Checkpoint 2 Documentation

What we've done

For this checkpoint we had to implement a Semantic Analyzer and Type-Checking for the C- programming language. The first step in our development process was to read the assignment documents, make appropriate changes to our Checkpoint 1 code, and familiarize ourselves with the C- language specification rules for semantics.

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. After this was done we copied our showTreeVisitor.java file and renamed it as SemanticAnalyzer.java. This is where we put the code to print out the Semantic table to a .sym file, as well as where we wrote the code to type check and print the errors out. We started with doing the code for getting the semantic table to print properly. After we were done with the Semantic scanning code, we tested it with the same command we used to test our Warmup Assignment scanners, and included the -s flag. This was done to ensure that our C- semantic table was actually outputting properly.

Next we decided to start working on type-checking. We started by looking at the CFG in our CUP file, to get an idea of what places in the CFG would lead to expressions that need to be type-checked. We then worked together using pair programming to come up with the logic for each visit function that needed to have type

checking. During this process we also set up type checking functions that we could use to get the type of any of our data structures. This went by fairly quickly, as a lot of the type functions just had to call other type functions.

After this we were ready to tackle error checking our semantic analysis. This took quite a while because we wanted to make sure we had as many edge cases covered as possible.The error checking also helped us find some flaws in our original Checkpoint 1 implementation that we then fixed. During this time, we also realized we were lacking the functionality to have blocks of code besides if, while, and function block. This forced us to go back and change the implementation of our semantic analysis to allow for blocks that were only designated with { } by themselves.

The next step was to set up the code that would check that function definitions match their corresponding function prototypes. This included making sure that each prototype had a definition later in the code, as well as making sure the parameter types and number of params matched. This led us directly into making sure that each function call had the right arguments for the function it was calling, as well as making sure that every function had a return statement that matched the return type of the functions. This section of our project took a very long time because there were a lot of different types of objects and functions that needed to work in unison to make sure these checks were accurate. We worked together on this, thinking about the solution together and then pair programming it.

We then decided to revamp our error messages to make them more descriptive and include line numbers. We also found some new requirements for our project as we were going through the lecture slides, like making sure that void main(void) is the last function defined in the code, and adding the input/output functions that are described in the lecture slides to our hashmap. We also went back and changed some repeated large sections of code into functions, to make our code more readable when we start Checkpoint 3.

Next we got to work on creating our 12345 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.

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.

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

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

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

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