CIS 4650 - Compilers

April 9th, 2022

Josh McCaskill

0790488

Arash Esfandiari

1019706

Checkpoint 3 - Project Report

Completed Components

**Overall Project Description**

The compilers project consisted of three main components, each of which was completed during a different checkpoint. The first component was the creation of Scanner that uses a grammar for the cminus language, which was implemented using Java classes and the CUP and Flex tools to parse input and create an abstract syntax tree. The second component of the project was the creation of Symbol classes which were used to check for semantic errors and output a Symbol Table. The final component of the compilers project was the creation of a CodeGenerator Java class to convert the output from the two previous steps into assembly code that can be ran by a TMSimulator.

**Checkpoint 1**

The two main components completed during Checkpoint 1 were the Parser and Scanner sections of a compiler.

The Parser was completed using jflex tool and can be seen with the cminus.flex file. This was used to separate the inputted cminus file into individual symbols.  
  
The symbols outputted by the Parser were then read by the Scanner, which was created with the CUP tool shown with cminus.cup. This file was used to define and implement the grammar found in the C- Specification file as well as implements error and error handling functios that corresponds to the rules of the cminus language.  
  
Included to these two main elements, the creation of Absyn classes corresponding to each of the terminals and non-terminals of the cminus language were created and used by the CUP file as embedded code.  
  
This embedded code was then used along with a ShowTreeVisitor Class to produce an Abstract Syntax Tree as output.

**Checkpoint 2**

The two main components completed during Checkpoint 2 were the SymbolTable and the SemanticAnalyzer portions of the compiler.

The SymbolTable was completed using the java symbol package and the creation of various symbol classes. It uses a HashMap to store the symbols created with the symbol classes found in the /symbol folder.  
  
Included to this was the creation of a SemanticAnalyzer class which uses the AbstractSyntaxTree from Checkpoint 1 to create each symbol and store it in the SymbolTable. Before creating a symbol, the SemanticAnalyzer performs type checking for semantic errors and reports any errors found to stderr.

The command line flags of -a and -s were added to the Main program specifying to print the AbstractSyntaxTree or the SymbolTable respectively.

**Checkpoint 3**

The main component completed during Checkpoint 3 was the creation of the CodeGenerator Java class which uses components of the previous two checkpoints to output a .tm file. This .tm file contains assembly code that can be inputted to and read by the TMSimulator to run the original .cm files that were parsed in Checkpoints 1 and 2. The CodeGenerator class makes use of the Visitor Pattern and the Symbol Table to create strings that correspond to TMSimulator instructions. Depending on the variables, functions and scopes within the inputted .cm file, the CodeGenerator will print instructions correlating these values to the required registers and memory locations to be used by the TMSimulator. These instructions are then saved to the .tm file which becomes input to the TMSimulator, resulting in the assembly instructions to be ran. This process can be executed by adding the -c flag to the run and compile instructions found in the ReadMe file provided. Included to these completed components, a file management system was added to organize the folder structure of outputted files and files required to run the program.

Techniques

Research was completed prior to implementation to allow for a better understanding of the process required to complete the project. This was done through the use of CIS 4650 lectures and lecture notes, as well as research on the TMSimulator that was to be used for the project.

The sample files provided were used with the TMSimulator for a better understanding of the required inputs and expected outputs of the system. Included to this, tracing through the provided TMSimulator .c code proved useful in understanding how the program works.

Once research was competed, progress for the project was broken down into smaller tasks as those explained below in the “Design Process”.

Design Process

Incremental programming techniques were used as:

1) Understanding of TMSimulator Assembly Commands

2) Understanding of TMSimulator Memory Allocation Requirements

3) Output of control structures, nested blocks, function calls and arrays to assembly code

3.1) Creation of Visitor Pattern class functions for cminus syntax

3.2) Creation of a FileWriter for output of Assembly Statements to a .tm file

3.3) Use of TMSimulator with outputted .tm file

4) Handling of “Out of Bounds” runtime error

5) Creation of testing files for TMSimulator containing various error types

Modifications

Improvements to the implementations done in the previous two checkpoints include:

Additional semantic errors added and more precise and specific output for error messages.

Improved output readability for the Symbol Table and the Abstract Syntax Tree.

A more organized folder structure for required classes and input/output files.

Lessons

Lessons gain during the completion of this project include estimating the requirements and time to complete these requirements properly. This would allow more development time and the opportunity to improve the project when unexpected errors, problems or misunderstanding of the specification occur. Another lesson that proved to be beneficial while completing the last segment of this project is that our group members worked much better when in the same room and were able to examine the code being written in person opposed remotely.

Assumptions and Limitations

Improvements

A possible improvement for this checkpoint would be to add implementation to test for more runtime errors. Since we have already completed the syntactic and semantic error checking, assuming the code determines all possible errors of these two types, then the only possible errors that could occur would happen during runtime. The current implementation only tests for one possibility of an out of bounds runtime error, though it is possible for other types of runtime errors to occur. Because of this it would be beneficial to research and test for these possible errors. This would be done with a better understanding of the cminus language, as well as with a better understanding of the TMSimulator and how it executes the .tm code.

Contributions

Josh

ReadMe

Documentation

Testing Files

Arash

CodeGenerator.java Class