CIS4650 Checkpoint 3 Documentation

Team: Christian Foote, Drew Mainprize

What has been completed this checkpoint:

During this checkpoint we started by fixing the issues that we were unable to fix in time for checkpoint 2, so that we would be able to properly begin our work on the implementation of C3.

For C3 we have implemented the functionality required for our compiler to be able to convert valid, semantically, and syntactically correct C- code into assembly that is able to be executed on the TM Simulator, as well as the flag, “-c” that is required to enable this assembly code generation.

Compiler Construction Process:

Before we even began our work on constructing the compiler, we decided to make use of git to be able to track the work that each of us had done, and to ensure that we were using the same version of the assignment in our development, so that we would avoid any issues of waiting for the other to respond with what they had been working on, or waiting on each other to send what we had finished.

The project started with Checkpoint 1, where we were tasked with creating the scanner and parser for the C- language. We began building our compiler making use of the Sample Parser that had been provided to us, with the first thing that we completed being the setup of our cminus.flex. Within this we were able to define all of the terminals that we would be making use of in cminus.cup, from simple numbers and identifiers, to booleans and comments, as well as all of the keywords and special symbols mentioned within the C- specification. Once our flex file had been completed we were able to complete our cminus.cup, once again making use of the C- specification to write the required grammar rules, define the types of the non-terminals, and define the precedence rules needed to avoid any shift/reduce errors in the grammar. Once we had defined our grammar according to the specification, we began writing our absyn class, and showtreevisitor, which work together to create and output the abstract syntax tree.

Our first task for checkpoint 2, was to fix any issues we had missed in our submission of checkpoint 1, to allow us to know that whenever we came across an error, it was due to something from checkpoint 2, and we wouldn't have to look back to find a bug we missed. Once this was done, we were able to begin in earnest on what was required for checkpoint 2, beginning with creating a symbol table using layered hash tables that allowed our compiler to easily, and cleanly, add and remove scopes, while also tracking any symbols that existed within that scope. After completing the symbol table we began work on the semantic analyzer, as well as a symbol class to accompany it. With this semantic analyzer our compiler was now able to work with the symbol table to perform type-checking, ensuring that any variable is declared before it is used, return types match function types, and other similar kinds of checks, ensuring semantic validity.

We finished up the compiler with checkpoint 3. We started by importing the TMSimulator code that was provided for us. We tested that with the various test files provided to investigate how it ran and see some sample output that we could use in our own code. Next, we created the CodeGenerator class which handles all assembly code generation and code processing. It includes a constructor to initialize the object and begin running the program. We needed to implement multiple constants for the register pointers and instruction locations that are updated and accessed at various points in the program to assure consistency and ease of use through the code. It contains an overloaded ‘visit’ function that handles the code generation process for every object in the CMinus language and generates assembly code for it.

Lessons learned:

Throughout the time we spent on this project we learnt a few new lessons, the importance of actually learning from the mistakes you’d made earlier, as well as having a few old lessons reinforced. One of the first lessons we learnt was the importance of incremental development, and time management. When we started on this assignment we were lacking in time management, and from this decided to ignore the incremental development steps that had been laid out for us, leading to even worse issues with the time we had, due to issues that would not have surfaced had we followed the steps given to us, creating a negative feedback loop we were never really able to break free from.

One of the most important tools to help mitigate these issues, however, was using git, as with this we often didn't need to communicate directly to know what the other had been working on, and what sort of progress had been made, as well as if there were any conflicts between what you had written, and what was recently uploaded by your partner.

One of the final lessons we learnt was how helpful, and important it was to stick to a consistent style throughout the assignment. Due to the nature of how work is done in the vast majority of cases we've both developed our own styles, ways we organize our code, and even our own way of naming variables. Leading to more than a few issues where there were several places we had to track down and rename or reorganize code because what was written by one person couldn't be run with what was written by the other.

Assumptions & Limitations:

We assume that OpExp consists of both bools, or both ints, using a 1 or 0 as a stand in for true and false respectively is not implemented.

Due to the limited time that we had available to work on this assignment, we consistently found ourselves lacking the time needed to implement error detection and recovery, and so we've found ourselves needing to limit any of our testing to c-programs that we know are semantically and syntactically correct, and any programs that are not both semantically and syntactically correct will either fail to run, or produce unexpected output.

Potential Improvements:

The first step towards future improvements of our compiler would be the implementation of the error checking that was mentioned above, as once this is completed, we would have a much simpler time finding any areas which could be improved, or edge cases that we missed which for the moment we have no option but to walk through manually to try to find any errors, using much more time than we should to fix any bugs.

Once this is done it would be beneficial for us to go through and clean up a lot of the code that makes our compiler, as, again due to time constraints, we often found ourselves lacking time needed to write proper, clean code, leading to later issues taking much longer than they should have to fix due to difficulties deciphering our own code. What has often felt like putting together a puzzle when looking back at what we'd written should be much more accessible, and that initial time investment should have been made for the sake of saving time in the future.

Individual Contributions:

Drew:

- Completed TM
- Implemented code generation

Christian:

- Finished incomplete SemanticAnalyzer and SymbolTable
- Wrote documentation