CS325 Compiler Design Coursework Report

U1504815

November 2, 2017

Introduction

The purpose of this report is to present the recursive descent parser written for the Classroom Object-Oriented Language (COOL). The document will outline the design, development, and testing processes, along with possible alternatives and improvements to the solution identified for this project.

Problem Analysis

In order to parse the given subset of COOL, the input program must first be read and tokenised according to the lexical structure of COOL. Next, the tokens must be fed to a syntax analyser to be checked against the COOL grammar, and finally an output must be produced, which is either the file structure as a list of classes and their methods, or a list of lexical and syntax errors found, with suggestions on how to fix them.

The lexer needs to separate the input stream into COOL tokens, and the parser needs to check that the token stream can be derived from the start symbol of the COOL grammar in an efficient manner, i.e. with as little backtracking and precomputation as possible. The two components should also be able to report as many errors as possible in a single run, and they should be independent of one another.

Solution Overview

The solution consists of two independent components: a lexical analyser and a syntax analyser. The lexical analyser reads the input file and splits it into tokens using the maximal munch principle, while the syntax analyser checks the structure of these tokens in a top-down manner, based on a backtrack-free version of the COOL grammar described in the manual.

The solution prints the structure of valid COOL programs, i.e. classes and their respective methods, one on each line, with methods indented to give a "tree view" of the program. It also includes comprehensive error reporting, in the form of messages which point to erroneous tokens and their locations, and provide suggestions on how to solve the corresponding errors. During syntax analysis, operation precedence is taken into account as well, such that the program can be more easily extended to include a semantic analyser.

Lexical Analysis

The lexical analyser splits the input file into lines, and each line into words, i.e. character sequences with no whitespaces, and finally, each word is split into lexemes using the maximal munch principle. This technique splits strings into multiple lexemes, so after this process, for each line, strings are recreated by binding lexemes found between quotation marks.

Next, the program reads the file one more time to obtain the coordinates (row and column) of each lexeme, such that error messages can point to where the problems are. Finally, each lexeme is converted to a token based on a set of regular expressions. Invalid lexemes generate errors and are discarded, while correct tokens are stored in a list and used by the syntax analyser.

Syntax Analysis

The grammar provided in Section 11 of the COOL manual has a clear and readable form, but it is unsuitable for an efficient parser, as it would require either a large lookahead, or a long time spent backtracking in order to parse tokens. In addition, it does not account for operation precedence. Because of this, it has been modified as follows:

- Precedence is added by splitting the `expr` rule (Appendix A).
- ϵ-productions are eliminated (Appendix B).
- Left recursion is eliminated (Appendix C).
- Left factoring is introduced (Appendix D).
- Small modifications such as eliminating some unit productions are made to make the grammar more compact (Appendix E).

These transformations make the grammar backtrack-free and eliminate the need for First and Follow sets. As such, precomputations are no longer necessary, and the parser becomes very efficient and simple.

In order to output as many errors as possible, the parser attempts to recover every time an error is encountered, by skipping tokens until one that is expected is encountered, such that the parser synchronises with the input [1]. This leads to catching more errors, thus easing the debugging process.

Testing

In order to test functionality for correct COOL programs, the provided COOL examples have been used (without comments), since they are fairly complex and contain a wide variety of instructions. In addition, a custom COOL file based on `hello-world.cl` was also used to test each production rule individually.

In order to test error reporting, illegal tokens were added to programs, both to check the lexer's output and to check how the parser recovered after encountering the wrong tokens. Several special cases were identified in regards to strings, such as having escaped quotation marks or unclosed strings. These required additional modifications to the lexer and to the regular expressions, such that lexemes would not be merged to a starting quotation mark if there was no ending quotation mark for it, and non-COOL characters such as '!' or '>' found inside strings would not throw errors. After correcting the lexer, further tests were performed to determine whether to leave or remove unrecognised tokens before proceeding to parsing. Eventually, the latter option was preferred, as it provided more accurate syntax errors.

Limitations and Possible Improvements

There are several known limitations of this implementation. First of all, when outputting a syntax error, the program outputs the erroneous token's value or name if it is a string, integer, or identifier. However, strings are outputted without whitespace and with double backslashes for escaped characters, which, while still readable, are not completely accurate. This could have been prevented by adding a string formatting function, but the low impact of such a modification and time constraints have led to this feature being excluded from the solution.

Secondly, the lexical analyser reads the file two times, once for lexing, and once for retrieving coordinates. This can hinder performance, regardless of the fact that reading complexity is still linear. This can be improved by writing a more condensed function which performs both tasks in one pass, but due to the significant implementation changes this would have incurred, this optimisation has not been added.

Finally, the parser has one rule which requires a lookahead of two tokens, thus making the grammar LL(2) instead of LL(1); this makes it more difficult to automatically convert the grammar to a recursive descent parser using another program. The lack of such a program makes further grammar modifications more difficult to implement, but there should be no need to make such modifications.

Alternative Implementations

The lexer could have been implemented using `shlex`. This implementation was attempted, but it proved to require much more effort, as there were several differences between `shlex` and COOL tokens. As such, a manual, maximal munch implementation was preferred.

The parser could have been implemented based on a different, simpler grammar (such as the one in Appendix A), and the First and Follow sets could have been computed when the program was run, or hardcoded into the parser, but this would have been either less efficient, in the case of the former, or more tedious, in the case of the latter, despite an increase in code readability and a decrease in the amount of effort required to transform the grammar.

Conclusion

The solution successfully parses COOL programs in an efficient manner, and outputs detailed error reports, should the input program be incorrect. The solution can be further extended to include semantic analysis, as the grammar has been rewritten to account for operation precedence, and the simple structure of the parser enables the usage of a parser generator, which automatically creates the required functions based on the given grammar.

Although certain aspects of the parser could have been more efficient and consistent, the solution still provides a robust, extensible basis for building a full COOL compiler.

References

1. Cooper, Keith D. and Torczon, Linda. Engineering a Compiler. San Francisco: Morgan Kaufmann, 2012, Chapter 3.

Appendices

The grammars below represent various stages of transformations performed on the original grammar, and are all written in the Backus-Naur Form.

Appendix A: grammarA.txt
Appendix B: grammarB.txt
Appendix C: grammarC.txt
Appendix D: grammarD.txt
Appendix E: grammarE.txt