Lexical Analysis & Syntax Analysis

1 Introduction

This report aims to introduce the compiler development work we have completed in the CS323 course project. Our task was to write a compiler for a toy programming language called SPL, which is similar to C but with most of the advanced features removed to make it easier to learn and use. This report will focus on the work we have accomplished in the first phase, which involves implementing the lexer and parser for the SPL language.

In the first phase, we utilized two powerful open-source tools, Flex and Bison, to implement the parser and wrote it using the C programming language. Our parser performs lexical analysis and syntax analysis on SPL source code, identifying tokens and checking the code structure. We achieved this by specifying regular expressions and grammar rules.

This report will first introduce the characteristics and requirements of the SPL language, followed by a detailed description of the design and implementation of our parser. Finally, we will discuss the challenges and issues we encountered in the first phase and provide suggestions for future improvements.

The purpose of this report is to showcase the work we have done in compiler development and illustrate the achievements and experiences we have gained. We hope that this report can be helpful and inspiring to other compiler developers.

2 Develop Tools

- GCC version 11.4.0
- GNU Make version 4.3
- GNU Flex version 2.6.4
- GNU Bison version 3.8.2

3 SPL Grammar

3.1 Lexical

Here is all of the valid tokens in SPL:

- Keywords: int char float **string** struct if else while return
- Brackets: Curly brackets { }, Square brackets [ ], Round brackets ( ).
- Operators and Separators: = < <= > >= != == + - * / && || ! . , ;
- INT, FLOAT, ID: the implementation and requirements are consistent with the requirements.
- CHAR: We have two types of character literals, decimal (base 10) and hexadecimal (base 16) forms. A hexadecimal integer lexeme always begin with "\x", followed by a sequence of hex-digits (0-9, a-f),
- **String: multiple character contained in a pair of double-quotes. when there are nested double-quotation mark in the lexeme, user can use \" to mark.**

3.2 Syntax

We design a Tree structure to store the information of parse tree in a form easy to print. We return the number of line and the value of different token for lex file and store it in the tree node. The nonterminal will be the parent of terminals and nonterminal which is consist of. After parsing the whole code we print the tree from root to leaf. And if there is any error, we will not print the tree.

3.3 Comments

We have implemented a commenting feature in our system, which includes both single-line comments using "//" and multi-line comments using "/* */". This feature allows developers to add comments within the code to provide explanations, notes, or documentation.

Single-line comments, denoted by "//", are used to annotate a single line of code. Anything following "//" on the same line is considered a comment and is ignored by the compiler or interpreter.

Multi-line comments, enclosed between "/*" and "*/", allow developers to add comments that span multiple lines. Any text between "/*" and "*/" is treated as a comment block and is not executed or interpreted by the system.

The commenting feature is particularly useful for improving code readability, documenting code functionality, and making it easier for other developers to understand and maintain the codebase. It

enables developers to provide insights, explanations, or reminders within the code, facilitating collaboration and code comprehension among team members.

4 Implementation And Result

4.1 Lexical

Code + output + analyse

4.2 Syntax

Code + output + analyse

4.3 Comments

Code + output + analyse

4.4 Conflict reducing

Code + output + analyse

5 Conclusion

In the first phase of our CS323 course project, we successfully implemented a parser for the SPL (SUSTech Programming Language) programming language using Flex and Bison. The parser performed lexical analysis and syntax analysis on the SPL source code.

By utilizing Flex and Bison, we automated the process of token recognition and code structure checking. This allowed us to specify regular expressions and grammar rules to define the language's syntax. We leveraged the power of these open-source tools, eliminating the need to manually implement NFA/DFA or parsing algorithms.

Throughout this phase, we focused on building maintainable and extensible code. This work serves as the foundation for subsequent parts of the compiler, enabling us to add further functionality such as semantic checking, intermediate code generation, and target code generation.

By the end of this phase, our SPL parser was able to analyze the source code and identify tokens and their relationships according to the specified grammar rules. This achievement sets the stage for the successful completion of the rest of the compiler.

Overall, this phase provided us with practical experience in lexical and syntax analysis, as well as an understanding of how to automate these processes using powerful tools like Flex and Bison. We

are now well-equipped to proceed with the next stages of the project and continue building a fully functional compiler for the SPL language.