CIS*4650 W24 Checkpoint 1 Project Report

Salman Azhar (1058981)

Overview:

In this report, I will be going over the process of how I finished this checkpoint, going through the preparation, then the Lexer, Parser and finally the error checking. Following this, I will provide some reflections on my process.

Implementation Process:

Preparation:

In order to get started on this project, I went through the following steps:

1. Read the checkpoint 1 description posted on Courselink and downloaded the starter code provided by Professor Fei Song
2. Download the necessary dependencies, including CUP
3. Run the given Tiny parser and see the format of the AST
4. Go through the files, starting from Main.java and see how the starter code implements this parser

a. Here I learned the role of the Absyn package, and how the Lexer is connected to the Parser through the sym.java file generated by CUP.
b. Also saw the format of embedding code in the CUP file, and what error handling possibly looks like.

Lexer:

To tokenize the input files, I needed to build a Jflex file. This was a fairly straightforward process, since the expected tokens were laid out in the C- specification file. These are the steps I followed:

1. Read through the C- specification file to see what kind of tokens I would need to expect from a .cm file.
2. Built the Jflex file with the token mentioned, using regular expressions for the numbers, ID’s, whitespace, comments and errors, but using literals for everything else.

Parser:

As expected, building the parser itself was the most challenging and time consuming part of the process. These are the steps I followed:

1. Typed out the grammar laid out in the C- specification file
2. Declared the various terminals and nonterminals, along with the precedence rules.
3. Simplified the grammar from the expression rule onwards, and combined the binary operators and the singular expressions all under the expression rule.

4. Referenced the lecture slides for “Lecture 6 - Syntax Trees” to find the classes that ought to be included in the Absyn package, and added them.
5. Populated the newly created files with code, following the format provided in the starter codes Absyn package.
6. Added the embedded code to the Parser file, and all the “RESULT” assignments.
7. Changed the ShowTreeVisitor.java file to incorporate all the new Absyn package files, and printed all the properties with their names. I simply followed the example Tiny AST format.
8. Filtered through what was being printed and figured out what nodes were important to be printed and not . For e.g VarExp was redundant to print, since it’s only job was to be a carrier for the Variable object.
9. Tested on the 5 given .cm files.

Error checking:

Although the parser was the bulkiest portion of this checkpoint, the error checking was the most complicated to get right and understand. I had to look through various documentation online to gather information about the dedicated “error” token in CUP. I got an idea from the example Tiny parser files on how to go about implementing some basic errors, then I tweaked them around and through a lot of trial and error gained some intuitive understanding of how it works. Essentially what I learnt was that the production rule with an error token is triggered when the other rules along with it in the same ::= statement are not found. So following this foundation, I was able to get some good error reporting and recovery going. However, I had to make my own error reporting function called lil_error, since the given report_error function was giving me trouble. By the end, my code could report and recover from a wide array of complex errors including missing symbols, and invalid formats for non-terminals.

Reflection:

Assumptions, Limitations and Possible Improvements:

My primary assumption for this checkpoint was that the input code would follow the C-specifications given in the doc. Now obviously, the point of implementing error recovery was to deal with situations where that was not the case. I believe I was able to cover a great deal of errors, hopefully enough to meet these checkpoints requirements. However, I don't believe I used CUP's error recovery potential fully. Such as my code not handling missing curly braces ('{ }') in statements. Partly due to not fully understanding it, and also because I tried not to deviate from the grammar laid out in the spec file. My error reports can also seem somewhat messy due to CUP's internal reporting for the error token, which I couldn't seem to override and just printed my report next to it. Although, I do believe I made it seem as clean as possible, I would have liked to make my error recovery cleaner and comprehensive. Also, I think I could have simplified the grammar further with the statements.