Checkpoint 1 Report

Wafa Qazi(0932477) & Aqsa Pehlvi(1018401)

CIS*4650

Author Note

Checkpoint 1 Report

Introduction

This report highlights the related techniques used and design process for our implementation of checkpoint 1, for our semester wide compiler project. For this checkpoint we implemented a scanner that generates a sequence of tokens for the C- language, a parser to generate an abstract syntax tree and error recovery that reports errors and continues the parsing process.

Techniques

For this checkpoint, our main task was to implement a parser using CUP to generate and show an abstract syntax tree. We first began this checkpoint by implementing a scanner to scan for our different tokens we were given according to the C- specification in the assignment outline. In order to connect the scanner with the parser we used the JFlex tool to scan each line of the input file. Once we had finished building our scanner and parsing our individual tokens, we then added the given grammar rules in our .cup file. Once we had the grammar rules in the file, our first step to creating the tree was to simplify these grammars. This was done in order to make implementation of our syntax tree simpler. After simplifying the grammars by doing derivation and removing unnecessary grammar rules, we began to add precedence ruling to the working grammars to help reduce conflicts and ambiguities.Then we began to add the embedded code to our rules to start producing the syntax tree. We began implementing the grammar rules that were non-recursive first and then worked our way up to grammar rules that used recursion and called other grammar rules. After the grammar rules were complete, we created different print functions in ShowTreeVsitor.java to print each related class that represented a node in the tree. After we had our tree complete and ready to print, we began to test our code by using small files that tested only one case at a time. After we tested and verified that individual cases of our tree were correct, we then went ahead and tested with the .cm files provided to us. After we were certain that our tree was printing the correct tokens with correct ordering/indentation, we moved on to error recovery. Some of the error recovery we implemented was verifying expression errors and minor syntax errors.

Design process

In terms of the design process, we began by rewatching/rereading the class lectures to better understand how to use the tools required for this assignment. After we gained a more in-depth understanding of grammar rules and syntax trees, we were then able to visualize how our syntax tree would work. Once we had the information needed to start coding, we decided to do a bottom-up approach, as recommended in the outline. This means we began by simplifying the grammar rules given to us, and then implemented each rule based on its dependencies of other rules. After that we began building on the given tree structure classes for our C- language, which was the first part of the process of designing our syntax tree output. During the design process we ran into some issues, specifically with implementing cases with epsilon. We ended up finding a solution to these issues with trial and error, as well as referring to the sample parser given to us.

Checkpoint 1 Report

Summarizing Lessons

In this assignment, there were many valuable lessons we learned. The primary lessons we gained from implementing this parser were how to use JFlex(more in-depth then the warm up assignment) as well as how to use the CUP tool. We also learned about many different techniques used within parser tools such as CUP, including adding grammar rules in BCNF, error recovery and techniques to resolve ambiguities such as precedence directives. By completing Checkpoint 1 we also gained insight on how to apply the theoretical concepts learned in class; such as context-free grammars, parsing techniques like left-most derivation, and creating syntax trees. We also gained an understanding of how we can use the visitor pattern, as described in class, to help with creating our syntax tree.

Assumptions and Limitations

While implementing our parser, there were a few aspects that we were led to assume while implementing the code. One major assumption we had to make was in regards to the epsilon cases in the parser. We assumed that when a case that had epsilon was invoked, an empty list should be created. Another assumption we made was related to error recovery. When coming across a syntax error we either created a dummy node or passed through "null" to allow our parser to continue parsing the tree and not crash. In terms of limitations, the primary limitations for our program are the amount of errors our program can catch. Due to time restrictions, we were not able to complete extensive error recovery.

Checkpoint 1 Report

Possible Improvements

The main improvement we could have made to our program was to implement more in-depth error recovery and error-handling. Aside from this, we believe to have satisfied all the other requirements of the assignment.

Contributions

Since we worked in a pair, we initially attempted to assign specific tasks to each other. However, this technique was not very effective as we found that working at the same time actually made us more efficient. This way, we were able to catch each other's mistakes and share ideas easily while coding. Wafa individually contributed implementing the scanner portion of the assignment. After the scanner was implemented, we both contributed equally by working simultaneously on the rest of the assignment.

Conclusion

In conclusion, the techniques and lessons gained from implementing Checkpoint 1 have given us a solid foundation to start Checkpoint 2. We have gained invaluable theoretical as well as coding skills from this assignment and believe we are in good shape for the next phase of this compiler.