CS 5363 001 Programming Languages and Compilers Spring 2016

Course Project Final Report by Venkat Rahul Dantuluri (fpj626)

This final report comprises of a brief discussion on how I developed my course project which is to build a standard compiler converting TL language into MIPS assembly code. The grammar and specification for TL language are provided beforehand.

For development purposes I used Python 3.5 language and Pycharm IDE. Pycharm is a new and robust IDE used for development of robust python Application. In the next few sections we will describe the Project Structure and design in order to better understand my development process.

Project Design:

The project 'compiler' contains two kind of python files. One kind act as data carriers and are part of the data structure used, other python files are the main files that hold the logic of execution. I will be describing the data structure for each phase of compilation followed by the brief description of logic involved.

1. Tokenization Phase: This is the preliminary phase of compilation and consists of scanning the source file and generating series of tokens. The data structure used is in **mytoken.py** and is a class called Token. Token class encapsulates value of the token and type of the token. The main logic used here is to create and code individual Finite state machine for each type of tokens you find in the code. For example we have a rule that all identifiers should be in capital case alphabet or numerals. We create a method in **scanner.py** that implements the FSM that is used to accept only such valid identifiers. In case there is a violation the FSM method will return False which tells us that there is a tokenization or Lexical error in the code and code does not follow the TL lexical or naming convention.

2. Parsing Phase: Once we have generated stream of tokens from the source file we pass the stream of tokens for next phase to a file named **tl_parser.py.** This file contains the main logic for parsing. The data structure used in this phase of compilation is in the file **Node_tl.py** in a class called Node. The output of parsing is a Parse tree hence the Node class represents a B-tree structure containing references to its child nodes. I am used LL1 top-down parsing technique to implement my parser. Accordingly I first created First and Follow sets based on the grammar given and then constructed the parse table. Creating first and follow sets and constructing parse table are not programmatic

activities and are done on paper. For each production in the grammar which is devoid of any left recursion and is left factored, I created a method in **tl_parser.py** file that simulates the production and checks whether the next token is syntactically accepted or not. In case we find a token that violates the syntax represented by grammar we raise an exception called **ParseError** which is defined as a Python Exception class.

3. AST Generation: From a coding perspective this was the phase that is the most complicated for me. In this phase I dealt with converting parse tree into an AST tree for better and easy utilization in later phases of compilation. The parse tree undergoes two rounds of processing to be completely modified into an AST or Abstract Syntax Tree. In the first round I process the parse tree and remove epsilon nodes, unnecessary non terminal nodes that do not derive any terminal. Once done, we have a tree that is half baked. We still need to rely on bunch of nodes for traversal. This half-baked tree is encapsulated using a B-tree structure in file **ast_node.py.** In the next round of processing we use a more rigid data structure called **AST.py**. AST.py contains an abstract class called AST which is implemented by every class that is part of AST. Structure of AST is explained in next section.

4. Semantic Analysis: Here I traversed over the AST class or generated AST tree to catch any type errors or declaration errors. For this purpose I first created **symbol table** which is a python dictionary that stores what all declared identifiers followed by what is the type of the identifier. After populating the symbol table I proceeded to traverse the AST from Top-down left-right fashion. Every expression is checked by comparing the left expression type with right expression types. Also certain type rules like comparison expression accepting only integer left and right expression is also assessed. If any identifier is encountered which is not in symbol table or any type mismatch occurs I am throwing **TypeError** exception. Also the corresponding **.ast.dot** file that is generated will highlight this type error by color coding the entire path from source of the error to the parent.

5. Intermediate Code Generation: In this phase of my compiler, I generate the 3-address code. Here I segregate all the statement lists into one block and whenever new statement lists are created I generate new block and store new statement lists in the new block. Each statement of a statement list is converted into corresponding Instruction and stored in **Instruction.py**. **Block.py** encapsulates a block and all the instructions in that block. There is a logical flow from one block to another block which is pictorially represented by **.cfg.dot** file created. This entire logic is captured in **IntermediateCodeGenerator.py** file

6. MIPS Code Generation: Here I converted the 3-address code into a MIPS code. The translation is fairly simple and straight forward. Each instruction in 3-address code has set of MIPS code statement involved. I simply replaced them and flushed them to the **.s** file. The logic is very simple. First I allocated memory address pointed by **$fp** or file pointer to each register

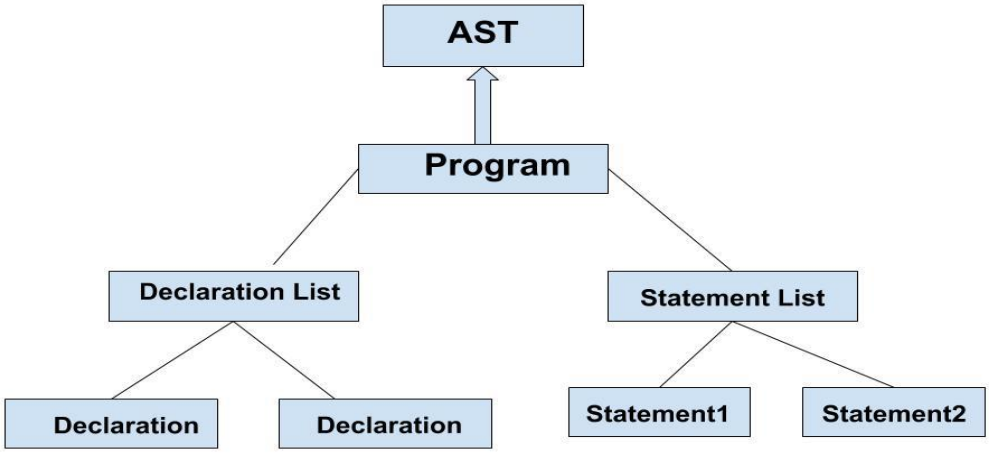
corresponding to the identifiers and literals used. Then later on manipulated on the memory address to hold values that were once held by registers of 3-address code. This logic is encapsulated in **MipsCodeGenerator.py** file. This is file that is executed using the shell script. It internally calls other python files.

Project Structure:

All the important data structures and logic used were discussed in the previous section. In this section I will showcase the structure of the project and structure of various data structures used.

1. Token class: Used to store type and value of each token.
2. Node class: In Node.tl file. Used to store the Node structure of a parse tree.
3. ASTNode class: Used to store the half-baked AST tree.
4. AST.class: Abstract class is inherited by other nodes of AST

   AST Structure:



5. Block class: Contains list of instruction belonging to a block