileStream(filename))
    # Create token stream from lexer
    stream = CommonTokenStream(lexer)
    # Initialize parser using token stream
    parser = LittleParser(stream)
    # Prevent parser error output (Tired of watching it fail)
    parser.removeErrorListeners()
    # Run parse tree using token stream initialized above
    parser.program()
    # If program is valid print accepted, otherwise print not accepted
    if parser._syntaxErrors == 0:
        print("Accepted")
    else:
       print("Not accepted")
```

The implementation of the parser in this project was through python 3.6 using mainly text editors to modify and run the scripts created, while this worked easily on our personal computers we did have to modify our code a decent amount to get running correctly on the lab computers. This was caused by the fact that the lab computers are running a version of ANTLR from 2015. Our group lodged a ticket with IT to see if we could get this updated to a more recent version of ANTLR. With the lab computers, UIT was unwilling to update to a newer version of ANTLR then what the operating systems native repositories supported. Another possible avenue was to look into using the esus servers to run our project code. In this case UIT informed us that the servers were running RHEL (RedHat linux) which only had a very antiquated version of ANTLR supported by the software repositories.

Problems - Parser

Some of the problems that we ran into at this phase of the project were actually related from Step 1 of the project where the lab computers were running a different version of ANTLR then what we were running. Ending up causing an error in our our original project code. The main issue being that the supported version of ANTLR was from 2015 and we were using a recently released version(2019). The solution was to download the matching run-time to the ANTLR4.5.2 and change some of our calls in the Micro file to target the python3 interpreter instead of leaving it up to the ambiguous python call. Since we didn't have administrator privileges, we had to install the ANTLR run-time locally to the project instead of installing to the normal /usr/local/bin or /usr/bin directories. The code below shows an example of installing with pip to a local directory.

```
if !(test -f antlr4/Lexer.py)
then
    DIR=$(pwd)
    pip3 install antlr4-python3-runtime==4.5.2 --target=$DIR
fi

if !(test -f Little.tokens)
then
   antlr4 -Dlanguage=Python3 Little.g
fi
```

2.4 Symbol Tree

Step 3 of this design was to develop the project to a point where the user could feed in a example Little file and define a structured tree detailing all the different seen aspects of the code fed in. This structure defines how a logically driven compiler looks at a piece of code and determines how to break it down for a later code generation. The structure generated is based on the scope of the code, thus defining where certain symbols are defined in the code. Specific information won't be defined since this form of the code is considered non-target specific.

An example of what the symbol tree would be built off of is displayed in the following code snippet from test11.micro. A file that was provided to test our design and confirm functionality.

test11.micro

```
PROGRAM test BEGIN
        INT a,b,c;
        FLOAT x,y,z;
        INT h,j,k;

        FUNCTION INT function1()
        BEGIN
        RETURN 35*45;
        END
        FUNCTION INT function2()
        BEGIN
                RETURN function2();
        END
        FUNCTION INT main()
        BEGIN
        WRITE (a);
        READ (b);
        a := function1();
        c := function2();
        c:=a+b;
        END
END
```

Which when running this as a input to the Micro file will generate an structured tree that details all of the scoped sections of the code.

test11.out

```
Symbol table GLOBAL
name a type INT
name b type INT
name c type INT
name x type FLOAT
name y type FLOAT
name z type FLOAT
name h type INT
name j type INT
name k type INT

Symbol table function1

Symbol table function2
Symbol table main
```

Figure 1 shows a visual of the scope of test11.micro, with initialized variables in proper scope.

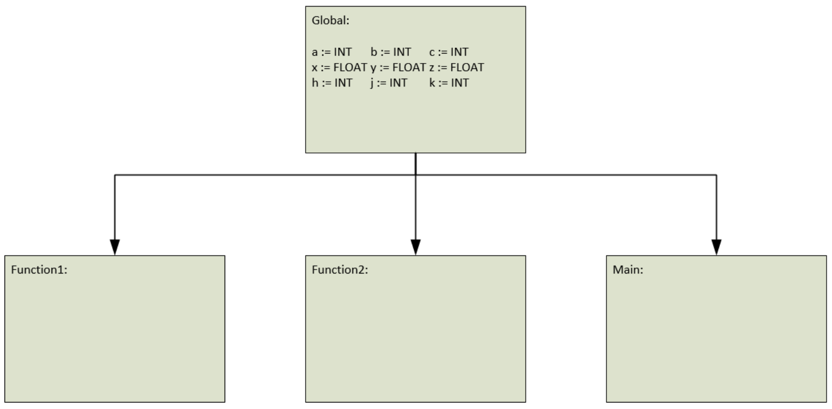
Global:
a := INT b := INT c := INT
x := FLOAT y := FLOAT z := FLOAT
h := INT j := INT k := INT
Function1:
Function2:
Main:

Figure 1: Visual of the symbol tree for test11.micro

Problems - Symbol Tree
The group didn't have many issues with completing this section of the project. Most of the new code is copy-pasted-modified functions to make sure all the cases that can be reached. One thing that the group addressed for this submission was the output of the install functions making sure that antlr4 will run on the CSCI lab computers. What we did in this case add an directional statement that took the output of the command and put it into a designated null section of Fedora Linux. What was added to each install command was a "> /dev/null". This allowed the Grading script to run with no discrepancies between the the output files and the user-generated output files, passing all of the given test cases.

```
    pip3 install antlr4-python3-runtime==4.5.2 --target=$DIR > /dev/null
```

2.5 Code Generation

The fourth and final section of the project involved the code generation based on the symbol tree output. To do this three separate steps needed to be completed. First was to generate an abstract syntax tree (AST) based on the code provided to the compiler. An AST is essentially a simplified version of a parse tree. With this in mind, the generation of an AST is technically optional but makes the rest of the process much simpler. The second step was to convert the AST into a sequence of IR Nodes that implement the provided function using three address codes. Finally, the IR nodes are traversed to generate assembly code capable of performing the function provided.

2.5.1 AST

As mentioned previously, the abstract syntax tree is simply a simplified, more readable version of a parse tree. This can be seen in the following figures, starting with an example parse tree in Figure 2

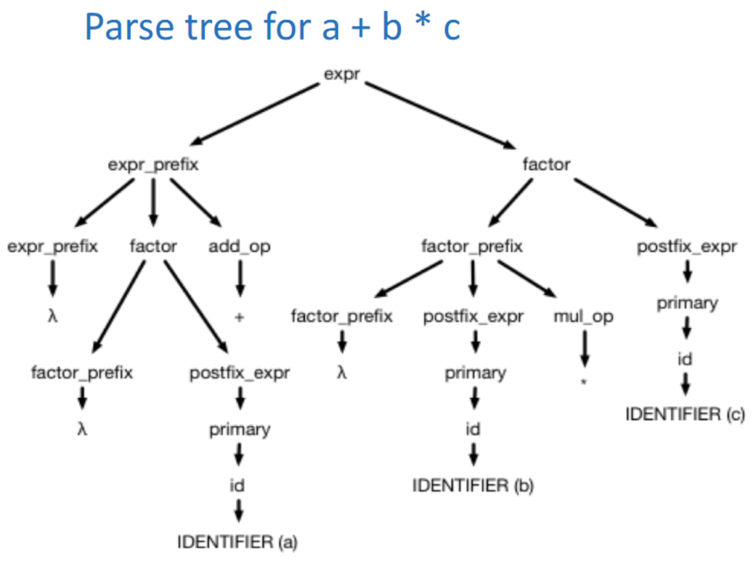

Figure 2: Parse tree example of a+b*c

This figure shows that a parse tree includes a lot of information that isn't necessarily needed to generated the associated assembly code. However, this information can be distilled into an AST tree that is much simpler to read and only includes the nodes relevant to the generation of the associated assembly code.

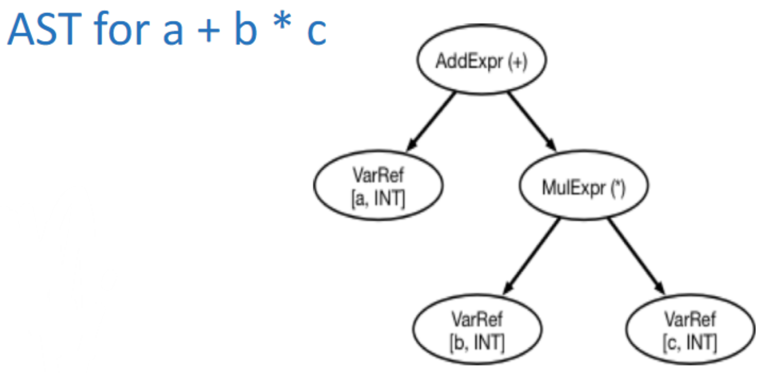

Figure 3: AST example of a+b*c

The AST figure replaces the detailed parse tree example containing 26 nodes with a much simpler tree containing only 5 nodes. This is done by passing information up the parse tree using semantic actions, resulting in the generated AST.

2.5.2 Three Address Code (3AC)

Once the AST is generated, the next stage is to generate IR using 3AC. 3AC serves an intermediate representation of the code where each instruction consists of, at most, two source operands and a single destination operand. However, at this stage the idea of registers does not yet exist. Instead of implementing code through the use of registers, temporaries are used. Temporaries serve as variables that are used to hold the intermediate results of computations in preparation of assembly code generation. An example can be seen in the following code segment.

```
Source Code = d := a + b * c

MULTI    b c \$T1
ADDI     a \$T1 \$T2
STOREI   \$T2 d
```

This code segment illustrates how a simple function can be broken down into the corresponding 3AC code. Following the order of operations, it can be seen that first a multiply is performed with two source operands, b and c, which is then stored into a temporary called T1. The results of that multiplication is then added to a and stored in another temporary called T2. Finally the value of temporary T2 is stored into d.

Ultimately, this step in the process will be used to convert the entire AST into a series of 3AC codes. The compiler being built as part of this project includes 15 different 3AC codes that can be used in conjunction with each other to describe the intermediate stage of the code generation. These codes can be seen in Figure 4.

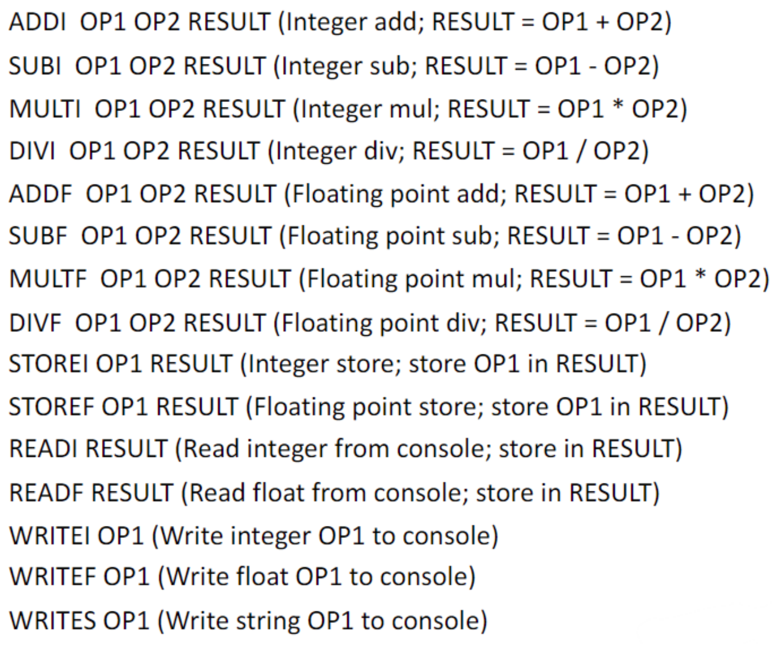
ADDI OP1 OP2 RESULT (Integer add; RESULT = OP1 + OP2)
SUBI OP1 OP2 RESULT (Integer sub; RESULT = OP1 - OP2)
MULTI OP1 OP2 RESULT (Integer mul; RESULT = OP1 * OP2)
DIVI OP1 OP2 RESULT (Integer div; RESULT = OP1 / OP2)
ADDF OP1 OP2 RESULT (Floating point add; RESULT = OP1 + OP2)
SUBF OP1 OP2 RESULT (Floating point sub; RESULT = OP1 - OP2)
MULTF OP1 OP2 RESULT (Floating point mul; RESULT = OP1 * OP2)
DIVF OP1 OP2 RESULT (Floating point div; RESULT = OP1 / OP2)
STOREI OP1 RESULT (Integer store; store OP1 in RESULT)
STOREF OP1 RESULT (Floating point store; store OP1 in RESULT)
READI RESULT (Read integer from console; store in RESULT)
READF RESULT (Read float from console; store in RESULT)
WRITEI OP1 (Write integer OP1 to console)
WRITEF OP1 (Write float OP1 to console)
WRITES OP1 (Write string OP1 to console)

Figure 4: List of possible 3AC codes

Possible codes include the standard arithmetic operations of add, subtract, multiply, and divide using both integers and floats. Additionally, the ability to store and read can be done using both stores and floats. Finally the ability to write to console can be done using integers, floats, and strings.

2.5.3 Generating Assembly

The final step in generating assembly code, is actually generating the assembly code. To do so, the 3AC codes outlined above are mapped to corresponding assembly instructions contained in the Tiny instruction set. Mapping is conducted by iterating over the list of 3AC codes generated in the previous step and converting each to the corresponding Tiny code. For this project, the simulator being used to represent the Tiny architecture consists of 1000 registers, allowing for essentially one to one mapping between the temporaries generated in the previous step and the existing registers in the Tiny architecture.

2.6 Full-Fledged Compiler

The final compiler was constructed using the culmination of the above outlined steps, each feeding into the next step. Once the ANTLR environment was setup the scanner was generated, followed by the parser, then the symbol table and finally the semantic actions and code generation.

Construction of the compiler was ultimately rather straightforward. A few design decisions were required, mainly in steps 3 and 4. In step 3 it was necessary to decide whether to use an observer/listener method or a visitor method to generate the symbol tables. This step also required the decision to be made between hash tables/dictionaries, trees, or lists of lists for the data structure used to store the symbol table. In the case of this group, the listener was used due to the simplicity of use, and a tree structure was used for similar reasons. Additionally, most the examples in class as well as the next step of the project seemed to be geared towards using tree structures.

In step 4 the decision needed to be made to either generate an AST or skip that step and just base everything off of using a parse tree. Since the parse tree would need to be read in very much the same method that is used to generate an AST anyways, our group decided to build the AST. This also allowed for a good stage to debug the code. Having this intermediate step helped ensure that each stage of the process was being performed correctly.

Beyond the technical aspects of actually creating the compiler was the implementation of version control. Our group utilized two separate systems, one for the code-base and one for the report. The code-base was version controlled using github. Github allowed for development to be conducted remotely by both group members at any time and really sped up development time. For the report portion of the project, overleaf was used. Overleaf is an online latex editor that allowed for one or both group members to work on the report either independently or simultaneously.

In the end, proper utilization of version control, python programming and ANTLR grammar description allowed for the completion of the compiler project.

3 Conclusion and Future Work

Over the course of this semester our group was able to create a compiler for the Tiny language from beginning to end. Testing was performed at four critical junctions, after implementing the scanner, after creating the parser, after creating a parse tree generator, and finally after the code generation step. This testing was completed successfully for every test case provided with the exception of a single test script used in part 4. The testcase_adv.micro file provided resulted in an index out of range error that was traced back to the ANTLR generated files. Ultimately this was ignored after running every other test case provided successfully.

In the future, we plan to expand on the foundation created in this course. Expansion will focus on the ability to add more functionality to the compiler as well as memory management. Currently the compiler is capable of running the 15 3AC codes defined in Figure 4. Future utilization of this project should allow for the functionality of shifting statements as well as logical operations.

While the initial focus of the expansion will be on functionality, further work will also focus on the optimization of implemented memory. The Tiny simulator used in the final stage of the project incorporated 1000 registers, which is great as a proof of concept but wildly inaccurate in reflecting modern processor architectures. The final stage of the project utilizes this by allowing each temporary created in the 3AC generation stage to map directly to a register. In reality, the register management needs to be improved work more realistically with 16 registers which is commonly used in modern architectures such as ARM and the MSP430 from TI.