Lab Terminal

Fa20-bcs-009

Maryam Amjad

Question 4: How functions work.

Regular Expression (RE) to DFA:

1. Lexical Analysis and Tokenization:

- - Use a lexer or tokenizer to convert the raw regular expression string into a sequence of tokens.
- - Tokens represent individual symbols, operators, and other elements.

2. Parse Tokens and Build NFA:

- - Parse the tokens generated in the previous step.
- - Build a Non-deterministic Finite Automaton (NFA) using algorithms like Thompson's Construction.

3. Convert NFA to DFA:

- - Implement the subset construction algorithm to convert the NFA to a Deterministic Finite Automaton (DFA).
- - DFA simplifies the representation and enhances efficiency.

4. Minimize DFA:

- - Apply a DFA minimization algorithm, such as Hopcroft's algorithm, to further optimize the DFA.
- - Identify and merge equivalent states to reduce the number of states while preserving language acceptance.

Putting it All Together:

- - Create a cohesive class or module that orchestrates the entire process.
- - Instantiate and use the necessary components: lexer, NFA builder, DFA converter, and DFA minimizer.
- - Pass the regular expression through each step to get the minimized DFA.

Lexical Generator:

1. Token Definition:

- - Define classes or structures to represent lexical tokens, including keywords, identifiers, operators, and literals.
- - Associate each token definition with a corresponding regular expression pattern.

2. Regular Expressions for Tokens:

- - Assign regular expressions to each token definition, specifying the patterns that correspond to each token type.
- - These regular expressions define the lexemes to be recognized during the lexical analysis.

3. RE to NFA:

- - Use the token-specific regular expressions to construct individual Non-deterministic Finite Automata (NFAs) for each token.
- - Employ algorithms, possibly Thompson's Construction, to build NFAs based on regular expressions.

4. Combine NFAs:

- - Merge the individual NFAs representing different tokens into a single NFA.
- - This combined NFA recognizes the entire lexical structure of the programming language.

5. Convert to DFA:

- - Utilize the subset construction algorithm to convert the combined NFA into a Deterministic Finite Automaton (DFA).
- - The DFA streamlines token recognition and improves efficiency.

6. Minimize DFA:

- - Apply a DFA minimization algorithm, such as Hopcroft's algorithm, to minimize the DFA.
- - Merge equivalent states to reduce the number of states while preserving language acceptance.

7. Lexical Analyzer Generation:

- - Generate code for a lexical analyzer based on the minimized DFA.
- - The lexical analyzer scans the input stream and recognizes tokens based on the minimized DFA.
- - Implement logic for token extraction and handling in the generated lexical analyzer.

Putting it All Together:

- - Develop a cohesive class or module that encapsulates the entire lexical generator process.
- - Instantiate and utilize the necessary components: token definitions, regular expressions, NFA builder, DFA converter, and DFA minimizer.
- - Sequentially pass through each step, from token definition to lexical analyzer generation, to achieve a fully functional lexical generator.

Parser Generator for SLR, LALR(from LR(1)), LALR(from LR(0)), LR(1):

1. Grammar Definition:

- Define the grammar for the language using appropriate notation (e.g., BNF).
- Clearly specify terminals, non-terminals, and production rules.

2. SLR Parser Generator:

- Implement an SLR (Simple LR) parser generator.
- Construct the LR(0) sets for the grammar.
- Build the SLR parsing table using LR(0) sets and follow sets.
- Generate parser code that utilizes the SLR parsing table for shift-reduce parsing.

3. LR(1) Parser Generator:

- Implement an LR(1) parser generator.
- Construct the LR(1) sets for the grammar.
- Build the LR(1) parsing table using LR(1) sets and lookaheads.
- Generate parser code that utilizes the LR(1) parsing table for shift-reduce parsing.

4. LALR(from LR(1)) Parser Generator:

- Implement an LALR (Look-Ahead LR) parser generator derived from LR(1).
- Construct the LR(1) sets for the grammar.
- Merge states with identical core LR(1) sets to form LALR sets.
- Build the LALR parsing table using the merged LALR sets and lookaheads.
- Generate parser code that utilizes the LALR parsing table for shift-reduce parsing.

5. LALR(from LR(0)) Parser Generator:

- Implement an LALR parser generator derived from LR(0).
- Construct the LR(0) sets for the grammar.
- Merge states with identical core LR(0) sets to form LALR sets.
- Build the LALR parsing table using the merged LALR sets and follow sets.
- Generate parser code that utilizes the LALR parsing table for shift-reduce parsing.

6. Code Generation:

- Develop logic for generating parser code in the target programming language (C#, in this case).
- Emit code for parsing actions, error handling, and constructing the parse tree.
- Ensure the generated code adheres to the parsing table entries for SLR, LR(1), LALR(from LR(1)), and LALR(from LR(0)) parsers.

7. Integration and Usage:

- Provide interfaces or methods for users to integrate the generated parser into their projects.
- Offer clear documentation on how to use the parser generator and the generated parser code.
- Demonstrate examples of parsing with the generated parser for various grammatical constructs.

Putting it All Together:

- Design a cohesive class or module that encompasses the entire parser generator process.
- Enable users to specify grammars, choose parsing algorithms (SLR, LR(1), LALR), and generate parser code accordingly.
- Empower developers to seamlessly integrate and utilize the generated parsers for language processing in their applications.