Checkpoint 1: Lexical and Syntactic Analysis
Project Report
CIS 4650* Compilers
March 9th, 2022

Umer Ahmed
Numan Mir

Introduction

The objective of CIS*4650's course project is to build a compiler for the C- (C minus) language. Creating this compiler will require building several connected items: a scanner for lexical analysis, a parser for syntactic analysis, a semantic analyzer, and an assembly code translator/code generator. This process will be executed incrementally over a series of three checkpoints.

For the first checkpoint of the course project, we have built a scanner and parser to handle lexical and syntactic analysis for the C- language. This consists of 1) a scanner specification JFlex file that scans all tokens of a C- file and reports lexical errors, 2) a CUP program that defines C- grammar rules, enforces syntactic error recovery, and produces abstract syntax trees, and 3) a set of classes defining tree nodes and a corresponding Java program (ShowTreeVisitor.java) that uses a visitor pattern to display the aforementioned abstract syntax tree. Our project is now ready for the next step in building a compiler: semantic analysis.

The rest of this report will outline the process of building the scanner and parser for the C- language. In particular, we will discuss our design strategies, insights from applying course knowledge, testing methodology, assumptions, limitations, and potential improvements.

Design Process

We implemented the requirements of this checkpoint in an incremental fashion, starting off with the scanner. Using the sample scanner provided by the professor, we were able to quickly produce a working scanner that connected to a primitive version of our CUP program. Firstly, in the scanner (cm.flex), we defined the appropriate regular expressions and keywords required for tokens in the C- language and created corresponding actions for each one. We then created cm.cup.bare, the first iteration of our CUP program, which laid out the terminals and

non-terminals involved in the C- grammar rules, and connected this to the scanner. At this point, we could compile and run the scanner to see that it was displaying tokens as expected.

We were now ready to implement the second iteration of the CUP program, cm.cup.rules, which includes grammar rules for the C- language. The completion of this program enabled us to check whether a C- file follows the defined rules. The next step in our design process involved creating class files (in the absyn folder) that describe the non-terminals in our CUP program and act as tree nodes. We used these class names as types for the non-terminals in cm.cup.layered, and used the class' constructors in the embedded actions for our syntax rules to build appropriate tree nodes.

We built the abstract syntax tree incrementally, starting off with simple rules and building our way up to more complex rules. We created simple, partial C- programs that exercised only one or two rules at a time to aid us in the development of the trees. Simultaneously, we added methods to our tree visitor program (ShowTreeVisitor.java), so our compiler would display the nodes as it iterated through them. Eventually, we worked our way up to the largest rule that encompasses an entire C- program, and were able to test full programs to see our abstract syntax trees being produced correctly. To further improve our CUP program, we created cm.cup.ordered to simplify the grammar rules by removing layering and introducing precedence directives. This resulted in a cleaner, more concise parser program.

Error recovery was a delicate process. To avoid introducing grammar conflicts we ensured that only one error scenario would be worked on at a time. To begin, we wrote short C- programs with simple errors and created corresponding conditions in our CUP program which display the type of error and let the parser continue. We created error recovery conditions for the major components types specified in the assignment outline; declaration lists, expression lists,

and expressions, including their subtypes. In our submission, we included C- programs: ‘2’, ‘3’, ‘4’, ‘5’ (all ending with .cm) which provide and describe various types of parsing errors to showcase the thoroughness of our error recovery implementation.

Course Knowledge Exercised

With the completion of this checkpoint, we worked with DFAs via our JFlex scanner, and a LALR(1) parser via our CUP program, including shift/reduce error-handling. By working on this assignment, we were able to deepen our understanding of the interaction between the scanner and the parser components in the “front-end” of a compiler. More specifically, working on the lexical and syntactic analysis of a simple language like C- allowed us to clearly see how a scanner reads an input stream of characters, tokenizes the stream, and provides these tokens to a parser, which then validates the syntax and produces an appropriate syntax tree. With this assignment we also reinforced ideas of how context-free grammars (in Backus Normal Form) and precedence definitions are applied in the setting of a compiler. Furthermore, through the process of resolving conflicts resulting from error recovery rules that we introduced, we exercised our knowledge about the importance of unambiguous grammar rules.

Assumptions, Limitations and Improvements

Assumptions:

- Given the simplistic specification of the C- language, we made the assumption that syntax involving the unary minus operator will not be tested. For instance, x = -2 + 3.
- In the case of running our program on a file with syntax errors, the abstract syntax tree displayed will not include the components with the errors. For example, if a function has an invalid name, the function and its children will not be displayed in the tree.

- We chose to implement a fatal error in the case where a function declaration has invalid parameters. For example, an empty params “()” instead of (void) or a single invalid param such as “(var;)” as its parameter.

Limitations and potential improvements:

- Error recovery conditions were not exhaustive: We would have liked to implement recovery for syntax including multiple sets of parentheses, to ensure there is balance for each pair.
- In some cases our error output has inconsistent line spacing/ordering. For example, errors in the body of an if-statement will be logged before errors in the test condition. This could be confusing for users expecting a linear report of errors.

Member Contributions

For the majority of our first checkpoint, we worked closely together via Microsoft’s Live Share extension for VS Code, in which one member gains remote access to the other’s terminal and workspace; this way, both members can see each other's changes in real-time. During these live sessions we often split up the current task to complete it faster. The scanner and parser were done collaboratively with relative ease, though when it came to error recovery we decided to assign ourselves specific tasks. Below you will find each member’s programs for which they wrote and handed the error recovery processes for:

- Umer: 1.cm, 2.cm, 5.cm
- Numan: 3.cm, 4.cm

Overall, both members produced quality work, put in equal effort, and ensured that the required tasks were divided evenly.