CPSC 323 Documentation

Name: George Hanna, Christopher Contreras, Burhan Taskin, and Brandon Wu

1. Problem Statement

This project aims to develop a lexical and syntax analyzer for the Rat24F programming language as part of the CS323 course requirements. The analyzer reads source code files written in Rat24F, tokenizes the input, and validates the syntactical structure according to the defined grammar rules. The program processes multiple test files, generating corresponding lexical and syntax analysis output files. This tool aids in understanding the fundamental processes of compiler design, specifically lexical analysis, and parsing, by providing clear insights into tokenization and syntax validation.

2. How to Use Your Program

- **Compiler:** Ensure you have a C++ compiler installed (e.g., g++, clang++).
- **Operating System:** The program is platform-independent but tested on Windows and Linux environments**.**
- **Input Files:** Prepare your Rat24F source code files (e.g., `t1.txt`, `t2.txt`, `t3.txt`) and place them in the same directory as the executable.

Execution Steps

1. **Compilation:**
    - Open your terminal or command prompt.
    - Navigate to the directory containing the source code (`lexer.cpp`).

Compile the program using the following command: `g++ lexer.cpp -o lexer`

    - This command compiles `lexer.cpp` and generates an executable named `lexer`.
2. **Preparing Test Files:**
    - Ensure that your test files (`t1.txt`, `t2.txt`, `t3.txt`) are correctly formatted Rat24F source code files.
    - Place these files in the same directory as the compiled executable.
3. **Running the Program:**
    - **Execute the program by running:** `./lexer`
    - **On Windows, you might run:** `lexer.exe`
4. **Output Files:**

    - Upon execution, the program processes each test file and generates two output files per input:
        - **Lexical Analysis Output:** `lexer1.output`, `lexer2.output`, `lexer3.output`
        - **Syntax Analysis Output:** `syntax1.output`, `syntax2.output`, `syntax3.output`
    - These files contain detailed information about the tokens identified and the syntactical structure derived from the input source code.
5. **Reviewing Results:**
    - Open the `.output` files using any text editor to review the lexical tokens and syntax rules applied.
    - Any syntax errors encountered during the analysis will be reported in the syntax output files with relevant messages and token information.

Notes

- The program automatically skips lines containing `[*` and `*]`, treating them as comments.
- Ensure that the input files adhere to Rat24F syntax to minimize errors during analysis.

3. Design of Your Program

The program is structured into two primary components: Lexical Analyzer and Syntax Analyzer.

1. **Lexical Analyzer:**
    - **Purpose:** Converts the input source code into a sequence of tokens.
    - **Process:**
        - **Tokenization:** Scans the input character by character to identify tokens such as keywords, identifiers, literals, operators, and separators.
        - **Regular Expressions:** Utilizes regex patterns to validate identifiers, integers, and real numbers.
        - **Operator and Separator Matching:** Implements functions to match the longest possible operators and separators to ensure accurate tokenization.
        - **Token Classification:** Differentiates tokens based on predefined categories (e.g., `KEYWORD`, `IDENTIFIER`, `INTEGER`, etc.).
    - **Data Structures:**
        - **Enums and Structs:** `TokenType` enum categorizes token types, and `Token` struct holds token type and value.
        - **Vectors:** Stores lists of keywords, operators, separators, and tokens.
2. **Syntax Analyzer:**
    - **Purpose:** Validates the sequence of tokens against the grammar rules of Rat24F.

- **Process:**
  - **Recursive Descent Parsing:** Implements a set of parsing functions corresponding to grammar rules, allowing for hierarchical analysis of the token sequence.
  - **Grammar Rules:** Each parsing function represents a grammar rule (e.g., `parseProgram`, `parseDeclaration`, `parseStatement`, etc.).
  - Error Handling: Detects and reports syntax errors with informative messages, indicating the type of error and the location within the token sequence.
  - **Indentation Management:** Uses an indentation mechanism to reflect the hierarchical structure of the parsed syntax in the output file.
- **Data Structures:**
  - **Vectors:** Utilizes vectors to manage and traverse the sequence of tokens.
  - **File Streams:** Outputs analysis results to designated output files for both lexical and syntax analyses.

Algorithms and Techniques

- **Longest Match Algorithm:** Ensures that the lexer matches the longest possible operator or separator to prevent ambiguity (e.g., distinguishing between `!=` and `!`).
- **Regular Expressions:** Facilitates the validation of token patterns, enhancing the accuracy of the lexer.
- **Recursive Parsing:** Enables the syntax analyzer to handle nested and hierarchical structures within the source code efficiently.
- **Error Recovery:** Implements mechanisms to skip invalid tokens and continue parsing, preventing the analyzer from halting prematurely due to errors.

File Processing

- **Input Handling:** Reads input files line by line, concatenating them into a single string while ignoring comment lines marked by `[*` and `*]`.
- **Output Generation:** Produces structured output files detailing tokens and syntax rules applied, aiding in debugging and educational purposes.

4. Any Limitation

None.

5. Any Shortcomings

None.