# Compiler Software Design Document

Project Overview:

This application is a mini-Pascal compiler that will be reading in a small version of Pascal and converts it into a MIPS assembly program. This document will be split into multiple sections that will cover in great detail each of the modules that go into creating the compiler.

## Part 1: The Scanner

1. Overview:

   This section of the compiler is a scanner created with Java and the JFlex library that will be reading in the small version of the Pascal language and creating tokens for each input.

2. Design:

   This section will consist of multiple files that will all be used to create a fully functioning scanner.

   There will be a jflex file that when compiled will generate a majority of the scanner code, however jflex does not generate all of the code needed so we will be writing functionality for handling the different input types. All of this functionality will be added to the jflex file which will then be compiled to create the scanner file.

   In addition to the jflex file there will also be a token file that will be called by the scanner and will handle all of the token creation functionality. The token created will consist of two parts:

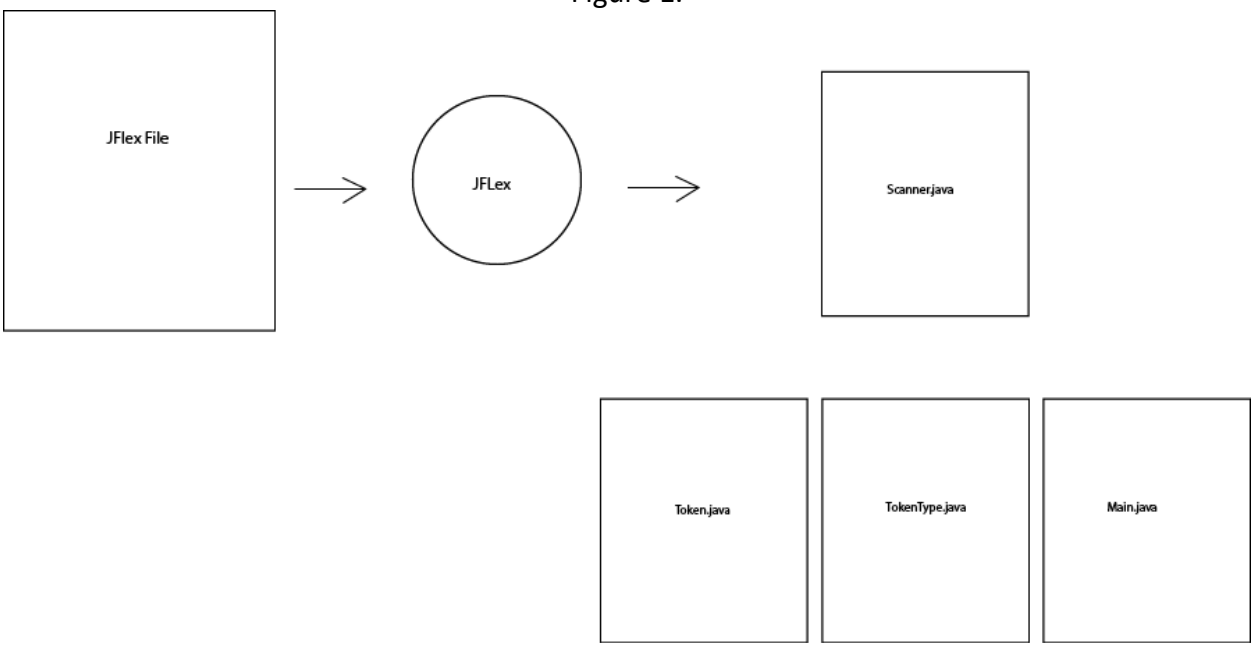
   - The lexeme, which is a combination of symbols to form a string or digit.
   - The type, which is determined by the end state of the scanner.

   There will also be a Hashmap lookup table that will also be called by the scanner. The lookup table will take in the current input's lexeme and check if it matches any key words in the Hashmap and will set the token type accordingly.

   Lastly there will be a main for testing purposes of all other files. This will be where the scanner is initialized and text files are inputted for testing to ensure that everything is working correctly with the scanner.

   For a look at how many files there will be and a visual representation of compiling the jflex look at Figure 1.

Figure 1.




Part 2: The Parser

1. Overview:

This section of the compiler is a top-down recursive descent parser created with Java that will be taking the tokens created from the scanner and reading them to ensure that the input file is a grammatically correct Pascal program.

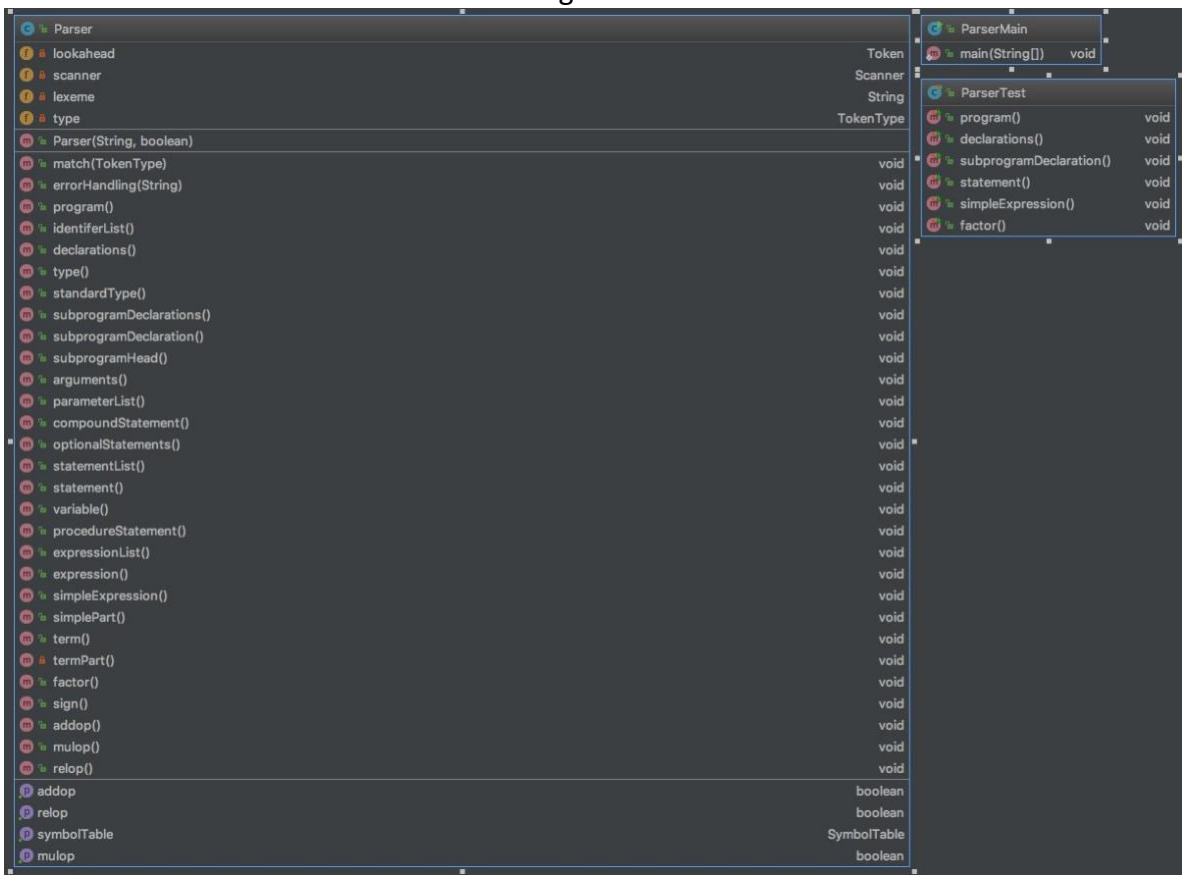
2. Design:

This section will be consisted of two new files: Parser.java and SymbolTable.java and will be used with the scanner files created in the first section.

The parser class will be where all of the functions will be created for each rule in the mini Pascal grammar as well as error handling for when an incorrect input is sent to the parser. The parser will operate by calling the scanner to read in an input file which will create tokens that will but read by the parser. As the tokens are being inputted the parser will be checking each token with the lookAhead function and will either be matching it to its token type or calling another grammar function that will eventually go down a chain of functions until a match is made; this will all be determined by the grammar function that is called initially. Once a token has been matched the parser will have the scanner call nextToken and the process will continue until the end of the inputted file.

See figure 2 for a look at the UML diagram of the Parser section

Figure 2.




Part 3: The Symbol Table

1. Overview:

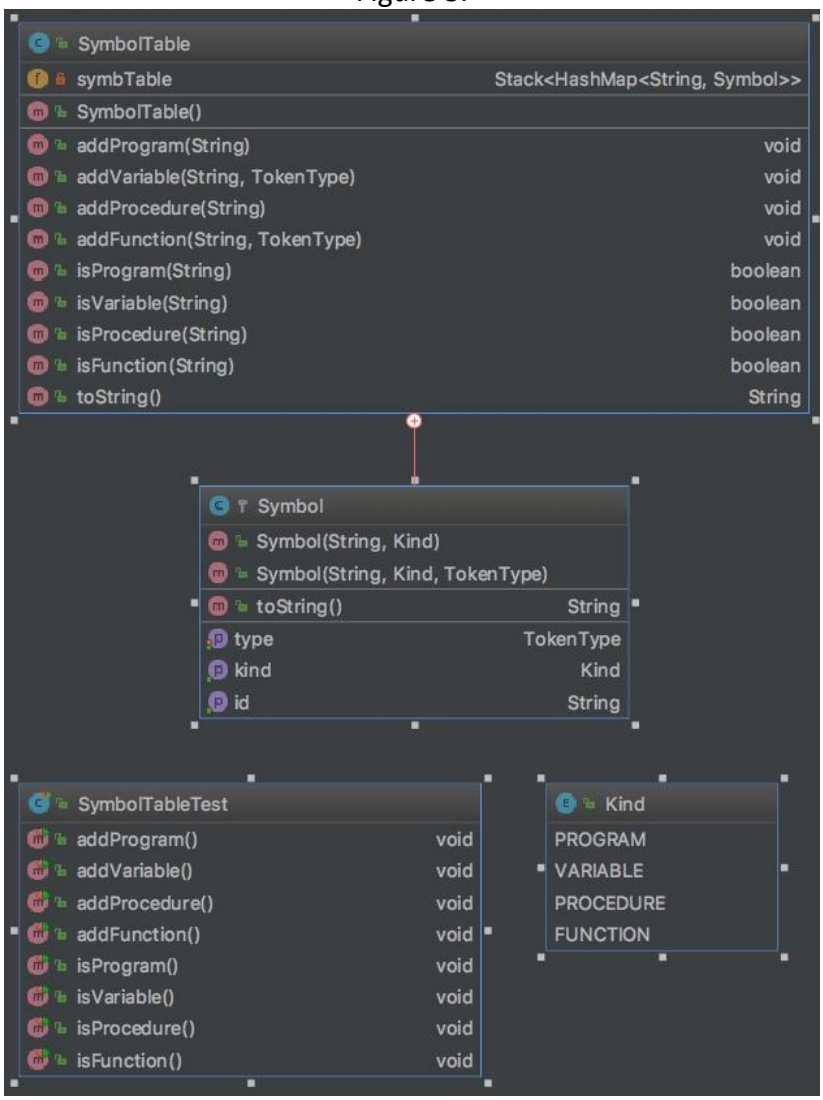
   This section of the compiler is the Symbol Table and will be used in conjunction with the Parser to resolve ambiguity with the mini-Pascal language.

2. Design:

   The symbol table class will be used initially for the storage of different id types. It will consist of a stack data structure that will store a Hashmap, which will act as a database table in the sense that each id stored in this Hashmap will have a name for each id entered along with a Symbol object which contains the attributes such as lexeme, kind, and any other information passed from the Parser. The symbol object creator will be an internal class called only by the Symbol Table class. Additionally, there is the enum class Kind which will be used for setting the four kinds of IDs entered (program, variable, procedure, and function.) Initially the parser will use this class for all ids found within the inputted Pascal program and will be used for handling parts of the grammar that are more ambiguous. As this project grows the symbol table will have more added on to it as needed.

See figure 3 for a look at the UML diagram of the Symbol Table section

Figure 3.




Part 4: Syntax Tree

1. Overview:

   This section of the compiler is the Syntax Tree and will be used in conjunction with the Parser to create a Syntax Tree containing nodes with information about the Pascal program.

2. Design:

   The syntax tree will be created by creating individual nodes for each key component of the Pascal program. Each of the different node classes will be

implemented into the corresponding Parser function and will have the Parser returning those nodes. The syntax tree visual representation will be created by calling each node class' indented toString function and concatenating together to form an indented representation of the syntax tree which will be written to a file via Compiler Main.

## Part 5: Semantic Analysis

1. Overview:

   This section of the compiler is the Semantic Analysis and will be placed within the Parser as a way of minor error handling for situations that throw errors that are not life threatening to the compiling process.

2. Design:

   There will be three types of semantic analysis implemented into the Parser class.

   1. Making sure all variables are declared before being used.
   2. Assigning a type (integer or real) to each expression and making sure the type of any expression node is printed in the syntax tree's toString.
   3. Making sure that types match across assignments such as declaring a variable type integer and not trying to later assign it to type real.

   Each of these types of semantic analysis will have the ability to throw an error that will be caught be a new error handling function. This new function will be responsible for throwing all non-threatening errors that occur and will be printing them to the console for the user to see. Each error message will have the sentence corresponding to the type of error occurring along with the line number in the program.

## Part 6: Code Generation

1. Overview:

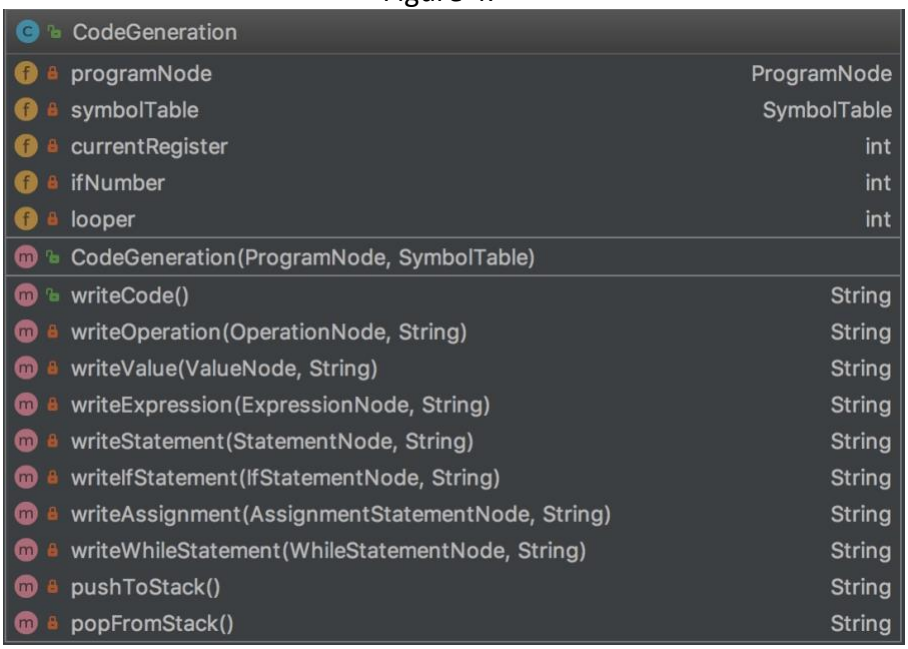
   The code generation class will be responsible for taking in the syntax tree created by the Parser and generating the MIPS assembly code.

2. Design:

   The code generation will consist of multiple functions that will be generating various parts of the MIPS assembly code. The generation class will be taking in the syntax tree from the Parser and then running it through the writeCode function. This function will serve a lot like the program function does for the Parser and will be the central hub for where a lot of the other functions are called. As the code generation progresses a string builder will be used to keep track of what has been generated. As each function is called and the

appropriate information is placed into this string builder more and more of the code will be generated and at the end of the process this string will be what the code generation class returns. If successful this string will be returned to Compiler Main which will then be written to a new file titled “MIPS.asm” which can then be ran through the QTSPIM software. See figure 4 for a look at the UML diagram of the Code Generation section.

Figure 4.




3. Change Log:

1-21-18: Created SDD: Bergstrom
1-31-18: Added Parser section: Bergstrom
2-11-18: Added Symbol Table section: Bergstrom
2-19-18: Updated Symbol Table & Parser sections: Bergstrom
3-09-18: Updated Syntax Tree section: Bergstrom
4-17-18: Added the Semantic Analysis and Code Generation sections: Bergstom