Checkpoint 1 Documentation

What we've done

For this checkpoint we had to implement a scanner, parser, and error recovery for the C- programming language. The first step in our development process was to read the assignment documents, set up our programming environment, and familiarize ourselves with the C- language specification.

The next step was to test the given tiny language files 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 made a copy of the tiny.flex file provided and renamed it as M1.flex. This is where we put the regular expressions to handle the different tokens of the C- language. After we were done with the M1.flex file, we tested it with the same command we used to test our Warmup Assignment scanners. This was done to ensure that our C- scanner was actually scanning the proper tokens in the right order, and ignoring the ones we didn't need.

Next we decided to start working on adding files to the absyn folder of the project. We started by looking at the files already provided in the folder, to get an idea of how to write the files for the rest of the classes listed in lecture 6 slide 7. We then divided the needed classes in half and each worked on the half we chose, while asking each other for help if needed. After this was done we went about implementing the visit functions in ShowTreeVisitor.java, and AbsynVisitor.java. This went by fairly quickly, as a lot of the visit functions were very similar to each other, and it helped that we had examples from the tiny language that were provided to help us.

After this we were ready to tackle the implementation of the CFG in our CUP file. This took quite a while, and served as the first major roadblock for us. It was a bit of a learning curve to understand the syntax for CUP files and grammars, so we started by just implementing a shell CFG with all the rules but no code. Then we tested this a few times and got some errors that were preventing our CFG from running successfully. We eventually realized that the errors were due to the fact that we had not defined our terminals and non-terminals in the CUP file. After we declared them, our CFG was able to run successfully with no output.

The next step was to set up the code within our CFG to allow for output of the Abstract Snytax Tree. This was a bit of a long process as well, because we had to understand which classes from absyn should be used to implement each part of the CFG. We worked together on this, splitting the CFGs in half and each working on half. In some cases if one of us had trouble with a specific rule in the CFG we would ask the other person for their thoughts, or just leave it for later. Anything that was left for later was worked on by us at the same time so that we could pool our ideas for how best to implement that rule. After implementing all the rules we hit another roadblock. We were getting a lot of NullVariable Exceptions in our code, stemming from our ShowTreeVisitor.java file. After spending a few hours debugging we decided to add null checks to each visitor function that called an accept function. This solved our problem immediately, and we were able to see our compiler run and output the Abstract Syntax Tree.

We then noticed that the output was not printing variable names, and was instead printing the whole java object for the variables. To fix this we changed the function for VarExp in ShowTreeVisitor.java so that it would route the code to the specific SimpleVar or IndexVar functions which had access to the name of the variable. We also made a change to the NameTy visitor function so that it would print the words BOOL, INT, or VOID instead of the ints associated with those values in the code.

Next we had to figure out how to do error handling. We worked on the errors one at a time to make sure they were getting caught the way we expected. We started with simple errors like wrong variable types (clint instead of int for example), and worked our way up to more complicated errors like in function prototypes, functions parameters, and errors within expressions. Around this time we also learned that we were supposed to output the AST to a file, not just to STDOUT, so we had to take a break from the errors to figure out how to write to a file instead. After this we went back to trying to add error handling into our CFG, with as many varieties of errors as we could. Lastly, we started creating our 5 test files for our demo, which was difficult because we had to decide which errors to include and which to omit.

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

- Working with abstract syntax trees in a hands-on manner provided a much deeper understanding of ASTs and how useful they are to compilers.
- We started to run out of time with error handling, and it became hard to know how many more errors we should add support for.
- We learned that we should start earlier on Checkpoint 2

Assumptions/Limitations

- The way our regular expression to detect comments is built means that any file that has '/' or '*' inside of a comment will not work with our program
- We assume that functions that are non-void don't necessarily have a return statement

Future Improvements

- Change the comment regex to allow for '*' and '/' inside of comments
- Improve readability of tree printing output
- Add more error handling

Collaboration Breakdown

- We (Michael Harris and Zachary Schwering) worked side-by-side from start to finish on this project. Most things were done together and in equal parts.

- While Zack was working on adding the necessary tokens to the M1.flex file, Michael was adding the terminal and non-terminal symbols to the tinyBare.cup.
- We each created half of the classes in the absyn folder.
- We both created half of the CFG rules.
- We both wrote half of the error recovery cases.
- We both declared half of the visit functions in AbsynVisitor.java
- Michael wrote most of the Error handling
- Zack wrote most of the documentation
- Michael did all the driving
- Zack ordered the food