Checkpoint 3

Ben Smyth, Eli Daniels, Nathan McGuire (Group 17)

CIS4650W24, Professor Fei Song

Design Implementations and Process

Design overview

This checkpoint's goal was to generate assembly code for the TM simulator for valid cminus programs. To achieve this goal, we first needed to understand the TM simulator to be able to write code that could be executed by it. This proved to be challenging, as we will discuss in the 'lessons learned' category. After understanding the architecture for the TM simulator, we started working on refactoring the syntax trees that we made in checkpoint 1 and 2. New variables were to be added to ArrayDec, FunctionDec, and VarDec so that they could be ready for the checkpoint 3 implementation. From there the emitters were needed so that the .tm file could be written to using the required formatting that the simulator needs. From there, we worked with a simple main function and incrementally added complexity to generate code for all the other features of cminus.

Design decisions

When making the CodeGenerator.java file we tried to follow the recommended approach as much as possible. First, all internal and simulator addresses were initialized in the CodeGenerator.java file. Next the “emit” routines were followed to maintain the code space and to generate the different kinds of assembly instructions. These instructions were written to the .tm file by changing the System output to the new file and then printing the generated code in the supported format. Since the prelude and the i/o routines were standard across all .tm files, we first implemented them. The finale was also made after these by referencing the notes on the implementation of it. At this point an empty main file could be generated by CodeGenerator.java and run by the tm simulator. From here the other features of cminus (assignment, arithmetic, control structures, functions and recursions, arrays, and nested blocks), were implemented by following the written notes.

Member Contributions

Ben Smyth: Set up checkpoint 3 repository, CodeGenerator.java, README.md, and execution script; implemented outputting of code and comments, prelude, finale, and function code.

Eli Daniels: Refactored previous code for checkpoint 3, implemented emitter methods, i/o standard code, made test files, made checkpoint 3 documentation.

Nathan McGuire: Refactored previous code, added improvements to previous checkpoints, code generation for arithmetic and assignments, control structures, arrays and parameters, functions, and runtime errors.

Improvements from Previous Checkpoints

In our checkpoint 2, we were able to get the correct output without using the “dtype” attribute for Exp.java. Since it was easier to use the dtype attribute for code generation, we added support for it in this checkpoint.

Lessons Learned

Code Generation

Due to the uniqueness of this assignment, much time was spent on understanding the simulator and with how to store/load/and manipulate the registers with the appropriate data. Understanding the uses of the register instructions took much longer than we anticipated. An iterative implementation of the guided steps given in the notes as well as using the simulator’s trace feature was useful in learning how the commands and registers worked. By taking this slow but thorough approach, we were able to produce most of the code generation’s features.

Symbol table

Since the symbol table used a similar structure to the show tree visitor method, we used a visit style that was like the visit style in show tree visitor. While implementing the symbol table, we learned that iterative additions with testing was the best way to develop it. This was because the symbol table file (SemanticAnalyzer.java) had many functions like adding variables to the table, traversing the abstract syntax tree, and checking for type errors. This iterative process allowed us to quickly recover from bugs and other errors in our code.

Type checking

The next challenge, that was also a learning opportunity, arose when implementing type checking for the C- variables. Since expressions did not carry a type with them, we needed to figure out a way to check the type of an expression to make sure that it is appropriate to use those variables. Through trial-and-error, we designed our own methods for extracting expression types based on information currently in the symbol table.

CUP file:

As it was our first time producing and maintaining a CUP file, we learned a lot about grammar specification, parsing, and error handling. The first major challenge that we encountered was the classifications of the non-terminals that cminus required. Much brainstorming and trial and error was needed to correctly classify the non-terminals to their proper classes. By following the cminus specifications as a guideline, we were able to iteratively classify all of the needed non-terminals.

Abstract Syntax Tree:

The next challenge, that was also a learning opportunity, arose when implementing the abstract syntax tree for the language. As there were many classes that were interacting with one another, it took time to understand how the tree could be set up and made to print the relevant information. Setting up the "ShowTreeVisitor.js" file to properly have the classes interact with one another also proved to have a learning curve. After getting the tree to output the simpler cminus productions (like a variable assignment) we were able to output the harder productions using what we learned.

Error handling

Throughout the process of creating messages and recovery for errors, we learned a lot about conflicts and how important they are to resolve so that the parser does not get stuck in one position. Error handling was done in a similar way to the checkpoint one and two programs. Additionally, we added a flag in the parser that tracks its error status. As we were not able to fully implement checkpoint 3, we were not able to cover all possible errors that the code generation process could give.

Assumptions and Limitations

The program assumes the input will be from a single file, a C- file (.cm), and the user will run the C- program by following the README instructions. This program does not have extensive error checking and error recovery implemented. This means that some errors can cause the parsing of a C- file to abort. The code generation currently does not support function parameters – they do not generate anything and will throw an error. A function's parameters will not be detected in any file run with the -c option.

Possible Improvements

As always, an improvement that we could implement is to have a more robust error handling system. More edge case checking and handling could also be done for a more complete program. Adding function parameters to the code generation will be the next step to having a more complete program.