Checkpoint 1 Report

Lance Paje, Pasan Undugodage

Related Techniques and Design Process

For the tiny.flex file we largely copied the structure of the flex file in the warmup assignment as we found no need to deviate from a working structure. The keywords, symbols and special tokens were pulled from the C- specification document and made into symbols.

For the tiny.cup file we translated the production rules from the C- specification document. We retroactively added the non terminal symbols as we wrote them from the C- specification document. We appropriately typed the non-terminals according to the lecture notes “6-SyntaxTrees”, 7th slide, using the closest parent class as the type. With the exceptions of NameTy and the List classes, as those use their own typings. Further changes to the typings of terminals were made according to the requirements of the classes called in the embedded code.

For the java files in the absyn folder, we made the required files according to the lecture notes “6-SyntaxTrees”, 7th slide. We copied the general structure of the classes from the pre existing java files that were there. Not much thought went into the design of these files as these were largely boilerplate code. At times we changed the parameter types of the classes in order for things to work.

ShowTreeVisitor.java mainly contained boilerplate code as well, copying code from existing classes. The two classes of note were the visit classes for OpExp and NameTy, as those classes required case switch statements.

Lessons Gained

With the amount of boilerplate code this checkpoint has, we made many small changes to our code as the project progressed. This was largely due to modifying code in one area to fix a problem and not modifying code in another area to match the recently changed code. This led to a lot of backtracking throughout the project and even led to reopening files that we thought we were finished with. Moving forward we would like to be able to test our components individually from one another to ascertain that it is completely finished before moving onto the next component.

Another lesson we had learned was that due to the lack of feedback in terms of errors for the program, it took us a long time to sort out the errors. Plenty of time was spent on scouring java files and trying to mentally connect how the program's files interacted with each other. We also learned at this point that there were plenty of classes that were largely mirroring each other in function, with only slight differences in parameters. It would be nice to sum up all of these groups of classes into a single file to help minimize the backtracking we had to do. The classes overall would become more complex, but this would be a preferred tradeoff to the boilerplate code we had to do.

Finally, we learned that the two of us had vastly different working schedules. Lance Primarily worked during the day while Pasan worked throughout the night. This made collaboration efforts sparse and short until closer to the deadline where the both of us decided to extend our work schedules to allow for more overlap.

Assumptions and limitations

The biggest limitation we both came across was the lack of testing we were able to do since we typically use compiler errors to tell us what's wrong with our code. The program when compiled also didn't output useful errors when parsing through a file, in most cases returning an empty array of symbols that it was expecting. This in turn prevented us from fixing the problems immediately. Due to the lack of an informative error output, that made every single file a possible source of error.

A related limitation to the first one is the difficulty of tracing code through the program. To elaborate, we had a problem with trying to figure out what class or method was being run when an error occurs. This is different from the first limitation as this was more so an issue with the general file structure not making it evident what class was causing the error. We had to cross reference between the tiny.cup file, tiny.flex file, ShowTreeVisitor.java file and the file we were trying to compile.

For this project we assume that all files passed through the parser will be of the type .cm and that they would contain C- code at the very least. Something like python code would likely create unexpected outcomes in .cm files. We are also assuming that we are only running one C- file in and not a group of files through the parser simultaneously. At this point it doesn't really matter if a.cm file contains more than one function as the parser will simply list out the tokens in the tree. Another assumption being made is that a lot of design elements and components are supposed to mimic that of the sample package we are given. Because of that, the output abstract parse tree appears similar to that of the sample package. We also make the assumption that if we run into a statement with an error, we skip over it and any subsequent elements belonging to that statement.

Possible Improvements

One possible improvement we could make to our code is to implement more precedence rules to remove ambiguity in our production. Currently our code follows the production rules found in the C-specification document closely. With the creation of precedence rules we could remove ambiguity from the mathematical operations plus and minus, and multiply and divide. Additionally by implementing the precedence rules in the cup file we could prevent any possible infinite loops.

Another improvement we could make is to add more error checking. As of right now we are implementing a few lexical errors into our error recovery. We currently check for symbols after semicolons, missing expected semicolons, variables that haven't been declared and missing the number of curly braces not matching. Additionally we could create error checking that makes warnings instead of full errors for things like empty bodies in compound statements, while loops and if/else statements.

Further improvements can be made in the ShowTreeVisitor.java file. While it currently meets the criteria for this checkpoint, we would like it to show the elements in order and for some classes to show more detail. As of the time of writing OpExp only outputs the operator and not the full expression on one line. It also prints the left hand side and right hand side of the expression beneath it. While not unreadable it does not make it obvious what's going on. ShowTreeVisitor.java also does not show array variables such as array[i] and we would like to see this implemented at a later date.

Distribution of Work

Work was largely done by separate people at first, focusing on the individual files of the Checkpoint while later on in development both members collaborated on bug fixes on the same files. To begin with, the tiny.flex file, the production rules in the tiny.cup file, most of the absyn files and ShowTreeVisitor files were created and modified by Lance. While the error recovery and embedded code in the tiny.cup file, and some of the absyn files were done by Pasan. However since we both encountered many bugs during development we made many modifications to most files and cannot claim full credit on any given file alone.