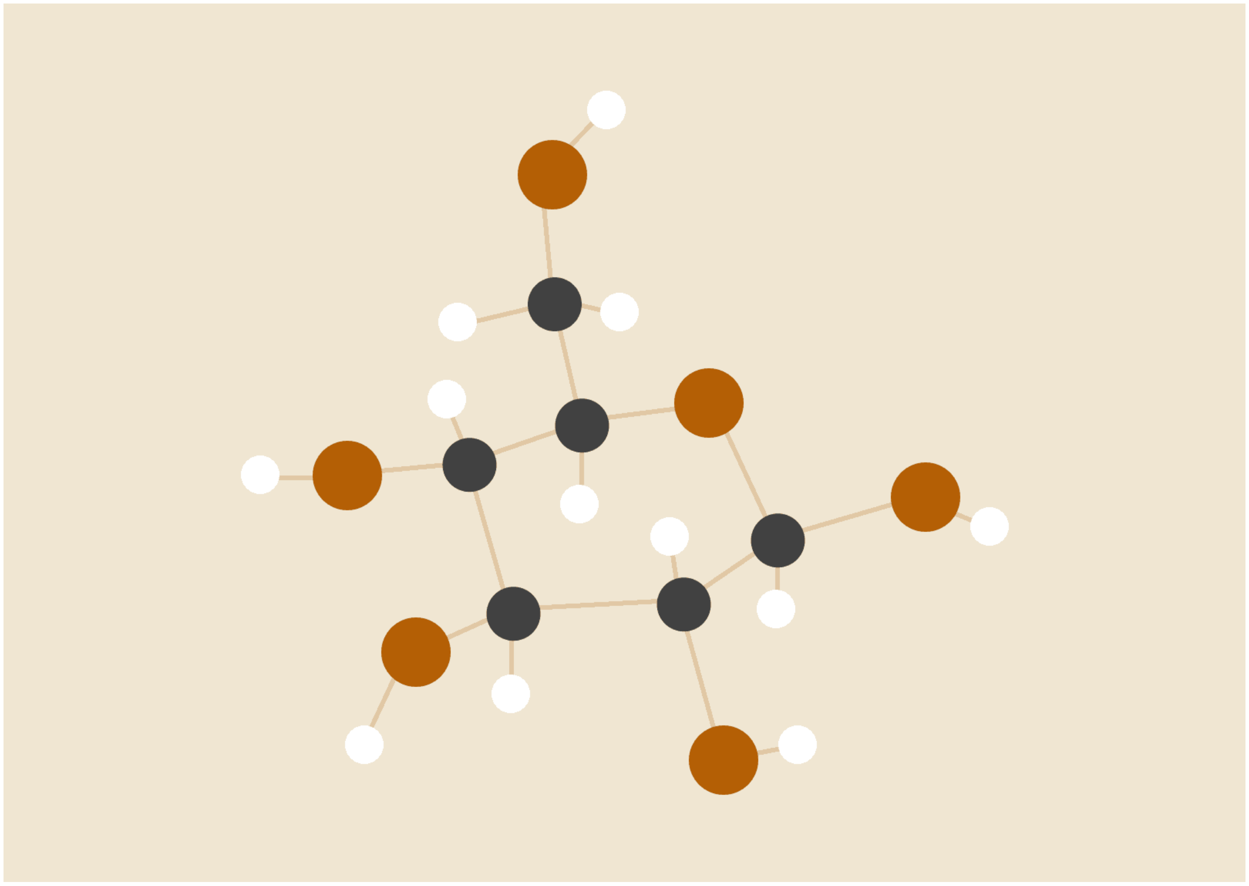
COMPILERS PROJECT REPORT

Phase 1: Lexical Analyzer Generator



Ahmed Mohamed Ali	**20010198**
Ahmed Medhat Saad	**20010218**
David Michael Nagib	**20010545**
Abdullah Samy Talha	**20010888**

04.12.2024

CSED 2025

0 Description & Objectives

Our task in this phase is to design and implement a lexical analyzer generator tool. This generator is required to automatically construct a lexical analyzer from a regular-expression description of a set of tokens. The included steps are constructing a non-deterministic finite automata (NFA) for the given regular expressions, combining these NFAs with a new starting state, converting that NFA to a DFA, minimizing that DFA, and finally, producing a lexical analyzer program that simulates the minimal DFA along with its transition table.

The generated lexical analyzer reads its input one character at a time until it finds the longest prefix of the input that matches one of the given regular expressions. It creates a symbol table of each lexeme and its token identifier. If more than one regular expression matches some longest prefix of the input, the lexical analyzer breaks the tie in favor of the regular expression listed first in the regular specifications. If a match exists, the lexical analyzer outputs the token class and the attribute value. If none of the regular expressions matches any input prefix, an error recovery routine is to be called to print an error message and to continue looking for tokens.

1 Code Modules (Classes)

We divided the required functionalities among multiple objects, each one handling a different problem in the production line of the lexical analyzer. The following table describes the functionality of the main classes.

Class	Functionality
NFA	This class stores the members of an NFA (states, initial state, transitions, accepting states), and it also encapsulates all functionality related to an NFA: - Transition function from a set of states given some input. - Epsilon closure for a set of states. - Operators over NFAs (union, concatenation, and closure).
DFA	It is a similar class to NFA with methods that deal with deterministic single states instead of a set. It does not include operators like those defined over NFAs and includes a function to validate the DFA (each state has a transition on receiving any input from the domain).
Regex Analyzer	The `RegexAnalyzer` class is designed to parse and process regular definitions, expressions, keywords, and punctuations from a file, generating a unified Non-Deterministic Finite Automaton (NFA). Its purpose is to resolve dependencies between tokens, and handle operators (`*`, `+`, `|`, etc.) all of this is done using these methods: parseLexicalRules(),resolveRegularDefToken(),resolveRegularExpToken(), generateNFA()
NFA2DFA	This class is designed to convert a **Non-deterministic Finite Automaton** (NFA) into an equivalent **Deterministic Finite Automaton** (DFA). This process ensures that the resulting DFA has the same language acceptance properties as the given NFA. It includes methods for: - Mapping NFA states to DFA states. - Performing conversion through subset construction. - Efficiently handling epsilon transitions and closures.
DFA Minimizer	Initialized with some DFA, this object returns the minimal DFA that accepts the same language by using minimize().
Lexical Analyzer	An object that internally works with a DFA object to parse some input stream into lexeme-token identifier pairs (symbol table).
Lexical Analyzer Generator	An object representing the combined pipeline of building a lexical analyzer based on some regular expressions in a file. Uses all the previous objects and manages data flow between them.

2 Algorithms

1. Regular expressions to NFA conversion

Dependency Resolution

- The resolveRegularDefToken() and resolveRegularExpToken() methods use a queue to resolve dependencies iteratively.
- A token is marked resolved if all components were resolved and turned into basic chars and operators.
- If a token's regex depends on another token it checks if it is resolved or not if resolved it will take the other token regex and replace it inside of its regex. If not it will be marked unresolved and put in the queue.
- If a token's regex is resolved the token will be marked resolved and removed from the queue.
- This keeps happening until all tokens are resolved and the queue is empty

NFA Construction

- Each regular expression token is turned into an NFA then all NFAs are combined using union using two functions:
    - generateNFA():
        1. Converts each RegularExpToken into an individual NFA using RegularExpTokenToNFA.
        2. Combines all NFAs into one unified NFA using a union operation.
        3. Returns the final NFA representing all tokens.
    - RegularExpTokenToNFA():
        1. Converts keywords into base NFAs, handling characters, ε-transitions (L), and custom IDs.
        2. Uses parentheses to group expressions and resolves operators:
            - Repetition (*, +): Modifies the preceding NFA.
            - Concatenation: Combines consecutive NFAs.
            - Union (|): Merges multiple NFAs with a new start state.
        3. Finalizes the NFA by assigning the token ID to its accepting state and returns it.

2. NFA operations

We have defined common operations used in composing the final NFA that capture all token patterns. Given NFAs A1 and A2 which accept languages L1 and L2, respectively, we can use methods defined for the NFA class to obtain an NFA that accepts L1 U L2, L1L2, L1*, or L1+. The methods used algorithms that we have studied in the lecture, such as linking the accepting state of A1 and the starting state of A2 in case of finding the NFA that accepts L1L2.

We have also defined an epsilon closure operator, which, given a set of states, will return these states as well as all other states reachable from them by empty transitions only. This was done by utilizing the BFS algorithm, where all visited states are added to the epsilon closure.

3. NFA to DFA conversion

- Implements the subset construction algorithm to convert an NFA into a DFA using a given input domain.
- **Algorithm (Subset Construction)**:
    1. **Initialization**:
        - Identify and map the NFA's epsilon closure of its initial state to the DFA's initial state.
        - Initialize a queue to manage unprocessed DFA states.
        - Use a `state_map` to maintain mappings between DFA state IDs and sets of NFA states.
        - Track DFA transitions, accepting states, and a dead state for undefined transitions.
    2. **BFS Exploration**:
        - For each DFA state derived from a set of NFA states, process all input symbols:
            - Determine the set of reachable NFA states for each input symbol.
            - Compute the epsilon closure of the reachable states.
            - Map this closure to a DFA state, creating a new state if necessary.
            - Record transitions in the DFA transition table.
        - Repeat until all reachable DFA states are processed.
    3. **Result Construction**:
        - Create a `DFA` object using the constructed states, transitions, initial state, and accepting states.

- **Used data structures**:
    1. **Unordered Map (`unordered_map`)**:
        - Maps DFA states to corresponding NFA state sets (`state_map`).
        - Maintains DFA transitions (`dfa_transitions`).
        - Tracks accepting states in the DFA (`dfa_accepting`).
    2. **Unordered Set (`unordered_set`)**:
        - Represents sets of NFA states (used during subset construction).
        - Stores DFA state IDs.
    3. **Queue (`queue`)**:
        - Facilitates BFS traversal of DFA states derived from NFA state sets.
    4. **Vector (`vector`)**:
        - Represents the input domain (alphabet) as a list of characters.

5. DFA minimization

For minimizing a DFA, we use the table-filling method which is an algorithm that utilizes the Myhill-Nerode theorem. This method is slightly different from the equivalence partitioning method (which we studied) but it boils down to the same concept of finding distinguishable states if they lead to distinguishable states on some input. We also modified the algorithm to include multiple accepting states, where two accepting states can be distinguishable if they accept two different patterns.

[Algorithm picture]

Once we obtain the distinguishable & indistinguishable pairs of states, we start constructing a mapping from old to new states where indistinguishable states map to the same state in the minimal DFA. Transitions are then mapped according to the same principle where source and destination states are mapped to the minimal DFA states while keeping the input triggering the transition the same.

6. Lexical Analyzer

Given some DFA which accepts multiple tokens with multiple patterns, we perform lexical analysis with some input file written in the language accepted by the DFA. This is done by parsing the file character by character (while utilizing a custom-made buffer), and using each character to transition through the DFA. To capture lexemes and their corresponding token IDs, we use the maximal munch algorithm that we studied, where the longest prefix of the unparsed portion of the file is captured as a lexeme with its corresponding token identifier and we remove it from the buffer, advancing to start the whole process again right after it to capture the next token.

Assumptions

1. Keywords have a higher priority
2. The Lexical Rules files will be in the correct format
3. The input file does not contain characters outside the specified input domain in the rules file