Phase one: Lexical Analyzer Generator

github repo

- **Structure:**

On the Automaton, Types, State, Utilities, and Conversions Classes:

Introduction

The Automaton, Types, State, Utilities, and Conversions classes are integral components of the first phase of the compiler project, providing a foundation for modeling, manipulating, and converting finite automata (NFAs). This collaborative effort is crucial for converting regular expressions into minimized Deterministic Finite Automata (DFAs), a fundamental step in lexical analysis.

Automaton Class

The Automaton class serves as the cornerstone of the FA framework, offering comprehensive functionalities:

1. **State Management:**

   - Addition, removal, and retrieval of states

   - Identification of the start state

   - Determination of accepting states

2. **Alphabet Management:**

   - Addition and retrieval of input symbols (alphabets)

3. **Transition Handling:**

   - Definition and retrieval of transitions between states

- Addition and removal of transitions

4. Accepting States:

- Identification and manipulation of accepting states

- Association of tokens with accepting states

5. Utility Methods:

- Copying the automaton

- Retrieval of next states for a given state and input symbol

Types Namespace

The Types namespace encapsulates various type definitions and utility functions, providing essential constructs for representing and manipulating automata:

1. Type Definitions:

- state_set_t: A set of states

- state_to_state_map_t: Mapping from one state to another

- epsilon_closure_map_t: Mapping from a state to its epsilon closure

2. Utility Functions:

- Set and vector equality checks

- Functions for adding elements to sets

State Class

The State class embodies individual states within the automaton, offering essential features:

1. **State Properties:**

   - Unique identifier for each state

   - Acceptance status indicating whether the state is an accepting state

   - Token associated with the state

2. **Transition Handling:**

   - Ability to add transitions to other states

3. **Utility Methods:**

   - Equality check for states

Utilities Class

The Utilities class provides a collection of utility functions related to automaton operations:

1. **Union of Automata:**

   - Combines two automata into a single automaton that accepts the union of languages.

2. **Concatenation of Automata:**

- Combines two automata into a single automaton that accepts the concatenation of languages.

3. **Closure of Automata:**

- Creates a new automaton that accepts the closure of the language.

Conversions Class

The Conversions class offers powerful methods for converting between different types of automata and minimizing DFAs:

1. **Epsilon Closure:**

- Computation of the epsilon closure for a given state

2. **Conversion Methods:**

- Conversion from NFA to DFA

- Removal of epsilon transitions

3. **Minimization:**

- Minimization of a DFA using Hopcroft's algorithm

4. **Utility Methods:**

- Creation of new states and dead states

- Retrieval of DFA states corresponding to NFA states

Conclusion

The cohesive framework formed by the Automaton, Types, State, Utilities, and Conversions classes establishes a strong foundation for modeling, manipulating, and converting FAs. Their well-structured design and comprehensive functionalities contribute to the maintainability and extensibility of the compiler project.

Creation Directory Report

Introduction

The Creation directory is a critical component of the first phase of the compiler project, responsible for constructing and manipulating lexical rules, automata, and essential utility operations.

Key Components

The Creation directory comprises four primary classes:

1. **Constants Class:**

- Central repository for essential constants used throughout the compiler project.

- Promotes encapsulation and standardized variable naming.

2. **InfixToPostfix Class:**

- Converts regular expressions from infix to postfix notation.

- Fundamental for subsequent stages in the compiler project.

3. **ToAutomaton Class:**

- Converts regular expressions into corresponding automata using Thompson's construction algorithm.

4. **LexicalRulesHandler Class:**

- Manages lexical rules and automata.

- Handles processing of rules specified in an input file.

Detailed Analysis of Key Components

Constants Class

- Centralized Constants:

- Defines constants related to lexical analysis.

- Promotes consistency and avoids magic numbers or strings.

- Readability and Maintainability:

- Descriptive names for constants enhance code readability.

- Easy modification without affecting the rest of the codebase.

InfixToPostfix Class

- **Infix to Postfix Conversion**

  - Employs algorithms for accurate expression conversion.

- **Integration with Lexical Rules Handling:**

  - Seamless integration with the LexicalRulesHandler class.

ToAutomaton Class

- **Thompson's Construction:**

  - Implements Thompson's construction algorithm for efficient automaton generation.

- **Integration with Lexical Rules Handling**

  - Collaborates with the LexicalRulesHandler class for a comprehensive lexical analysis workflow.

LexicalRulesHandler Class

- **Rule Processing:**

  - Reads lexical rules and interprets regular expressions.

  - Utilizes the ToAutomaton class for automaton creation.

- **Automata Generation:**

- Converts regular expressions into minimized DFAs.

- Manages priorities and exports automata to output files.

- Prioritization:

- Establishes priorities for different tokens based on their order in the input file.

- Integration with Other Classes:

- Seamless integration with InfixToPostfix and ToAutomaton classes.

Overall Structure

The Creation directory exhibits a well-structured and modular design, adhering to object-oriented principles. The classes work cohesively, contributing to code organization, readability, and maintainability.

Conclusion

The Creation directory serves as a foundational element, handling lexical rules, constants, and automata creation. Each class within the directory is designed for a specific role, contributing to the success and clarity of the compiler project. The integration of these classes forms a robust framework for lexical analysis and automaton manipulation.

Prediction Directory Report

Introduction

The Prediction directory plays a critical role in predicting and extracting tokens from a given program text using a predefined automaton. The Predictor class orchestrates this process, managing priorities and efficiently extracting tokens.

Key Components

The Prediction directory primarily encompasses the Predictor class:

1. **Predictor Class:**

   - Core component for token prediction and extraction.

Detailed Analysis of the Predictor Class

The Predictor class offers several crucial functionalities:

1. **Automaton Integration:**

   - Utilizes a shared pointer to an automaton for flexible linkage.

   - Handles transitions within the automaton to predict and extract tokens.

2. **Priority Management:**

- Incorporates a map of token priorities for resolving ambiguity.

- Assigns priorities based on predefined order or external considerations.

3. **Program Text Handling:**

- Reads program text from a file and converts it into a string for efficient processing.

- Manages the program text index during token extraction.

4. **Token Extraction:**

- Iterates through the program text using the automaton to predict and extract tokens.

- Considers whitespace characters for accurate tokenization

.

5. **Dead State Handling:**

- Identifies and handles dead states within the automaton, optimizing token extraction.

Utility Methods

The Predictor class incorporates several utility methods:

1. **File Reading:**

   - Method to read program text from a file, promoting modularity and reusability.

2. **Dead State Identification:**

   - Method to find and handle dead states within the automaton, improving prediction efficiency.

3. **Next State Calculation:**

   - Determines the next state in the automaton based on the current state and input symbol.

Overall Structure

The Prediction directory exhibits a well-structured and cohesive organization, adhering to object-oriented principles. The Predictor class encapsulates prediction logic, abstracting complexities of automaton traversal and token extraction. Integration with automata and priority management ensures a seamless workflow for predicting tokens.

Conclusion

The Prediction directory, with its Predictor class, bridges lexical analysis and parsing phases. Efficient prediction and extraction of tokens from program text are essential for subsequent stages in the compiler pipeline. The Predictor class, with its thoughtful design and integration with automata and priority management, contributes to the overall success and accuracy of the compiler project.

Main Class Report

Introduction

The main class is pivotal in the compiler project, acting as the orchestrator for the first phase. It seamlessly integrates creation and prediction processes, ensuring efficient and accurate analysis of program text.

Key Components and Functionalities

The main class encapsulates several critical components and functionalities:

1. **Input Handling and Validation:**

   - Command-Line Argument Parsing: Parses command-line arguments for essential paths.

   - Input Validation: Ensures all necessary paths are specified.

2. **Creation Process:**

   - Automaton Generation: Initiates creation phase for generating automata.

   - Automaton Export: Exports generated automata for use during the prediction phase.

3. **Prediction Process:**

   - Automaton and Priority Import: Imports final automaton and token priorities.

- Token Prediction and Extraction: Initiates Predictor class for extracting tokens.

- Token Output: Prints extracted tokens and exports them to an output token file.

4. **Encapsulation and Modularity:**

- Clear Separation of Concerns: Separates creation and prediction processes.

- Maintainable and Readable Codebase: Enhances maintainability and readability.

Conclusion

The main class stands as a cornerstone of the compiler project, orchestrating the initial phase and driving the overall compilation process. Its effective integration of creation and prediction functionalities, coupled with clear separation of concerns, underscores its significance in the development of a robust and maintainable compiler. This report provides a comprehensive overview of the main class, highlighting its essential components and contributions to the success of the compiler project.

- **Assumptions:**

- The predictor takes into account that a token in line n has a higher priority than a token in line n + 1. Based on that, you should do the following:
    - Put punctuation and keywords definitions before id and num in rules.
    - If you encounter a 10, it will be digits, not num, as digit has a higher priority.
- Try to add the dependent rules last to minimize the calculations needed, for example:
    - Digit = definition
    - Digit: definition
    - Num: definition
- For the character ';':
    - The program can't read it (i don't know why) so don't use it.
    - ;:
    - Start State: 1
    - Final States: 0
    - Transition Table:
    -
    - 0 2 2
    - 1 2 3
    - 2 2 2
    - 3 0 2
    - Regex: (.)
    - Tokens: ;
    -
    -
    -

- **Minimized DFAs examples:**
  - Here is the rules example I am working on, and this is the program example I am parsing.
  - Tokens:
  - chose not to minimize the DFA of the whole grammar to preserve the information of the tokens. That is, each state should have one accepting token. If we minimize the DFA, we will have some states that can accept multiple tokens, which is theoretically possible, but it clashes with the priority of the tokens read from the rules file. For example, one state can accept [id, keyword, punctuation, and more if possible], but we should keep the state pure [id, keyword, letter, etc.]. So here is the DFA of the grammar but not minimized.