**Compiler Project**

1. **Methods and Discussion** 
   1. ANTLR Setup
   2. Scanner/Lexer
   3. Parser
   4. 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(log2 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, 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.” (*The Definitive ANTLR 4 Reference,* Terence Parr 2012, https://media.pragprog .com/titles/tpantlr2/listener.pdf) 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. …

Difficulties were numerous but not impossible to overcome.