Pascal to MIPS Compiler

Software Design Document

Version <2.3>

HeeChan Kang
April 26th, 2018

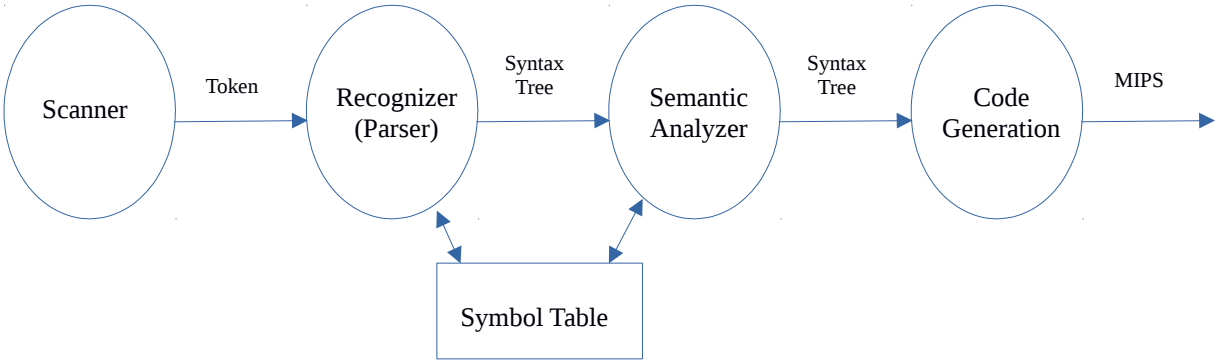
Revisions

Date	Description	Version
January 23rd, 2018	Created the document.	1.0
January 30th, 2018	Added cover page and separated sections.	1.1
January 30th, 2018	Added documentation about Parser.java	1.2
February 1st, 2018	Added graph containing the overview of the compiler.	1.3
Februrary 3rd, 2018	Added documentation about ParserTest.java	1.4
February 10th, 2018	Added documentation about SymbolTable.java, Kind.java, and SymbolTableTest.java. Edited Overview section.	1.5
February 18th, 2018	Added documentation about CompilerMain.java	1.6
February 20th, 2018	Added documentation for SymbolTableIntegrationTest.java	1.7
March 4th, 2018	Added documentation for SyntaxTree	1.8
March 25th, 2018	Added additional documentation for SyntaxTree	1.9
April 14th, 2018	Added documentation for Semantic Analyzer	2.0
April 14th, 2018	Added documentation for Code Generation	2.1
April 21st, 2018	Added documentation for final JAR product	2.2
April 26th, 2018	REAL values can be taken care of! Whoo!	2.3

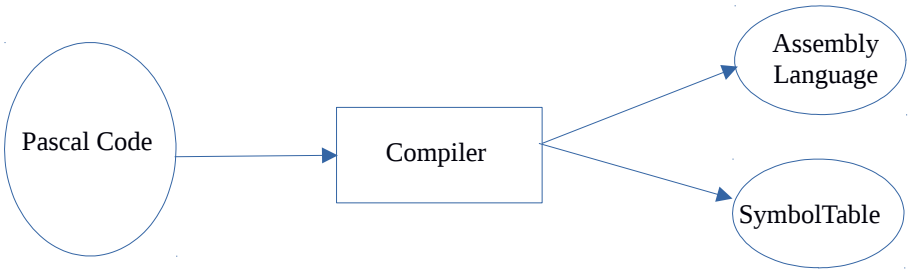
1. Introduction

This project is for Programming Languages and Compilers course, keystone for Computer Science majors, at Augsburg University. It contains a compiler in Java for Pascal, to finally, convert the code to MIPS. Its components are described below.

2. Overview



We want to be able to take in a Pascal file, and output an assembly language file and a .symboltable file that contains the symbols used in the code.



3. Implemented Components

/src/scanner/

a) Scanner.jflex – we used a JFlex tool to create Scanner.java which currently contains the skeleton of being able to read-in a file and creating their respective Token.

b) Scanner.java – created from the JFlex tool.

c) Token.java – This class contains the simple Token object, which contains String lexeme and TokenType.

d) TokenType.java – This class is an ENUM that contains all the relevant token types in Pascal.

e) LookUpTable.java – This class is a HashMap containing lexeme as a key and TokenType as value. The lexeme is matched to their respective TokenType using this LookUpTable.

f) ScannerTest.java – This class currently contains JUnit testing for yytext() and nextToken().

/src/parser/

a) Parser.java – contains a parser for Pascal based on the grammar provided by professor. On top of abiding the rules of the grammar, there are six other methods, isMulop(), mulop(), isAddop(), addop(), isRelop(), and relop() for simplicity.

b) ParserTest.java – contains JUnit testing cases for program(), declarations(), subprogram_declaration(), statement(), simple_expression(), and factor().

c) SymbolTableIntegrationTest.java - JUnit testing for statement() after integrating SymbolTable to the Parser.

d) SyntaxTreeTest.java – JUnit testing for the integration of SyntaxTree to the parser.

/src/symboltable/

a) SymbolTable.java – contains constructor for our symbol table implemented using a HashMap that holds lexeme as a key and an object called DataStructure that holds a lexeme and the kind of the ID that we would like to store.

b) Kind.java – This class is an ENUM that contains all the types of ID that we will be storing.

c) SymbolTableTest.java – contains JUnit testing cases for SymbolTable.add(lexeme, DataStorage) and SymbolTable.getKind().

/src/syntaxtree/

a) This package contains the code for our syntax tree provided by our professor, Erik Steinmetz. Each node will contain essential information of the code, which will be used to create a .symboltable file with neatly indented contents. Additional nodes were created and added by myself.

/src/semanticanalyzer/

a) SemanticAnalyzer.java – contains the code that uses the produced syntax tree and checks to see if any of the below conditions are unmet:

- All variables are declared before being used.
- ExpressionNodes will hold a type, and these types must match across assignments.

If a condition is unmet, code generation would not occur and the user would know what needs to be changed.

/src/codegeneration/

a) CodeGeneration.java – contains the code that uses the syntax tree that the semantic analyzer goes over to produce assembly language code. Real values are handled appropriately!

/src/compiler/

a) CompilerMain.java - Contains the main for the compiler; primarily, as of now, to test the production of code in assembly language.

4. Final Product

/product/

a) KangCompiler.jar – Final product of the project. Running this .jar file with a .pas file in the same directory would result in the production of three files: a symbol table, syntax tree, and the final assembly code.

b) UserManual.pdf – As its name suggests, a user manual to use the final product.