BARZ Compiler: An Overview of the Lexical Analyzer Implementation

Ismael Talib Ridha Barzani
Student ID: 40188139
Department of Electrical and Computer Engineering
Concordia University - Gina Cody School
Montreal, Quebec, Canada
ismaelmergasori@gmail.com

***Abstract*—Lexical analysis is usually the first step in conventional compilers where the conversion of character streams to tokens is performed. In this report, I present the design and implementation of a lexical analyzer for the Broken Automated Reliable Zipper (BARZ) compiler written for a custom programming language. Generally, the lexical analyzer is responsible for converting a sequence of characters from the source code into a sequence of tokens that can be used by a parser. The design of the lexical analyzer is based on well-defined lexical specifications expressed as regular expressions which are given. These specifications are carefully respected to ensure accurate tokenization of the source code. The report includes lexical specifications written in terms of regular expressions, and a finite state automaton (FSA) diagram that illustrates the operation of the lexical analyzer, providing a visual representation of the state transitions based on input characters. The overall structure of the solution is described, highlighting the role of each component in the implementation. Additionally, the report discusses the tools, libraries, and techniques used in the analysis and implementation, justifying the choices made. The implementation leverages the Google Test framework for unit testing to ensure the correctness and robustness of the lexical analyzer. The results demonstrate the bare-minimum functionality of a lexical analyzer in accurately identifying tokens, handling various edge cases, and providing meaningful error messages for invalid input.**

***Index Terms*—Compiler Design, Lexical Analyzer, Regular Expressions**

I. Introduction

The development of compilers remains a fundamental aspect of computer science that is usually not on the highlight, where the translation of high-level programming languages into machine code occurs through several distinct phases. The lexical analyzer, often called a scanner, represents the first phase of this translation process where it transforms the source code from a sequence of characters into a stream of specified tokens that can be processed by subsequent compiler phases. This report presents the design and implementation of a lexical analyzer for my compiler project named Broken Automated Reliable Zipper (BARZ) compiler. The analyzer follows specified lexical rules to recognize various token categories including identifiers, numbers (both integer and floating-point), operators, punctuation, keywords, and comments. Special attention is given to error detection and recovery, ensuring the analyzer can handle invalid inputs gracefully while providing meaningful error messages. The implementation is written in C++ for robust and high performance compilation. The design emphasizes modularity and maintainability through distinctive methods, with comprehensive unit testing to ensure reliability. Through careful consideration of the lexical specifications and systematic state management, the analyzer successfully handles the complexities of tokenization while maintaining robustness and performance.

This is submitted as part of COMP-6421 Assignment #1.

II. Lexical Specifications

The lexical specifications used in the implementation are expressed as regular expressions, closely following the original specifications with some modifications for implementation clarity:

```
id := letter (letter | digit | '_')*
letter := [a-zA-Z]
digit := [0-9]
nonzero := [1-9]
integer := nonzero digit* | 0
float := integer fraction [e[+|-] integer]?
fraction := '.' digit* nonzero | '.0'
operator := '==' | '<>' | '<' | '>' |
            '<=' | '>=' | '+' | '-' |
            '*' | '/' | ':=' | '=>'
punctuation := '(' | ')' | '{' | '}' |
               '[' | ']' | ',' | '.' |
               ':' | ';'
comment := '//' .* \n | '/*' (not-*/)* '*/'
```

These specifications were implemented with careful attention to edge cases and error conditions. The analyzer differentiates between valid and invalid lexemes based on

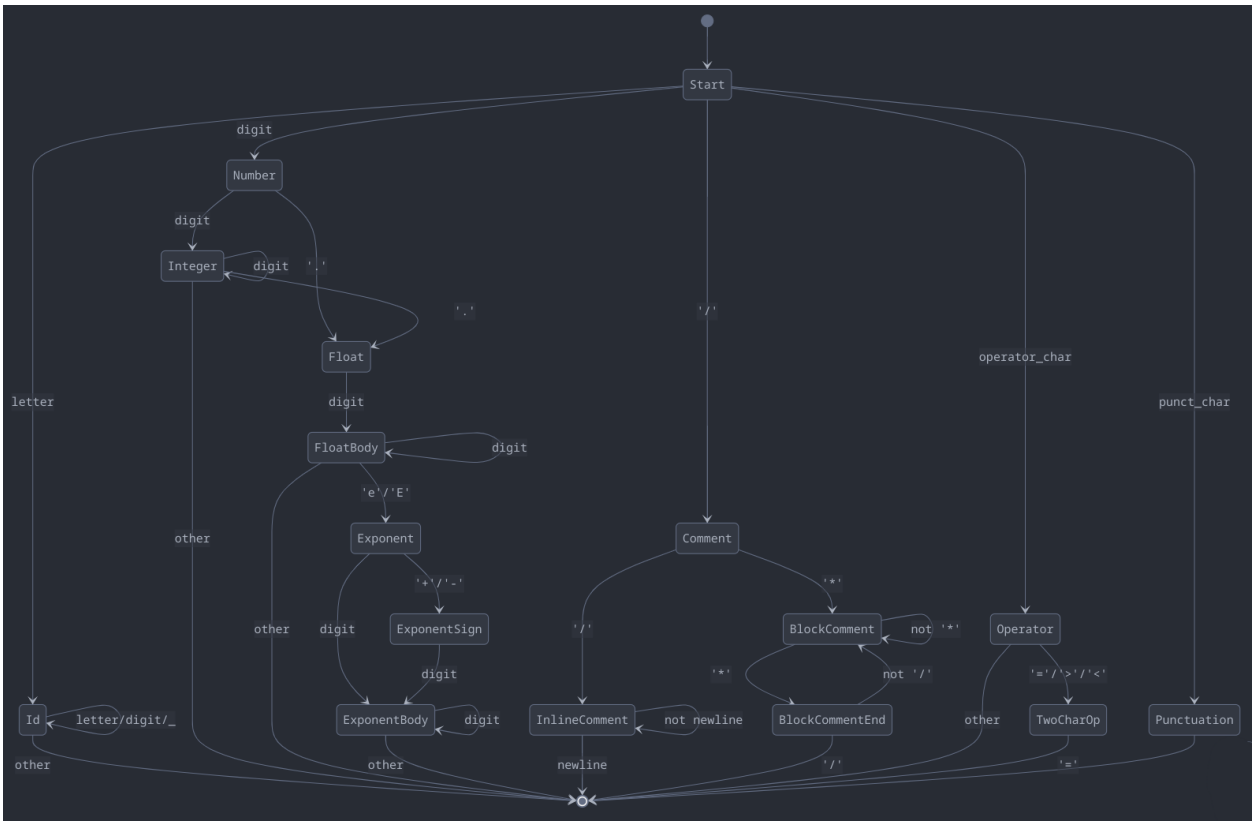

Fig. 1. Finite State Automaton for the Lexical Analyzer written in Mermaid

these patterns, producing appropriate error tokens when invalid input is encountered.

III. Finite State Automaton

The lexical analyzer's operation is governed by a finite state automaton (FSA) that processes the input character stream. Figure 1 illustrates the state transitions and token recognition process which was written in Mermaid.

The FSA includes the following key components:

- Start state that branches to specialized states based on the first character
- Identifier recognition path handling letters, digits, and underscores
- Number recognition paths for both integers and floating-point numbers
- Operator recognition with single and multi-character operator support
- Comment handling for both inline and block comments
- Error states for invalid input patterns

IV. Design

The lexical analyzer implementation follows a modular design with clear separation of concerns. The main components and their roles are as follows:

A. Scanner Class

The core component responsible for tokenization is implemented in `Scanner.h` and `Scanner.cpp`. Key responsibilities include:

- Maintaining state information (current line, column, character)
- Implementing token recognition through specialized scanning methods
- Handling error detection and reporting
- Managing input/output file streams

B. Token Structure

A token is represented by a structure containing:

- Token type (identifier, number, operator, etc.)
- Lexeme (actual text of the token)
- Location information (line number, column)
- Support for error tokens with invalid input handling

C. Token Categories

The implementation organizes tokens into distinct categories:

- Identifiers and keywords (stored in `reservedWords` map)
- Numbers (integers and floating-point values)
- Operators and punctuation symbols

- Comments (both inline and block variants)

The implementation uses a character-by-character scanning approach with lookahead capabilities for multi-character tokens. Error recovery is implemented to continue scanning after encountering invalid tokens, ensuring robust processing of the input stream.

V. Use of Tools

The implementation leverages several modern C++ tools and libraries, each chosen for specific advantages:

A. Google Test Framework

Selected for testing due to its:

- Comprehensive unit testing capabilities
- Structured test organization
- Support for both simple and parameterized tests
- Clear failure reporting and debugging support

B. C++ Standard Library

Core functionality is built using standard library components:

- `std::unordered_map` for efficient token lookup tables
- `std::string` for lexeme manipulation
- File I/O streams for input processing and output generation

C. CMake Build System

Build management is handled through CMake, providing:

- Cross-platform build configuration
- Dependency management
- Seamless test integration
- Build artifact organization

These tools were selected based on their reliability, widespread adoption in the C++ community, and excellent documentation support. The combination provides a robust development environment while maintaining code quality and testability.

VI. Conclusion

The implemented lexical analyzer successfully fulfills its role as the first phase of the BARZ compiler. It accurately tokenizes input according to the specified lexical rules while providing meaningful error messages for invalid input. The modular design and comprehensive test suite ensure maintainability and reliability.

Key achievements include:

- Robust token recognition for all specified lexical elements
- Efficient error recovery and reporting
- Comprehensive test coverage
- Well-documented and maintainable codebase

Future improvements could include:

- Performance optimizations for large input files
- Enhanced error reporting with suggested corrections
- Integration with syntax analysis phase
- Support for additional token types and language features