Checkpoint 3 Documentation

What we've done

For this checkpoint we had to implement conversion from a syntax tree into assembly code for the C- programming language. The first step in our development process was to read the assignment documents, make appropriate changes to our Checkpoint 2 code, and familiarize ourselves with the TM assembly code structure and rules.

The next step was to read the lecture slides in order to understand what kind of output we were expected to produce, as well as get a feel for how to start the development of our assignment. This took a lot of time because the TM assembly code is quite dense to understand. We decided to take a break from this, and get the functionality for the Prelude and Finale working because they are mostly the same for every TM file. Because of this, we put the prelude in our main statically. Then we implemented the -c command line flag. After we were done with this code, we tested it with the same command we used to test our Warmup Assignment scanners, and included the -c flag. This was done to ensure that our prelude and finale were working properly.

Next we decided to start trying to understand the TM assembly code again. We started by looking at the example .tm files and comparing them to their .cm source code, to get an idea of how different structures of the code would be generated in assembly. We did this while also going through the slides to find any information that

could be useful, like the syntax of each line in the TM assembly language. During this process we also went through our notes from our Microcomputers course from second year which was very focused on working in Assembly code. This helped us remember some terms and key ideas of assembly code which we had forgotten.

Eventually we realized that we were having a very hard time understanding the TM assembly language, and we would not have time to produce a fully functional product with the assembly code implemented. It was at this point that we decided to scrap what we had, and make sure that our project still worked 100% for the previous 2 checkpoints. This was a tough decision but we felt it set us up for the best opportunity to at least get some marks on this assignment.

Next we got to work on creating our 1234567890 test files. We wanted to include as many different errors as possible because we had spent a lot of time making sure the type checking included a lot of edge cases. We started with simple errors like wrong variable types for the operator being used, and worked our way up to including more complicated errors in our files.

Issues Encountered During the Whole Project

One of the main issues we encountered right away were the learning curves for each checkpoint. For the first checkpoint we learned right away that we would need to dedicate a lot of time to future assignments to give ourselves time to really grasp the

material before starting. Starting early would also give us time to ask questions in class if we ever got really stuck at a certain part.

Early on we also faced the issue of error handling, and trying to get our program to recover from errors. Neither of us are really great at making robust error handling for projects like this, but this project gave us a chance to improve that aspect of our coding skills. By the end of this project we were becoming very good at identifying where possible errors could pop up before they happened, and adding handling to make sure the program still ran to completion after these errors occurred.

Design Process/Techniques

We worked on this assignment in person, using VS Code Live Share so we could work on the same files at the same time. We split up most of the work in half, and worked on our halves simultaneously so that we could ask each other questions if needed, but also keep the work moving quickly and efficiently. This helped us answer each other's questions, and greatly improved our learning because we both had a chance to do all of the different steps required to implement the project. We also had many monitors and kept the lecture slides/other reference material open on these monitors so that we could always look at it if we needed to confirm something/get a quick refresher on what we needed to do.

Lessons Learned

- Dedicating time to understanding the lecture slides before working was imperative to grasping the assignment
- Learn, code, repeat is a recipe for success

- We could've done a bit more planning in order to avoid refactoring
    - Some big chunks of code were repeated and then made into functions later
    - Incrementing level in CompoundExp was much cleaner and more effective than doing it in each of FunctionDec, IfExp, WhileExp
- Assembly code is hard to understand
- We learned to have a lot of respect for people who make compilers for a living because it is very hard work, but someone needs to do it

Assumptions/Limitations

- Assumes "if" and "while" cannot be names of functions
    - this would cause issues with if and while statements anyway so we feel it's an acceptable limitation
- Assembly code is not generated for .cm files except for the prelude and finale

Future Improvements

- Find more edge cases to add to error handling
- Code could be cleaner
    - a few blocks of code would be better if they were functions
    - commenting would help with a file as big as SemanticAnalyzer.java
- Actually have the main body of assembly code be generated for .cm files
- Have a menu that the user can use to select what options they want instead of using command line args

Collaboration Breakdown

- We (Michael Harris and Zachary Schwering) worked side-by-side from start to finish on this project
- Every step in the project from reading slides to writing code to testing was done in equal parts by both group members