Maxwell Herron
CSC 451

AlmostC Compiler SDD

General Overview

This project is to be used to compile and run programs written in the AlmostC programming language. It is written in Java, and consists of seven main packages: scanner, parser, symboltable, syntaxtree, semanticanalyzer, codegenerator, and compiler. These packages all lie within the src folder of the root directory, titled Compiler. Each package is described in detail in the following subsections.

Scanner

The scanner used for this compiler is generated by a Jflex file, and all the necessary files for this component are found in the scanner package. The design of my Jflex scanner is as follows: the jflex file Scanner.jflex contains several regex patterns that the generated scanner will parse for. It also contains the instructions for what the scanner will do upon finding each pattern. There is a Token class, which has the attributes lexeme and tokentype. The lexeme is the literal string that is parsed, while the tokentype is another object, created by the TokenType class. TokenType is simply an enum of all necessary types of tokens for this assignment. Finally, there is a LookupTable class. The LookupTable is a hashmap that is used for all the symbol, operator, and keyword tokens. When the scanner picks up any of those tokens, it sends the lexeme as a key to the LookupTable to get back the value associated with it, and assigns that value as its TokenType. There is also a class for a custom exception object, called BadCharacterException.

This exception is thrown when the scanner picks up a lexeme that is not associated with any of the regular expression patterns defined in Scanner.jflex. It holds an error message that lets the user know what the bad character was.

Parser

The purpose of the parser is to read in an input, then determine the validity of the input, while also generating a structure known as a Syntax Tree. The parser used by this compiler is referred to as a Recursive Descent Parser. This is because it works by making a call to the top level parsing method, known as program, and recursively descending down through the grammatical productions, generating syntax tree nodes along the way. When it is complete, it returns a Program Node, which is the root node of the Syntax Tree generated from the input program. The Parser also relies on the Symbol Table to keep track of declared identifiers.

Symbol Table

The purpose of the symbol table is to encapsulate information about any identifier that the recognizer comes across. Some relevant information to hold would be the kind of identifier (variable, function, array, program), along with the datatype of the identifier. This will allow the compiler to keep track of all the identifiers, and actually have an idea as to what they are, as opposed to simply seeing them as just generic identifiers.

Syntax Tree

The syntax tree package holds all the necessary classes used to build the specific node types of the syntax tree, which is returned from the parser. Because of this, not much work is

actually being done within the syntax tree package, it just contains all the classes for the different node types. The package also contains a class dedicated to testing out the syntax tree classes.

Semantic Analyzer

Once the parser has generated a syntax tree from an input AlmostC program, the final step is to run semantic analysis on the tree. The semantic analyzer has three core responsibilities: ensuring all variables have been declared, setting a data type to all ExpressionNodes in the tree, and ensuring type matching across assignments. What is important is that these potential issues do not ‘break’ the compiler, they simply let you know that the input program cannot have assembly code generated from it if any semantic errors are encountered. All relevant files are found in the semanticanalyzer package.

Code Generator

The code generator is the final component of the compiler. The primary function is to take an input program node (after running semantic analysis on it) and generate a working assembly program. Specifically, it generates MIPS assembly code, which can be run in your preferred MIPS simulator. The generator starts by writing all boilerplate MIPS code and initializing all declared variables. It then iterates through all statements in the program main body, and calls an overloaded method writeCode() on all statements within to generate the assembly. Within the codegenerator package there is a test class that runs JUnit testing on assembly generation for ProgramNode, ExpressionNode, and StatementNode.

Compiler

The compiler package acts as the driver for all other packages in the compiler. In essence, it bundles together all the separate tools built in the compiler to get them working together. In its current state, the main is used to run the parser on an input AlmostC program. It then writes out the tabular form of the symbol table generated from the input file into an output file for the user to view. Along with that, it also writes out the indentedToString() of the ProgramNode that the parser generates from the input file.