Compile Principles: Lab01

2025-11-29

Aksel Shen
20234001053@m.scnu.edu.cn
South China Normal University

Abstract

In this lab, we implemented a lexer for SysY2022 using Logos — a lexer generator for Rust. The lexer analyzes a source code of SysY2022 and produces a sequence of tokens, which can be used for further processing by the parser.

1 Introduction

As a crucial component of a compiler, a lexical analyzer (or lexer) is responsible for reading the source code and breaking it down into a sequence of tokens, which are the basic building blocks of the programming language.

In this lab, we need to implement a lexer for SysY2022, a simplified version of the C programming language. The lexer should be able to recognize various tokens, including keywords, identifiers, literals, operators, and punctuation.

2 Implementation

There are various ways to implement a lexer, such as handwritten code with state machines, or using lexer generators like Flex (for C/C++) and ANTLR (for multiple languages). The first approach provides more control over the lexing process but can be error-prone and time-consuming; the second approach is more convenient and less error-prone, but may produce less efficient code.

We chose to use Logos, a lexer generator for Rust, to implement our lexer. Logos provides a convenient way to define token types and their corresponding patterns using Rust's attribute macros, combining them into a single derterministic finite automaton (DFA), thus producing a fast and efficient lexer.

Using Logos, we defined a Rust enumeration `Token` that represents all possible token types in SysY2022. Each variant of `Token` uses the `#[token]` or `#[regex]` attribute to specify a literal keyword or a regex pattern for token matching. For example, `KwIf` corresponds to the "if" keyword, while `IntConst` matches both signed and unsigned integers, including hexadecimal and octal representations.

Whitespace and comments are skipped using `#[logos(skip)]` rules.

The main function of our lexer processes command-line arguments to accept optional input and output file paths. If no paths are provided, it defaults to standard input and output. After reading the source code input, it initializes the lexer by calling `Token::lexer`.

The lexer iterates over the source code, tokenizing it into `Token` instances, which are simultaneously associated with their source text and location. Each token is outputted with debug information, including line numbers and its lexeme, as the lab requirement specifies. Error tokens are also gracefully handled and redirected to the output.

3 Testing

To ensure the correctness and robustness of our lexer, we created a integration test that verifies the lexer's behavior against three predefined test cases. These test cases are stored in the `testcase/` directory, each case consisting of a SysY2022 source file (`.sy`).

Our testing logic, implemented in `tests/basic.rs`, automatically discovers all `.sy` files within the `testcase/` directory. For each source file found, the test invokes our lexer executable, instructing it to process the input and generate an output file.

The test suite covers a variety of language constructs, including a simple "hello world" program, logical and assignment operations, and different integer literal formats like hexadecimal and octal. These cases are chosen from the attached `testcase.zip`.

4 Conclusion

In this lab, we successfully implemented a lexer for the SysY2022 programming language using the Logos lexer generator in Rust. Our lexer accurately recognizes various token types and handles whitespace and comments effectively. We also developed a comprehensive testing framework to validate the lexer's functionality against multiple test cases.

This experience has enhanced our understanding of lexical analysis and the practical aspects of compiler construction. The use of Logos significantly streamlined the implementation process, allowing us to focus on defining token patterns rather than low-level parsing logic.