Pascal to MIPs Compiler

Joseph Miller

April 19, 2018

1 Overview

This program will be able to compile Pascal code into MIPs assembly code. The compiler will be made divided into 5 major sections:

- Scanner
- Parser
- Syntax Tree
- Semantic Analysis
- Code Generation

2 Design

2.1 Scanner

This program will be able to take in a file with Pascal code and output each token in the order that it was written. Each of the following keywords and symbols will be identified as tokens, as well as variable names (IDs).

- Keywords: and, array, begin, div, do, else, end, function, if, integer, mod, not, of, or, procedure, program, real, then, var, while
- Symbols: ;, ,, ., :, [,], (,), +, -, =, <>, <, <=, >, >=, *, /, :=

This project is made of 3 Java/Class files:

- **Scanner**: the DFA scanner made by JFlex to analyze the Paacal code and identify token types.
- **Token**: the token object creator, this file also takes in what kind of token it is and adds a more specific distinct type by using a look-up table (switch function / if-else function). Each Token object contains the lexeme and the type.
- **TokenType**: creates the specifications for a the TokenType attribute of the Token attribute

Illegal symbols and sequences not accepted by our simple pascal compiler will have the TokenType: ERROR. A visual of the tokens created can be seen in Figure 1, one of which produces an error.

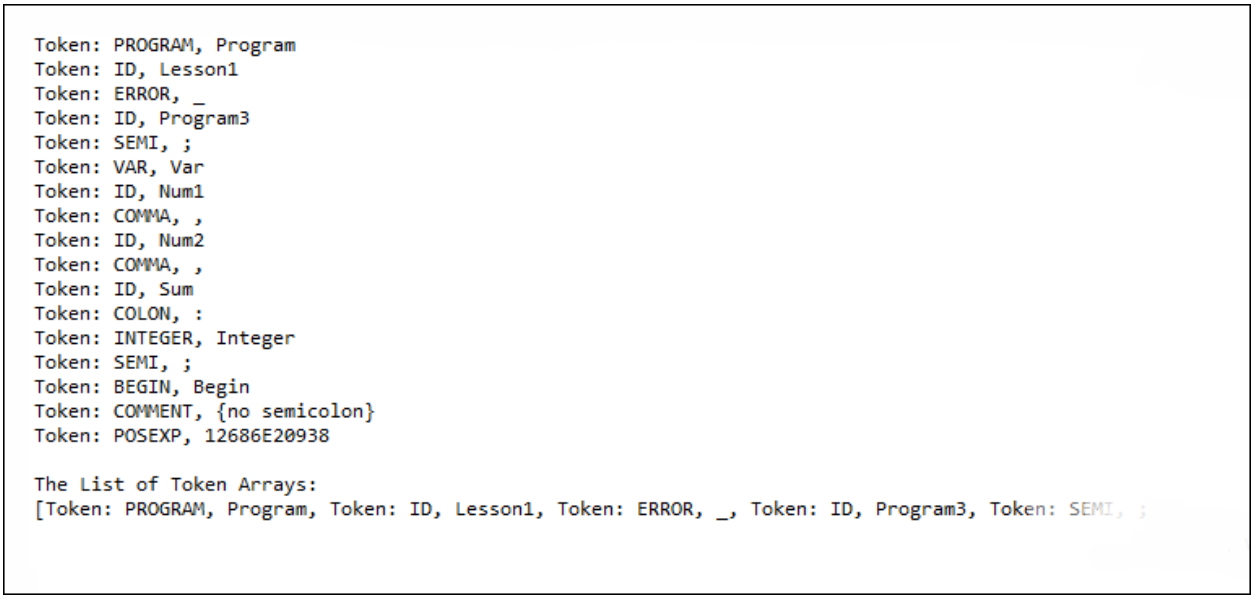

Figure 1: Token output to console

2.2 Parser

The parser takes the tokens from the scanner and verifies whether it is correctly formatted Pascal code. The parser also creates the syntax tree, a series of nodes that format the Pascal code for assembly code generation.

This project is made of 3 Java/Class files, only one of which (Parser) relates explicitly to the parser:

- **Parser**: The parser class takes the series of tokens created by the scanner and uses functions for each production rule to either accept the Pascal code or reject it. If an error results it will also spot the location and reason for the error and end a message to the console informing the coder of the error. The parser adds identifiers to the symbol table with their proper type, and also uses the symbol table to resolves the ambiguity in the "Statement" production rule, where it calls either the "Procedure statement" or "Variable " production rule respectively. The parser also creates the syntax tree, using the classes in the syntax tree as nodes to construct a tree of the pascal code.

- **SymbolTable**: The symbol table is independent of the parser but is put in the parser package since the parser using it most and its not a large enough part of the compiler to be given its own package. The symbol table is used to take in all identifiers used in the code and associate them with the types of identifier (or symbol) they are. Other parts of the compilers will need to reference the symbol table.

  SymbolType: This is an enum file for SymbolTable and has no relation to the parser itself. The five types of identifiers are: PROGRAMTYPE, VARIABLETYPE, FUNCTIONTYPE, PROCEDURETYPE, and ARRAYTYPE

2.3 Syntax Tree

The syntax tree package is a collection of classes that each represent a node in the syntax tree and have methods and functions that connect to other nodes in the proper order. An example of some pascal code can be found in Figure 2 and the corresponding syntax tree code can be found in Figure 3 These are the classes/nodes:

- Nodes: Assign, Assignment Statement, Compound Statement, Declarations, Expression, Function, If Statement, Operation, Procedure, Program, Read, Sign, Statement, Sub Program Declarations, Sub Program Head, Sub Program, Syntax Tree, Value, Variable, While Statement, Write

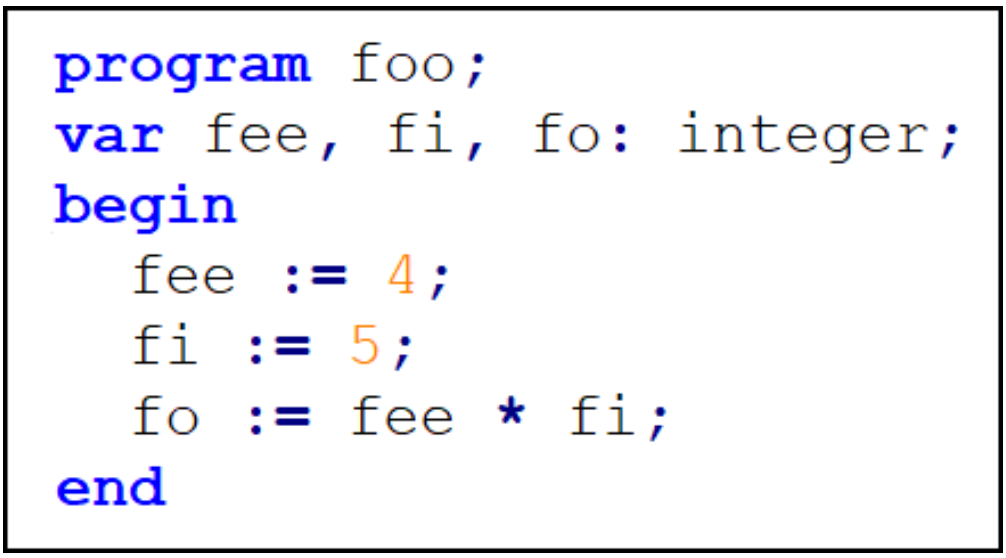

Figure 2: Simple Pascal Code

2.4 Semantic Analysis

The semantic analysis takes in a program node and goes through the syntax tree in order to make sure all variables are declared before they're used and also declares expressions either real or integer. The methods are as shown in 4 and described below. None of the errors stop compilation.

- analyze: Gets called by main to execute semantic analysis. It gets the compound statements to be parsed for expressions and calls verifyVarDecs anf assignExpTypes.
- verifyVarDecs: Verifies that all variables used were declared before they were used.
- assignExpTypes: Gets a compound statement and goes through the statements and depending on what syntax node they are an instance of, gets the expressions to be declared.
- setExpTypes: Takes in an expression node and sets the type of expression.
- setVarVal: Takes in an expression node and sets a variable or value to be a certain type (real or integer)
- getLNode: Takes in an expression node and returns the left node (not always necessary)
- getRNode: Takes in an expression node and returns the right node (not always necessary)

2.5 Code Generation

The code generation takes the syntax tree after its been analyzed and creates the MIPs assembly code. Like semantic analysis, it goes through the statements and calls the respective method for each kind of statements. But in this case the statement creates assembly code which is added to the large assembly code string.

The driver takes the asm code and writes it to a file: [input file name].asm, which can then be used on a MIPs processor or simulator.

3 Change Log

- 18 January 2019 - Miller - Created

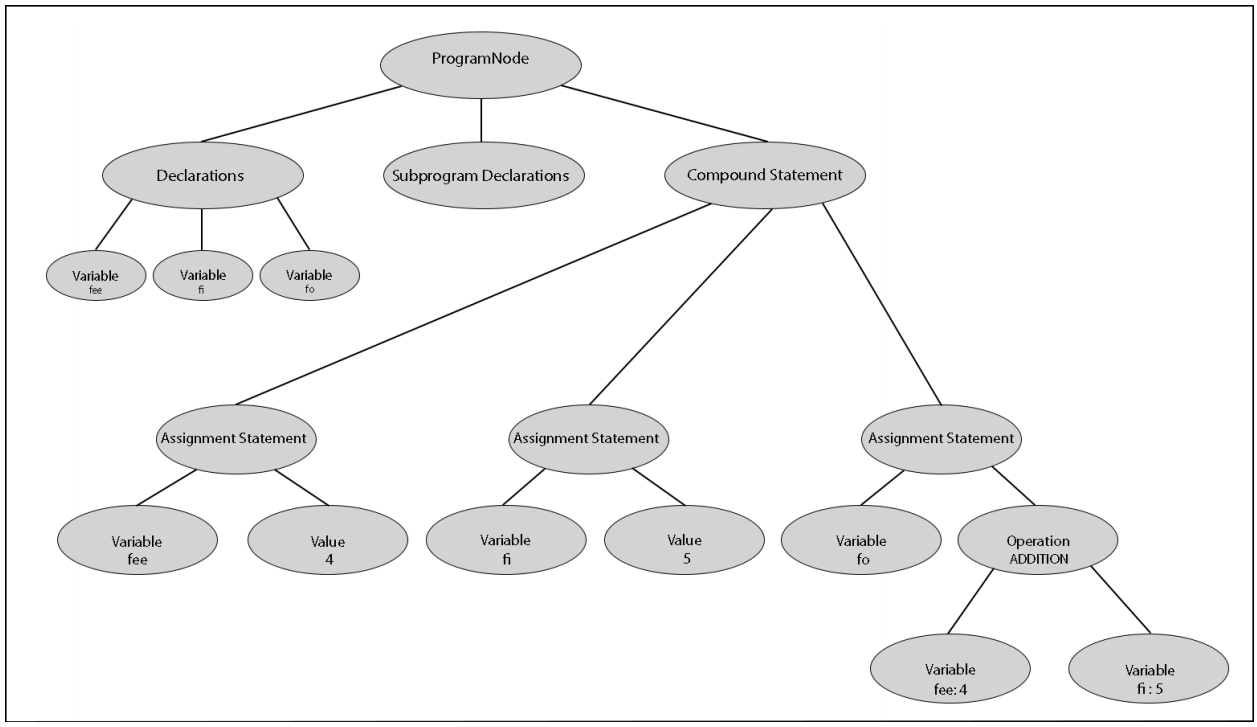

Figure 3: Syntax Tree Corresponding to Simple Pascal Code

```
public class SemanticAnalysis {

    private ProgramNode progNode = null;
    private SymbolTable symTab = null;
    private HashMap<String, TokenType> varTypes = new HashMap<String, TokenType>();

    public SemanticAnalysis(ProgramNode progNode, SymbolTable symTab) {

     * function called by main, organizing/executing semantic analysis, puts
    public ProgramNode analyze() {

     * verifies that all variable in the program are declared before use, yields an
    private void verifyVarDecs() {

     * parses through the statements and assigns a type (real or integer) to each
    private void assignExpTypes(CompoundStatementNode compStatNode) {

    private void setExpTypes(ExpressionNode expNode) {

    private void setVarVal(ExpressionNode expNode) {

    private ExpressionNode getLNode(ExpressionNode expNode) {

    private ExpressionNode getRNode(ExpressionNode expNode) {

}
```

Figure 4: Semantic Analysis Methods

- 3 February 2018 - Miller - Added parser section and content
- 11 February 2018 - Miller - Added symbol table info under parser section
- 3 March 2018 - Miller - Added syntax tree section and content
- 2 April 2018 - Miller - Added semantic analysis section with figure
- 14 April 2018 - Miller - Added code generation section