Phase Two Report

Github link

ReadCFG Class

Introduction

The `ReadCFG` class is a critical component in the implementation of LL(1) parsing for context-free grammars (CFGs). This class is responsible for reading and processing CFG rules from a file, converting the grammar to LL(1) form, and providing essential information about the grammar's structure.

Class Structure

Constructors

- The class includes a default constructor and a constructor that takes a file name as a parameter. The latter is used to read the CFG rules from a specified file.

Public Methods

1. **Read and Process CFG Rules:**
   - `read_cfg_file`: Reads and processes CFG rules from a file, populating internal data structures.
   - `printCFG`: Displays the original and LL(1)-converted CFG rules for inspection and verification.

2. **CFG Conversion:**
   - `convert_to_LL1`: Modifies the CFG to LL(1) form by resolving left recursion and left factoring.

3. **Accessors:**
   - `get_non_terminals`: Returns a set of non-terminals in the CFG.
   - `get_terminals`: Returns a set of terminals in the CFG.
   - `get_productions`: Returns the production rules for a given non-terminal.
   - `get_epsilon_symbol`: Returns the symbol representing epsilon in the CFG.
   - `get_dollar_symbol`: Returns the symbol representing the end of input in the CFG.
   - `get_sync_symbol`: Returns the symbol used for synchronization in the parsing table.

4. **Utility Methods:**
   - `is_terminal`: Checks if a given symbol is a terminal.
   - `is_epsilon_symbol`: Checks if a given symbol represents epsilon.
   - `contains_epsilon`: Checks if a set of symbols contains epsilon.
   - `remove_epsilon`: Removes epsilon from a set of symbols.
   - `set_to_string`: Converts a set of symbols to a string for display.

Implementation Details

- The `ReadCFG` class utilizes data structures like `std::map` and `std::set` to efficiently store and manage CFG rules.
- The conversion to LL(1) form involves resolving left recursion and left factoring, ensuring the grammar conforms to LL(1) requirements.
- Accessor methods provide easy retrieval of essential CFG components, enhancing code readability and organization.

Usage

- The class is instantiated with a file name, and the `read_cfg_file` method is called to process CFG rules.
- The LL(1) conversion is achieved through the `convert_to_LL1` method.
- Accessor methods facilitate the retrieval of non-terminals, terminals, and production rules for further processing.

Conclusion

The `ReadCFG` class plays a pivotal role in the LL(1) parsing implementation, serving as the gateway to CFG rules and their conversion. Its clean and organized structure, along with well-defined methods, makes it a robust and versatile component in the LL(1) parsing system. The class effectively encapsulates CFG-related functionalities, contributing to the overall clarity and maintainability of the codebase.

FirstFollow Class

Introduction

The `FirstFollow` class is an integral component in LL(1) parsing, specifically designed to compute and manage the First and Follow sets for non-terminals in a context-free grammar (CFG). These sets play a crucial role in constructing a predictive parsing table, essential for LL(1) parser implementation.

Class Structure

Constructors

- The class includes a constructor that takes a `ReadCFG` object as a parameter, establishing a connection with the CFG rules.

Public Methods

1. **First Set Computation:**
   - `get_first`: Computes and returns the First set for a given non-terminal or rule.

2. **Follow Set Computation:**
   - `get_follow`: Computes and returns the Follow set for a given non-terminal.

3. **LL(1) Parsing Check:**
   - `is_LL1`: Checks if the grammar is LL(1) by examining common elements in First and Follow sets.

4. **Accessors:**
   - `get_first`: Returns the computed First sets for all non-terminals.
   - `get_follow`: Returns the computed Follow sets for all non-terminals.

5. **Print Methods:**
   - `print_first`: Displays the computed First sets for all non-terminals.
   - `print_follow`: Displays the computed Follow sets for all non-terminals.

Implementation Details

- The `FirstFollow` class employs a stack to manage non-terminals in reverse order, facilitating efficient First and Follow set computation.
- Recursive algorithms are used to compute First and Follow sets based on CFG rules and symbols.
- The LL(1) parsing check involves identifying common elements in the First and Follow sets of non-terminals.

Usage

1. **Instantiation:**
   - The class is instantiated with a `ReadCFG` object, establishing a connection with the CFG rules.

2. **First and Follow Set Computation:**
   - The `get_first` and `get_follow` methods are used to compute First and Follow sets for non-terminals, respectively.

3. **LL(1) Parsing Check:**
   - The `is_LL1` method is employed to check if the grammar is LL(1), crucial for predictive parsing.

4. **Accessing Results:**
   - Accessor methods like `get_first` and `get_follow` provide access to the computed sets for further analysis.

5. **Visualization:**
   - The `print_first` and `print_follow` methods facilitate visualization of the computed First and Follow sets.

Conclusion

The `FirstFollow` class serves as the engine for deriving the First and Follow sets, pivotal in LL(1) parsing. Its recursive algorithms and systematic approach contribute to accurate and efficient set computation. The class encapsulates LL(1) parsing logic, providing essential functionalities for constructing a predictive parsing table. The clear structure and utility methods

enhance code readability and maintainability, making the `FirstFollow` class a crucial component in LL(1) parsing implementation.

Table Class

Introduction

The `Table` class is a fundamental component in LL(1) parsing, responsible for constructing and managing the parsing table. This class plays a crucial role in determining the actions to be taken during parsing, making it an essential component in the implementation of a LL(1) parser.

Class Structure

Constructors

- The class includes two constructors. One is a default constructor, and the other takes a file name to initialize the parsing table based on the given CFG.

Public Methods

1. **Parsing Table Construction:**
   - `build_table`: Constructs the LL(1) parsing table by utilizing the First and Follow sets computed by the `FirstFollow` class.

2. **Printing Methods:**
   - `print_table`: Prints the LL(1) parsing table in a tabular format for visualization.

3. **File I/O:**
   - `export_to_file`: Exports the parsing table to a file.
   - `import_from_file`: Imports a parsing table from a file.

4. **Accessors:**

- `get_rule`: Retrieves the production rule for a given non-terminal and terminal pair.
- `get_start_symbol`: Returns the start symbol of the grammar.
- `is_terminal`: Checks if a given symbol is a terminal.

5. **Utility Methods:**
- `get_first_follow`: Retrieves the `FirstFollow` object associated with the grammar rules.
- `get_rules`: Retrieves the `ReadCFG` object containing the grammar rules.

Implementation Details

- The `Table` class relies on the `FirstFollow` class to compute the necessary First and Follow sets for LL(1) parsing.
- Parsing table construction involves iterating through non-terminals, productions, and terminals, populating the table based on computed sets.
- Error handling is implemented to identify and report inconsistencies in the LL(1) parsing table.

Usage

1. **Table Initialization:**
   - The `Table` class is instantiated either with a default constructor or by providing a file name containing the CFG rules.

2. **Parsing Table Construction:**
   - The `build_table` method constructs the LL(1) parsing table, utilizing First and Follow sets.

3. **Accessing Parsing Rules:**
   - The `get_rule` method allows access to the production rule for a given non-terminal and terminal.

4. **Exporting/Importing Parsing Table**:
   - The `export_to_file` and `import_from_file` methods facilitate the storage and retrieval of parsing tables.

5. **Accessing Grammar Information:**
   - The class provides methods to retrieve the `FirstFollow` and `ReadCFG` objects for further analysis.

6. **Printing Parsing Table:**
   - The `print_table` method visually displays the LL(1) parsing table for easier comprehension.

Conclusion

The `Table` class is a crucial component in the LL(1) parsing implementation, responsible for constructing the parsing table based on computed First and Follow sets. Its clear structure, along with utility and accessor methods, enhances code readability and ease of use. The class encapsulates the logic for LL(1) parsing table construction, providing essential functionalities for parsing and error handling. The integration with the `FirstFollow` and `ReadCFG` classes establishes a comprehensive framework for LL(1) parsing in a user-friendly manner.

Parser Class

Introduction

The `Parser` class is a key component in the LL(1) parsing process, responsible for parsing input tokens using the LL(1) parsing table generated by the `Table` class. This class encapsulates the logic for applying production rules, handling errors, and providing feedback on the parsing process.

Class Structure

Constructor

- The class includes a constructor that takes a shared pointer to a `Table` object, initializing the parser with the associated LL(1) parsing table.

Public Methods

1. **Parsing Method:**

- `parse`: Initiates the LL(1) parsing process by applying production rules based on the parsing table. It handles terminal and non-terminal symbols, matches tokens, and provides detailed feedback during the parsing.

2. **Utility Method:**
- `get_rules`: Retrieves the `Table` object associated with the parser, allowing access to parsing rules and information.

Implementation Details

- The `Parser` class utilizes a stack-based approach for parsing, manipulating a stack to manage the symbols and tokens during the parsing process.
- It communicates with the `Table` class to retrieve parsing rules and information about the LL(1) parsing table.
- Error handling is implemented to identify and report parsing errors, such as unmatched symbols and unexpected input.

Usage

1. **Parser Initialization:**
- The `Parser` class is instantiated by providing a shared pointer to a `Table` object, establishing the connection between the parser and the LL(1) parsing table.

2. **Parsing Tokens:**
- The `parse` method is invoked to initiate the LL(1) parsing process. It takes a vector of input tokens, matches them against the parsing table, and provides detailed feedback on the parsing progress.

3. **Accessing Parsing Rules:**
- The `get_rules` method allows access to the associated `Table` object, enabling further exploration of parsing rules and grammar information.

Conclusion

The `Parser` class serves as a crucial element in the LL(1) parsing workflow, bridging the gap between the LL(1) parsing table and the input tokens. Its stack-based parsing approach, coupled with error handling mechanisms, ensures a systematic and insightful parsing process. The class encapsulates the logic for applying production rules, making it an integral part of the

overall LL(1) parsing implementation. The ability to access the underlying parsing rules and table information through the `get_rules` method enhances the flexibility and utility of the `Parser` class.