Compilers
Checkpoint One Report

BY

Braydon Johnson

Neivin Mathew

March 6, 2017

Contents

1 Summary

For the Checkpoint One, we implemented scanning, and parsing for the C- language. The scanner can recognize valid tokens according to the C- specification and the parser will analyze the syntactic correctness of the programs. The compiler builds an Abstract Syntax Tree and can print out the structure of the program.

The compiler can also recover from syntax errors by consuming invalid input until it reaches a valid structure according to the specification. It will try to output the structure which caused the error in most cases, or at least mention the line number of the error.

2 Design Process

2.1 Implementation

The compiler project for Checkpoint One was built in an incremental process.

To build the scanning and parsing portions of the compiler, we began by analyzing the C- specification document. First, we outlined the scanner to recognize tokens for the language, then we listed out all the grammar rules from the specification exactly how they were outlined, without simplifying the unambiguous grammars.

Next, we followed the recommended syntax tree structures for the C- language to create the necessary classes required to represent the syntax tree. After writing all the required classes, we defined the methods to display the syntax tree.

With the structure of the tree completed, we added the embedded code into the parser to actually build the tree. After the tree was able to build successfully, we introduced three new classes that would consume errors for their corresponding types. These enabled the parser to recover from syntax errors gracefully.

Finally, we simplified some of the grammar rules by using the directives for precedence and associativities of mathematical operations.

2.2 Lessons Learned

We really understood the importance of building the project incrementally, since it allowed us to break down a large problem into multiple smaller problems. It also allowed us to pinpoint where our errors were, since we knew the project was building successfully before adding new functionality.

Using Git for version control and collaboration also taught us the importance of version control, which gave us the ability to roll back mistakes we made and enabled other group members to see what new parts had been added to the project since we last saw it.

3 Assumptions and Limitations

Some of the assumptions and limitations of the project are as follows:

- The scanner simply uses a regular expression to recognize comment blocks, rather than macro states. This could result in unexpected behaviour when comment blocks are nested within each other.
- The parser will try to consume invalid input until it find the next valid block. As such, it will display the location of the error by line number and abstract structure.

4 Potential Improvements

Some possible improvements that could me made are as follows:

- Using macro states to recognize comment blocks from normal blocks could be a more robust way to recognize comments.
- Errors could display more detailed messages about column number at lower levels of the abstract list hierarchy.

5 Contributions

5.1 Braydon Johnson

- Writing Scanner regular expressions
- Writing Parser grammar rules
- Creating test files

5.2 Neivin Mathew

- Writing Parser grammar rules
- Building syntax tree classes, display methods and main program
- Writing documentation for the project

6 Acknowledgments

For this project, we used some of Professor Song's starter code from `java_tiny.tgz` as well as followed the recommended syntax tree structure for the C- language from the course slides.