Names: Kaylee Bigelow & Jonah Stegman

Date: March 8, 2021

Documentation

What has been done

As a team throughout this checkpoint we worked together to create a scanner, parser, and parse tree for the language of C-. This process began with a design discussion to verify that both of us were on the same page concerning how we would approach the assignment. Next we went through the sample code that was provided to understand how the parser dealt with the tokens as well as how it interacted with the parse tree.

The contributions that each member made is difficult to put a border on. For much of this assignment we worked together while pair programming. We found that working together best ensured that we were on the same page.

When implementing the scanner we reviewed the C- specifications document and wrote down all the different possible tokens that could exist. Then we went to start writing the scanner like we did for the warm up assignment. Quickly we realised that the scanner would need much less regex as most cases of tokens were very specific. Once all the tokens and their conditions were added into the .flex, and the token names were added to the .cup file; we ran just the scanner on each of the 3 cm files given. This was to test and see that all the tokens generated were accurate and nothing was missed.

Next the parser was implemented. To do this we broke up what needed to be done for the parser and worked on one piece at a time. First was translating the BNF to

the format used by cup. From there after gaining an understanding of how the rules interacted with one another we began creating the return vales and the corresponding classes. Once the classes were created this began the parse tree process.

To implement the parse tree we had to discuss how it would look including the levels and what the naming convention would be. This process was the main work in generating the parse tree. After some trial and error we reached a point where the parse tree looked how we wanted, which left us with only one path left, error handling.

Error handling was a lot of trial and error. We found that the process that worked best was to think through each rule and then add error handling as seemed fit. This allowed us to verify that each case was working before moving on to the next case.

After all the parts were implemented we ran through extensive case checking based on files that we created, contained good code and code with errors. Overall, the process we used allowed us to understand each step and verify all parts as we went along.

Lessons gained

As this process went on there were a few lessons we gained. The first was learning how in depth a parser is. While we have been using parsers for years it is difficult to realize how complex they can be. This assignment helped us to learn how a parser works and appreciate them in our day to day.

The second lesson that we had learned in the past but was enforced, was to work incrementally. When adding the error checking it became apparent very quickly that it had to be done one at a time. Many classes have taught us to test as you go,

however we as students often try to save time by combining steps. This project showed the importance of testing as you go, so you do not end up with 16 errors and no idea what is causing any of them.

The third lesson that we learned was how precise you have to be when parsing inputs. Even with our tokens being exhaustive we still had times where our grammar would fail because it went into the wrong path. It taught the importance of mapping out the entire flow of the code and each path it could/should take before writing it. It allowed us to see that preplanning not only saves a lot of headaches later but it also allows you to better understand what you are coding and what might possibly go wrong.

Assumptions and Limitations

There were a few assumptions that we made working on this assignment. The first was that the user would understand the variables presented in the parse tree. While the variable names that we chose are understandable (eg. ParListExp is used to represent a parameter list), we assume that the user does not need the full name displayed in the tree.

During error checking we made a few different assumptions concerning what should be considered an error and what type of error it should display to the user. To make these assumptions we pulled from our experience using compilers in the past to display error messages that will help the user.

A final assumption we made was that the parse tree should show errors as place holders. This was decided so that the user can still see the layout and all of the elements, instead of removing everything related to the error. One example of this is if

during a function declaration the parameters cause an error. In this case the program will still output the function and the internal information, along with an error about the parameters.

There are a few limitations that exist in our project, the first being not handling all error cases. While we were able to catch most error cases, in this short of time frame we cannot say that we got every error case that can occur. This leaves space for future improvements to the project that are outlined in the section below.

Another limitation of our project is all local declarations in a function must be made before a statement. This is a limitation created from the given grammar for C-. To fix this limitation the grammar would need to be changed and updated.

Improvements

There are a few ways to improve upon what was accomplished for this assignment. The first would be to make the overall error handling more verbose. While we implemented fairly extensive error checking there are more edge cases that we could check with more time. Some examples of these would be checking for missing brackets and missing semicolons. Another example of an error that we could check in the future would be things like [] or {} not attached to any object or function.

Another way to improve upon this assignment is to print the tree in an easier to understand format. This would include using ASCII characters to create a more tree like structure to display to the user, thus improving readability.