Lexical and Syntax Analysis for Mathematical Expressions

1. Project Overview

This project focuses on implementing key stages of compiler design, specifically **Lexical Analysis** and **Syntax Analysis**, for mathematical expressions. The application will tokenize expressions, verify their syntactical correctness, and optionally construct Syntax Trees and Abstract Syntax Trees (AST). Through this process, students will gain a deeper understanding of compiler design concepts and their practical applications.

---

2. Objectives

The project has the following objectives:

- Develop an application to perform **Lexical Analysis** and **Syntax Analysis** on mathematical expressions.
- Clearly display outputs for both analysis phases.
- Generate derivation steps that demonstrate the application of grammar rules.
- Document the designed grammar and implementation steps in detail.
- Optionally generate Syntax Trees and Abstract Syntax Trees (AST) for bonus points.

---

3. Functional Requirements

3.1 Lexical Analysis

- Tokenize mathematical expressions into identifiable components such as:
    - Numbers (e.g., `123`, `4.56`)
    - Operators (`+`, `-`, `*`, `/`, `^`)
    - Parentheses (`(`, `)`)
- Produce a list of tokens with their types (e.g., "NUMBER", "OPERATOR", "PARENTHESIS").

3.2 Syntax Analysis

- Verify the correctness of the input expressions using a pre-defined grammar.
- Display **derivation steps**, showing how the input conforms to grammar rules.
- Report any syntactical errors with meaningful error messages (e.g., "Mismatched parentheses").

3.3 Grammar Design

- Document the grammar rules for supported mathematical expressions.
- Ensure the grammar handles:
    - Basic arithmetic operations (`+`, `-`, `*`, `/`)
    - Parentheses for operator precedence.
- For advanced functionality, extend grammar to support:
    - Exponentiation (`^`)
    - Factorial (`!`)
    - Trigonometric functions (`sin`, `cos`).

3.4 Syntax Trees and AST

- Syntax Trees: Represent the grammatical structure of an expression.
- Abstract Syntax Trees (AST): Simplify the representation, focusing on the logical structure.

---

4. Implementation Plan

4.1 Step-by-Step Process

1. **Lexical Analysis Phase**
    - Design a tokenizer to scan the input expression and generate tokens.

Example tokens for `3 + (5 * 2)`:

```
[  {"type": "NUMBER", "value": "3"},  {"type": "OPERATOR", "value": "+"},  {"type": "PARENTHESIS", "value": "("},  {"type": "NUMBER", "value": "5"},  {"type": "OPERATOR", "value": "*"},  {"type": "NUMBER", "value": "2"},  {"type": "PARENTHESIS", "value": ")"}]
```

2. **Syntax Analysis Phase**
    - Define grammar rules using **BNF (Backus-Naur Form)** or equivalent notation.
    - Implement a parser that verifies if the tokenized expression conforms to the grammar.
    - Display derivation steps (e.g., using **Leftmost Derivation**).
3. **Grammar Design**

Example grammar for basic operations:

```
<expression> ::= <term> | <term> "+" <expression> | <term> "-" <expression>
<term>       ::= <factor> | <factor> "*" <term> | <factor> "/" <term>
<factor>     ::= NUMBER | "(" <expression> ")"
```

Extend grammar for advanced features (optional):

```
<factor> ::= NUMBER | "(" <expression> ")" | <factor> "!" | "sin"
"(" <expression> ")"
```

4. **Optional Syntax Tree and AST Construction**
    - Implement a recursive structure to generate trees.
    - Example (for `3 + (5 * 2)`):
        - **Syntax Tree**: Represents the detailed grammar rules.

AST: Focuses on the hierarchical structure of operations:

```
    +
   / \
  3   *
     / \
    5   2
```

---

5. Testing Plan

- Prepare a set of valid and invalid test expressions to evaluate the application's functionality.
    - Valid expressions: `3 + (5 * 2)`, `sin(30) * 2`, `5! + 2^3`
    - Invalid expressions: `3 + * 5`, `sin30`, `2 / (3 - )`
- Ensure outputs for Lexical and Syntax Analysis are accurate.
- Check if error handling correctly identifies and reports issues.

---

6. Programming Language Selection

You can choose any programming language; however, consider the following:

- **Python**: Easier for implementing tokenization and parsing with libraries like `re` (for Lexical Analysis) and `ply` (for Syntax Analysis).

---

7. Documentation

The final project report should include:

1. **Introduction**: Overview of the project and its objectives.
2. **Grammar Definition**: A clear explanation of grammar rules and their role.
3. **Design Process**: Step-by-step description of Lexical and Syntax Analysis implementation.
4. **Testing Results**: Screenshots or text outputs of the application with test cases.
5. **Conclusion**: Reflection on challenges faced and lessons learned.

8. Evaluation Criteria

1. **Minimum Requirements (70 Points)**
   - Tokenization of basic expressions.
   - Syntax validation using the defined grammar.
   - Display of derivation steps.
2. **Advanced Features (Bonus Points)**
   - Support for exponentiation, factorial, and trigonometric functions.
   - Construction of Syntax Trees and AST.
3. **Documentation (Mandatory)**
   - Clear and detailed explanations of the grammar and implementation steps.
4. **Presentation**
   - Demonstrate the application functionality on the submission date.

---

9. Timeline

- **Research Phase**: Learn about Lexical and Syntax Analysis concepts (1 day).
- **Implementation Phase**: Develop and test Lexical and Syntax Analysis (2–3 days).
- **Optional Features**: Add advanced functionalities (1 day).
- **Documentation and Finalization**: Complete the project report and prepare for the presentation (1 day).

---

10. Conclusion

This project is an excellent opportunity to bridge theoretical concepts with practical application. By completing it, students will gain a solid understanding of key compiler design stages and develop essential problem-solving and programming skills.

Student Name: Ismail Seker

Number: 201504027