Text

Description automatically generated

**Activity based**

**Project Report on**

**Compiler Design BTECCE22603**

**Project - I**

**Submitted to Vishwakarma University, Pune**

**Under the Initiative of**

**Contemporary Curriculum, Pedagogy, and Practice (C2P2)**

**By**

**Anujkumar Ashok Sutar  
SRN No : 202200728**

**Roll No : 50**

**Div : B**

**Third Year Engineering**

**Department of Computer Engineering**

**Faculty of Science and Technology**

**Academic Year**

**2024-2025**

**Problem Statement :**

Design and implement a lexical analyzer and syntax analyzer for a State Machine Specification Language (SMSL) that allows users to describe and define state machines in a structured format. The primary goal is to develop a robust system capable of parsing state machine specifications, identifying components such as states, transitions, events, actions, and other relevant detail and to create a robust system capable of analyzing the semantic meaning of state machine specifications and generating intermediate representation code for further optimization or execution by downstream systems. change the according to the question

**Tokens for the language specification in the problem:**

In the context of the **State Machine Specification Language (SMSL)** lexical analyzer, the tokens are the distinct elements that lexical analysis breaks the input into. These tokens can be classified into various categories such as:

* **Keywords (State Machine Components)** – Represent essential components of a state machine, such as defining states, transitions, and actions.  
  **(e.g., state, transition, event, action, start, end)**
* **Identifiers (Names of States, Transitions, and Events)** – Represent user-defined names for states, transitions, and events.  
  **(e.g., State1, transitionA, event\_trigger, myAction)**
* **String Literals** – Represent text enclosed within double or single quotes.  
  **(e.g., "Hello World", 'State Transition Message')**
* **Numbers** – Can be integers representing values in state definitions or event priorities.  
  **(e.g., 25, 100, 0, 1)**
* **Symbols (Delimiters and Separators)** – Special characters used for defining the structure of the state machine.  
  **(e.g., {, }, ;, :, ,, ->)**
* **Operators (State Transition Operators)** – Represent transition-related operators.  
  **(e.g., -> for defining state transitions)**
* **Whitespace** – Includes spaces, tabs, and newline characters used to format the code. (Ignored during tokenization.)
* **Comments** – Represent single-line comments that start with #. Used for annotating code.

**Grammar for the language specification in the problem**

**Grammer for SMSL**

1. <state\_machine> ::= <state\_declaration\_list> <transition\_list>
2. <state\_declaration\_list> ::= <state\_declaration> | <state\_declaration> <state\_declaration\_list>
3. <state\_declaration> ::= "state" <identifier> { <action\_list> }
4. <action\_list> ::= <action> | <action> <action\_list>
5. <action> ::= "action" <identifier>
6. <transition\_list> ::= <transition> | <transition> <transition\_list>
7. <transition> ::= "transition" <identifier> "from" <identifier> "to" <identifier> "on" <event>
8. <event> ::= "event" <identifier>
9. <identifier> ::= [a-zA-Z\_][a-zA-Z0-9\_]\*
10. <string> ::= " [^\"]\* "
11. <number> ::= [0-9]+(\.[0-9]+)?
12. <comment> ::= "#" [^\n]\*

**Explanation of Grammar Elements for SMSL**

* <state\_machine>: Represents an SMSL state machine, consisting of state declarations and transitions between states.
* <state\_declaration\_list>: A list of state declarations, defining the various states of the state machine.
* <state\_declaration>: Declares a state using the "state" keyword, followed by an identifier (state name) and a block { ... } containing actions associated with the state.
* <action\_list>: A list of actions defined within a state. Each state can contain multiple actions that are executed when the state is active.
* <action>: Defines an action inside a state using the "action" keyword followed by an identifier (action name).
* <transition\_list>: A list of state transitions, defining how the system moves from one state to another.
* <transition>: Specifies a transition between states using the "transition" keyword. It includes:
  + The transition name (identifier).
  + The source state (from keyword).
  + The destination state (to keyword).
  + The triggering event (on keyword).
* <event>: Represents an event that triggers a transition. Defined using the "event" keyword followed by an identifier (event name).
* <identifier>: Represents names for states, actions, transitions, and events. It follows standard variable naming rules (letters, digits, underscore).
* <string>: Represents text values enclosed in double quotes ("...").
* <number>: Defines integer or floating-point numbers.
* <comment>: Represents single-line comments, starting with #. Comments are ignored by the parser and are used for documentation.

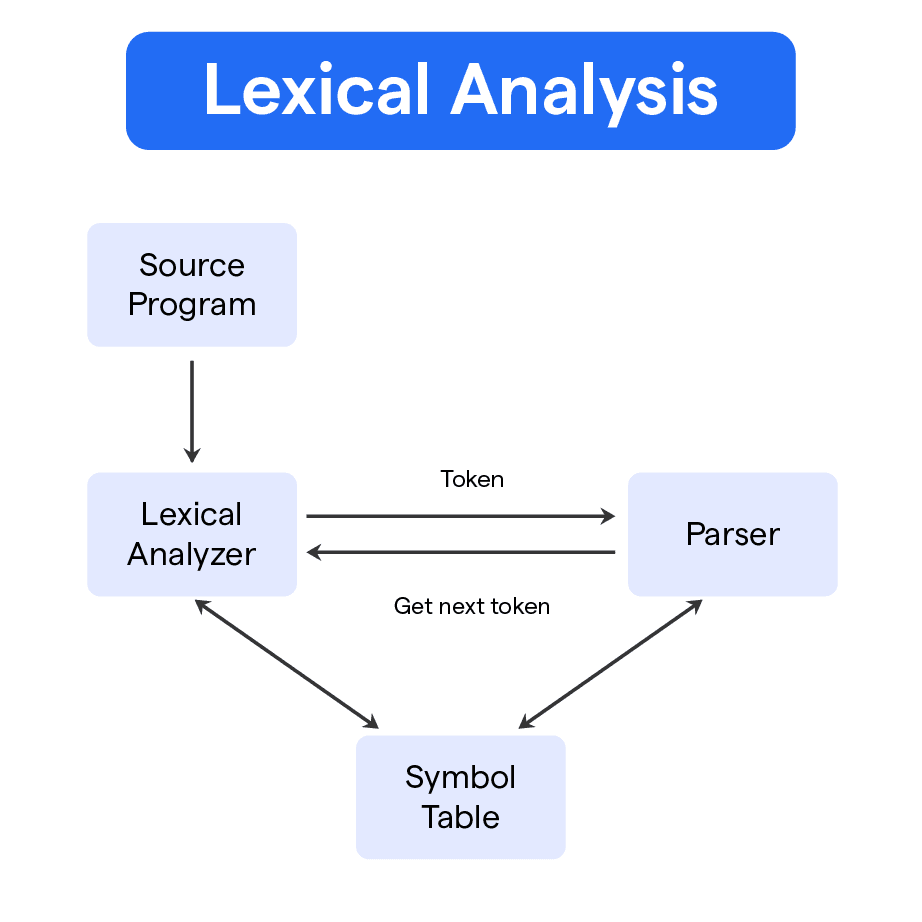
**Implementations details of Lexical and Syntax Analysis ( Diagram, algorithm, error detection etc.)**

**1. Lexical Analysis (Tokenization)**

The purpose of **lexical analysis** is to break down the **SMSL** into **tokens** that will be used in **syntax analysis**.

**Lexical Analysis Algorithm for SMSL**

1. **Read** the input SMSL script line-by-line.
2. **Remove comments** (#) and **ignore blank lines**.
3. **Skip white spaces** and analyze characters to **identify tokens**.
4. **Classify tokens** into different categories:
   * **Keywords** (state, action, transition, event, from, to, on).
   * **Identifiers** (State names, transition names, event names) → [a-zA-Z\_][a-zA-Z0-9\_]\*.
   * **Operators** (= for assignments).
   * **String literals** ("Running", 'Stopped').
   * **Numbers** (25, 3.14).
   * **Control characters** ({, }, (, ), ;).
5. **Store** recognized tokens in a **Token Table** (mapping **tokens** to their **types**).
6. **Output tokens** for the **syntax analyzer**.
7. **Store recognized tokens** in a **Token Table** (mapping tokens to their types).
8. **Output tokens** for syntax analysis.



**DIAGRAM 01: LEXICAL ANALYSIS**

**2. Syntax Analysis**

The Syntax Analysis (Parsing) phase ensures that the sequence of tokens follows the valid grammar rules defined for SMSL.

**Algorithm for Syntax Analysis**

1. Read the list of tokens generated by the lexical analyzer.
2. Verify whether the tokens match predefined grammar rules for SMSL, such as:
   * State Definition: state <IDENTIFIER> { <STATEMENTS> }
   * Event Declaration: event <IDENTIFIER> { <TRANSITIONS> }
   * Transition Definition: on <EVENT> -> <STATE>
   * Assignment Statement: <IDENTIFIER> = <VALUE>
   * Conditional Statement: if ( condition ) { ... }
   * Loop Statement: while ( condition ) { ... }
   * Function Definition: function <NAME>() { <STATEMENTS> }
   * Function Call: <NAME>(<ARGUMENTS>)
3. If a token violates a grammar rule, report a syntax error.
4. If the structure is correct, generate a Parse Tree representing the hierarchical structure of the SMSL script.

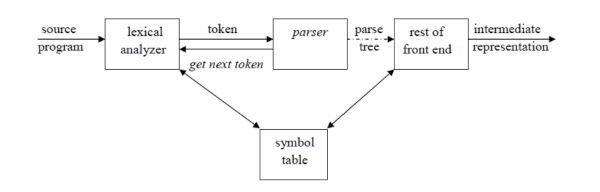


DIAGRAM 02: SYNTAX ANALYSIS

**3. Error Handling**

To improve error handling, the parser should:

• Print Detailed Error Messages

* Display errors with line numbers and column positions to pinpoint the exact location of the issue.
* Include error codes or categories (e.g., SYNTAX\_ERROR, UNDEFINED\_VARIABLE, MISSING\_OPERATOR).

• Highlight Violated Lexemes or Syntax Rules

* Underline or color-code the specific lexeme causing the issue in terminal output.
* Clearly state which rule was violated (e.g., "Expected an operator, found identifier instead").

• Provide Meaningful Hints & Suggestions

* Suggest possible fixes (e.g., "Did you mean == instead of = for comparison?").
* Recommend correct syntax with examples (e.g., "Use if [ condition ]; then ... fi for conditional statements").
* If an assignment is missing, suggest valid variable assignments (e.g., "x=10 instead of x 10").

• Handle Common Errors Gracefully

* Detect unclosed strings and suggest adding closing quotes.
* Identify mismatched brackets or parentheses and suggest corrections.
* Warn against unsupported special characters in variable names or commands.

• Provide Debugging Information

* If an error occurs, print a summary of parsed tokens before the error.
* Offer a step-by-step analysis to help users debug their scripts efficiently**.**

**Code with input , output screen shots**

**LexicalAnalyzer.java**

import java.io.\*;

import java.util.\*;

import java.util.regex.\*;

public class LexicalAnalyzer {

// Define ANSI color codes for terminal styling

public static final String RESET = "\u001B[0m";

public static final String CYAN = "\u001B[36m";

public static final String MAGENTA = "\u001B[35m";

public static final String GREEN = "\u001B[32m";

public static final String RED = "\u001B[31m";

public static final String YELLOW = "\u001B[33m";

// Define SMSL token types with regex patterns and index mapping

private static final Map<String, String> TOKEN\_TYPES = new LinkedHashMap<>();

private static final Map<String, Integer> TOKEN\_INDEX = new LinkedHashMap<>();

static {

TOKEN\_TYPES.put("STATE", "\\bstate\\b");

TOKEN\_TYPES.put("TRANSITION", "\\btransition\\b");

TOKEN\_TYPES.put("EVENT", "\\bevent\\b");

TOKEN\_TYPES.put("ACTION", "\\baction\\b");

TOKEN\_TYPES.put("START", "\\bstart\\b");

TOKEN\_TYPES.put("END", "\\bend\\b");

TOKEN\_TYPES.put("IDENTIFIER", "\\b[a-zA-Z\_][a-zA-Z0-9\_]\*\\b");

TOKEN\_TYPES.put("NUMBER", "\\b\\d+\\b");

TOKEN\_TYPES.put("STRING", "\"[^\"]\*\"|'[^']\*'");

TOKEN\_TYPES.put("SYMBOL", "[{}();,:]");

TOKEN\_TYPES.put("WHITESPACE", "\\s+");

int index = 1;

for (String key : TOKEN\_TYPES.keySet()) {

TOKEN\_INDEX.put(key, index++);

}

}

// Function to classify lexeme

public static String getTokenType(String lexeme) {

for (Map.Entry<String, String> entry : TOKEN\_TYPES.entrySet()) {

if (Pattern.matches(entry.getValue(), lexeme)) {

return entry.getKey();

}

}

return "UNKNOWN"; // Default if nothing matches

}

// Function to get token index

public static int getTokenIndex(String tokenType) {

return TOKEN\_INDEX.getOrDefault(tokenType, 1);

}

// Function to analyze a file

public static void lexicalAnalyzer(String filename) {

try (BufferedReader br = new BufferedReader(new FileReader(filename))) {

System.out.println(CYAN

+ " -----------------------------------------------------------------------------------------------"

+ RESET);

System.out.println(CYAN + "| Line | Lexeme | Token | Token Index |" + RESET);

System.out.println(CYAN

+ " -----------------------------------------------------------------------------------------------"

+ RESET);

String line;

int lineNo = 1;

List<String> parsedLines = new ArrayList<>();

while ((line = br.readLine()) != null) {

parsedLines.add(line);

Matcher matcher = Pattern

.compile("[a-zA-Z\_][a-zA-Z0-9\_]\*|\\d+|\"[^\"]\*\"|'[^']\*'|[{}();,:]")

.matcher(line);

while (matcher.find()) {

String lexeme = matcher.group();

String tokenType = getTokenType(lexeme);

int tokenIndex = getTokenIndex(tokenType);

if (!tokenType.equals("WHITESPACE")) { // Ignore whitespaces

System.out.printf(

CYAN + "| %-8d " + MAGENTA + "| %-32s " + GREEN + "| %-14s " + YELLOW + "| %-11d |\n"

+ RESET,

lineNo, lexeme, tokenType, tokenIndex);

}

}

lineNo++;

}

System.out.println(CYAN

+ " -----------------------------------------------------------------------------------------------"

+ RESET);

syntaxAnalyzer(parsedLines);

} catch (FileNotFoundException e) {

System.out.println(RED + "Error: File not found!" + RESET);

} catch (IOException e) {

System.out.println(RED + "Error reading file!" + RESET);

}

}

// Function to perform syntax analysis

public static void syntaxAnalyzer(List<String> lines) {

System.out.println(CYAN + "\n------------------ Syntax Analysis ------------------" + RESET);

int lineNo = 1;

for (String line : lines) {

if (isValidSyntax(line)) {

System.out.println(GREEN + "Line " + lineNo + ": Valid syntax [Valid]" + RESET);

displayParserTree(line);

} else {

System.out.println(RED + "Line " + lineNo + ": Syntax Error [Error]" + RESET);

identifySyntaxError(line);

}

lineNo++;

}

}

// Function to validate SMSL syntax

public static boolean isValidSyntax(String line) {

// Trim the line to remove leading/trailing whitespace

line = line.trim();

// Handle start and end keywords

if (line.equals("start;") || line.equals("end;")) {

return true;

}

// Handle state declarations

if (line.startsWith("state ") && line.endsWith("{")) {

return true;

}

// Handle transitions

if (line.startsWith("transition ") && line.contains(": event ") && line.contains("->") && line.endsWith("{")) {

return true;

}

// Handle actions

if (line.startsWith("action ") && line.endsWith(";")) {

return true;

}

// Handle opening and closing braces

if (line.equals("{") || line.equals("}")) {

return true;

}

return false; // Default: invalid syntax

}

// Function to identify syntax errors

public static void identifySyntaxError(String line) {

line = line.trim();

if (line.startsWith("state") && !line.endsWith("{")) {

System.out.println(RED + " ➤ Error: State declaration must end with '{'." + RESET);

} else if (line.startsWith("transition") && (!line.contains(": event ") || !line.contains("->") || !line.endsWith("{"))) {

System.out.println(RED + " ➤ Error: Transition must follow the format 'transition <name>: event <event> -> <state> {'." + RESET);

} else if (line.startsWith("action") && !line.endsWith(";")) {

System.out.println(RED + " ➤ Error: Action must end with ';'." + RESET);

} else if (line.equals("{") || line.equals("}")) {

System.out.println(RED + " ➤ Error: Misplaced or unmatched brace." + RESET);

} else {

System.out.println(RED + " ➤ Error: Unrecognized statement structure." + RESET);

}

}

// Function to display a hierarchical parse tree

public static void displayParserTree(String line) {

System.out.println(YELLOW + "\nParse Tree for: " + line + RESET);

System.out.println(YELLOW + "Root -> " + RESET);

if (line.startsWith("state")) {

String stateName = line.split("\\s+")[1];

System.out.println(YELLOW + "Node: State -> " + stateName + RESET);

} else if (line.startsWith("transition")) {

String[] parts = line.split("\\s\*:\\s\*|\\s\*->\\s\*");

String transitionName = parts[0].split("\\s+")[1];

String eventName = parts[1];

String targetState = parts[2];

System.out.println(YELLOW + "Node: Transition -> " + transitionName + RESET);

System.out.println(YELLOW + " Event -> " + eventName + RESET);

System.out.println(YELLOW + " Target State -> " + targetState + RESET);

} else if (line.startsWith("action")) {

String actionName = line.split("\\s+")[1];

System.out.println(YELLOW + "Node: Action -> " + actionName + RESET);

} else {

System.out.println(YELLOW + "Node: " + line.trim() + RESET);

}

}

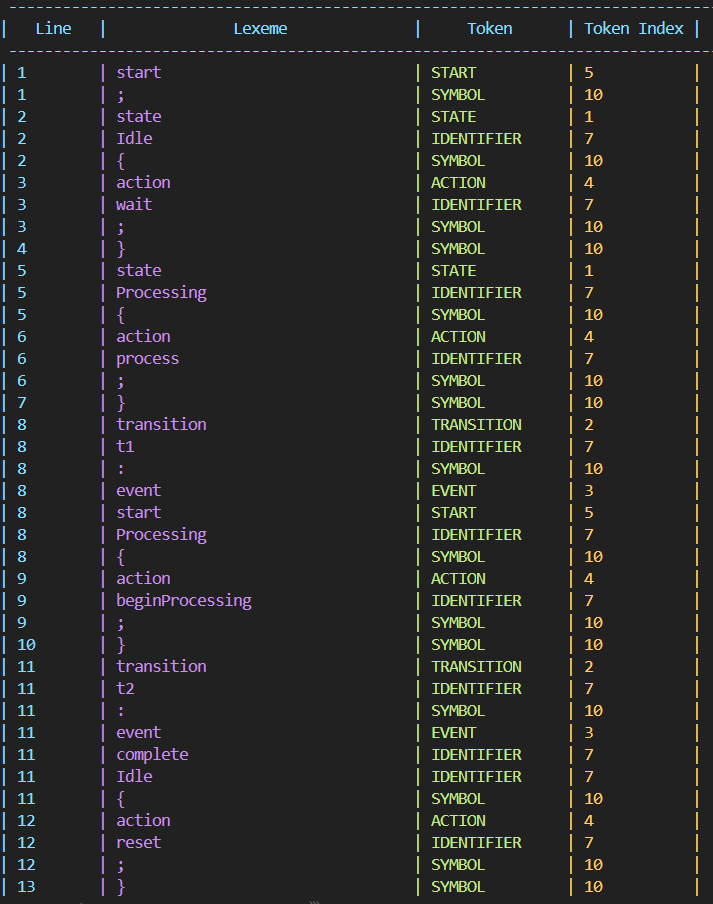
public static void main(String[] args) {

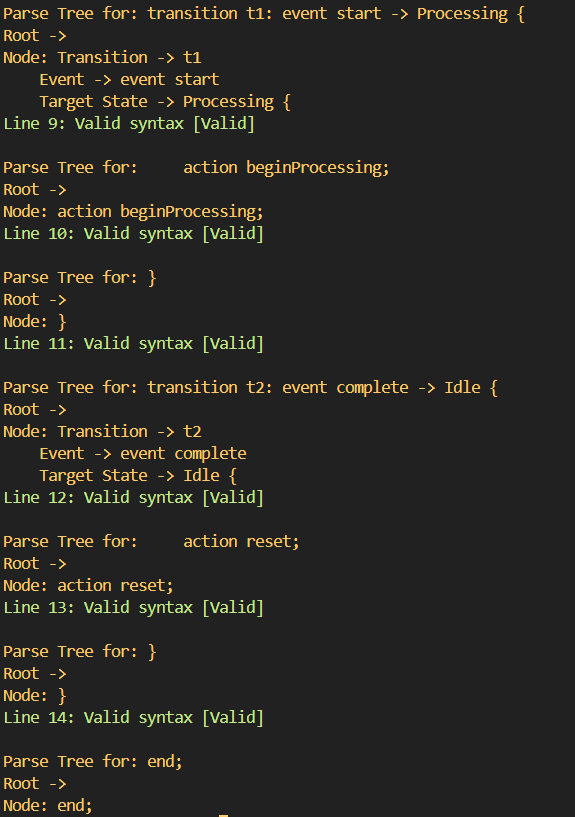
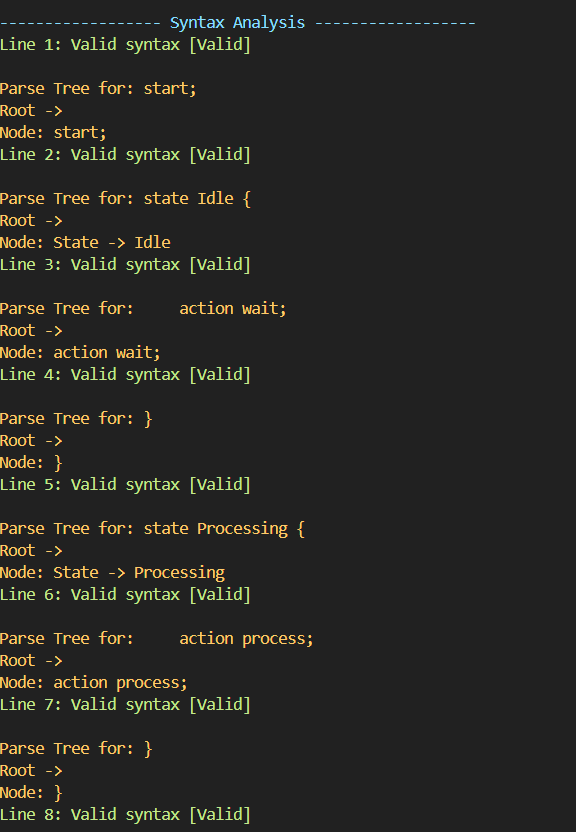
lexicalAnalyzer("smsl\_spec.txt");

}

}

**Output:**

****

****

**Conclusion**

In this project, I implemented lexical analysis and syntax parsing for processing a structured scripting language. The lexical analyzer efficiently tokenizes the input by identifying keywords, operators, identifiers, and literals, ensuring a well-defined token classification. The syntax analyzer validates the parsed tokens against predefined grammar rules, ensuring correct syntax structure for assignments, conditionals, loops, and function definitions.

A key aspect of this project was error detection, which improves reliability by identifying syntax errors such as missing operators, incorrect assignments, unclosed strings, and invalid command usage. By incorporating meaningful error messages and debugging support, the system provides clear feedback to users, making troubleshooting more effective.

Through this project, I have gained valuable insights into lexical and syntax analysis, grammar design, and error-handling techniques. This experience has enhanced my understanding of compiler design principles and structured language processing, equipping me with the skills to develop more robust parsing and validation systems in future projects.