Compiler Project

2. Methods and Discussion

A. ANTLR Setup
B. Scanner/Lexer
C. Parser
D. Symbol Table

A compiler creates and maintains a data structure called a symbol table so that information can be stored about occurrences of various items in the language "such as variable names, function names, objects, classes, interfaces" and the like (https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm). A compiler will use this data structure in both synthesis and analysis. Depending on the language used, this table may serve several different purposes. These include variable declaration verification, type checking (semantic correctness of expressions and assignments), scope resolution of a name, and centralized and structured names of all entities. This symbol table can take the form of a linear list, linked list, binary search tree or a hash table. For small languages a linear or linked list is adequate since traversing this list wouldn't take too much time or computing power. For larger languages a hash table or binary search tree is necessary for reducing time (O(1) and $O(\log_2 n)$ respectively) and cutting down on the usage of the CPU (https://www.geeksforgeeks.org/symbol-table-compiler/). The information stored in the table about each symbol is its name, type, and attribute and can be formatted as follows:

<symbol name, type, attribute>

The first decision to be made was to use either the default setting (listener pattern) for ANTLR or choose the visitor pattern. The decision was made based on the pros and cons of each pattern. The visitor pattern is good for using the parser output directly for interpretation. Tree traversal is in full control of the user, meaning only one branch is visited in conditionals and ***n*** number of visits can be made in loops. The visitor pattern also seems to be more flexible. Both pattern have their virtues when the input is translated to a lower level such as instructions on a virtual machine. In his book, *The Definitive ANTLR 4 Reference* (Terence Parr 2012, https://media.pragprog .com/titles/tpantlr2/listener.pdf), Terence Parr states:

> *"The biggest difference between the listener and visitor mechanisms is that listener methods are called by the ANTLR-provided walker object, whereas visitor methods mush walk their children with explicit visit calls. Forgetting to invoke visit() on a node's children means those subtrees don't get visited."*

Another important deciding factor between these two patterns is that the visitor uses the *call stack* and the listener used an *explicit stack*. The explicit stack is "allocated on the heap" and managed by the walker provided by ANTLR (https://stackoverflow.com/questions/20714492/antlr4-listeners-and-visitors-which-to-implement#30762056). This means that for large inputs the visitor pattern could run into stack overflow issues but the listener pattern wouldn't have

a problem. The main deciding factor came with the amount of management it would take to use the visitor pattern. With the listener pattern a programmer is only interacting with the provided tree walker. This seemed like the least amount of added headache for the most gain even though the language is small. The second decision was to come to a consensus on what type of data structure to use in making the symbol table. Since this is a little language a stack was decided upon for this purpose. Three changes to the grammar file became necessary at this stage in the project. As seen below in Figure 1, *func_declarations:* recursively refers to itself or an empty string. The caused a new function declaration to be created after an existing declaration that was empty. The second line in the figure shows the change to fix this issue.

```
21 func_declarations: func_decl func_declarations |;
22 func_declarations: func_decl func_declarations | func_decl;
```

Figure 1: Change to func_declarations.

The second change to the grammar file was for the *if_stmt*. An if statement doesn't necessarily need to have an *else* block to follow it. The change made was so that the *else* block would be optional, as seen in Figure 2.

```
47 if_stmt: 'IF' '(' cond ')' decl stmt_list else_part 'ENDIF';
48 else_part: 'ELSE' decl stmt_list |;
47 if_stmt: 'IF' '(' cond ')' decl stmt_list else_part 'ENDIF' | 'IF' '(' cond ')' decl stmt_list 'ENDIF' ;
48 else_part: 'ELSE' decl stmt_list;
```

Figure 2: Change to if_stmt and else_part.

The third change was in the *pgm_body* declaration. Since one of the test files contained a body declaration but no function declarations, an adjustment to the grammar file was necessary here (see Figure 3).

```
5 pgm_body: decl func_declarations;
5 pgm_body: decl func_declarations | decl ;
```

Figure 3: Change to pgm_body.

There were six total new classes used for part three of the project. The first was Symbol.java to represent a symbol held in a table. The next was SymbolTable.java; this holds all the symbols with a single scope. The class SymbolTables.java links all the tables in their relative scopes. The last of the table classes was SymbolTableVisualizer.java. This class recursively traversed all of the symbol tables in a tree structure while printing each of the table names and contents. Two enumerator classes were also added to support the symbol classes. The first held the possible attributes of a symbol (int, float, string, or void). The second was for the two possible types of a symbol (var or procedure).

The difficulties for this portion of the project came in the form of a few smaller hurdles to get past. One of the example output files listed the attribute for a procedure to be an *int* but it didn't seem to be an integer. The issue ended up being reading the return type from the procedure. For the Little language a procedure can either return an integer value (*int*) or return nothing (*void*). So either will show up depending on the whether the procedure returns anything or not. Another small issue that needed to be hunted down was that all child tables added to the *global* table and beyond were added twice. This turned out to be a minor oversight. It was thought that children needed to be added manually but when the

SymbolTable.java class was called with a new symbol it automatically added it as well. Once found, this was a simple matter to correct. Another thing that was realized at this point in the project was that not all the methods in the LittleBaseListener.java class needed to be filled out. Only the methods that dealt with variables, procedures, integers, floats, strings, and voids.