Compiler Design

Documentation for Expression Evaluator

This documentation describes the implementation of an expression evaluation system designed to process and evaluate mathematical expressions. The system includes lexical analysis (tokenization), syntax analysis (parsing), and expression evaluation. The grammar rules, functionality, and key phases are outlined below.

Grammar Definition

The system uses the following context-free grammar for parsing expressions:

- **Expression Grammar**:

```
E -> E + T | E - T | T
T -> T * F | T / F | F ^ F | F
F -> F ! | sin ( E ) | cos ( E ) | ( E ) | number
```

Explanation of Grammar Rules:

- **E (Expression)**: Represents addition and subtraction operations. An expression can also be a term (T).
- **T (Term)**: Represents multiplication, division, and exponentiation operations. A term can also be a factor (F).
- **F (Factor)**: Represents more granular elements such as numbers, factorials, trigonometric functions (`sin`, `cos`), and parenthesized subexpressions.

This grammar supports basic arithmetic operators, parentheses, trigonometric functions, exponentiation, and factorial operations.

Implementation Overview

1. Lexical Analysis (Tokenization)

Purpose:

To convert the input mathematical expression (a string) into a sequence of tokens, where each token represents a meaningful component of the expression, such as numbers, operators, or functions.

Key Classes and Functions:

- **Class:** `Token`
  - Represents individual tokens with the following attributes:
    - `type`: The type of the token (e.g., `NUMBER`, `OPERATOR`, `FUNCTION`).
    - `value`: The actual value of the token (e.g., `+`, `sin`, `42`).
    - `children`: An array reserved for future use in tree representation.
- **Class:** `Tokens`
  - A doubly linked list for managing tokens.
  - Provides methods:
    - `push(token)`: Adds a token to the list.
    - `traverse()`: Converts the token list into an array for inspection.
- **Function:** `tokenize(expression)`
  - Scans the input expression using predefined regular expressions for each token type.
  - Token types include:
    - `NUMBER`: Integers or decimals.
    - `OPERATOR`: Arithmetic operators (`+`, `-`, `*`, `/`).
    - `PARENTHESIS`: Open and close parentheses (`(`, `)`).
    - `POWER`: Exponentiation operator (`^`).
    - `FACTORIAL`: Factorial operator (`!`).
    - `FUNCTION`: Trigonometric functions (`sin`, `cos`).
    - Whitespace is ignored.

Process:

- The `tokenize` function iterates over the expression, matching substrings with predefined patterns.
- If a match is found, the token is added to the token list.
- If no match is found for a character, an error is thrown.

---

2. Syntax Analysis (Parsing)

Purpose:

To validate the structure of the token sequence based on the defined grammar and construct a parse tree for evaluation.

Key Classes and Functions:

- **Class:** `ParseTreeNode`
  - Represents a node in the parse tree.
  - Attributes:
    - `type`: The type of the node (e.g., `NUMBER`, `OPERATOR`).
    - `value`: The value of the node.
    - `children`: An array representing child nodes.
- **Function:** `syntaxAnalyze(tokens)`
  - Parses the token sequence and builds a parse tree based on the grammar rules.
  - Uses recursive-descent parsing with the following functions:
    - `parseExpression`: Handles addition and subtraction.
    - `parseTerm`: Handles multiplication, division, and exponentiation.
    - `parseFactor`: Handles numbers, functions, parentheses, and factorials.
  - Validates the sequence of tokens and ensures that parentheses are properly matched.

Process:

- The parser starts with the head of the token list and recursively builds the parse tree by following the grammar rules.

- For example:
  - An expression like `3 + 5 * 2` is parsed into a tree where `+` is the root, `3` is the left child, and ` ` (with `5` and `2` as its children) is the right child.

---

3. Expression Evaluation

Purpose:

To compute the result of the expression by traversing the parse tree.

Key Functions:

- **Function:** `evaluate(node)`
  - Traverses the parse tree recursively to compute the value of the expression.
  - Handles:
    - Numbers: Directly converts them to floating-point values.
    - Functions (`sin`, `cos`): Computes trigonometric values of the child node.
    - Factorials: Computes the factorial of the child node's value.
    - Operators (`+`, ` `, ` `, `/`, `^`): Applies the respective operations to the values of the child nodes.
- **Helper Function:** `factorial(n)`
  - Recursively computes the factorial of a non-negative integer.

Process:

- Starting from the root of the parse tree, the `evaluate` function computes the result by recursively processing all child nodes.

Example

This section explains step by step how the expression `3 + 2 * 5` is evaluated using the grammar functions `E`, `T`, and `F`.

Step 1: Tokenizing the Input Expression

The `tokenize` function splits the input expression into tokens:

```
3 + 2 * 5
```

The tokens are as follows:

1. `3` → NUMBER
2. `+` → OPERATOR
3. `2` → NUMBER
4. ` ` → OPERATOR
5. `5` → NUMBER

Step 2: Parsing Phase

In this phase, the `syntaxAnalyze` function uses the grammar rules defined in `E`, `T`, and `F` to parse the expression.

Grammar Rules:

- `E -> E + T | E - T | T`
- `T -> T * F | T / F | F ^ F | F`
- `F -> F ! | sin ( E ) | cos ( E ) | ( E ) | number`

Parsing Process

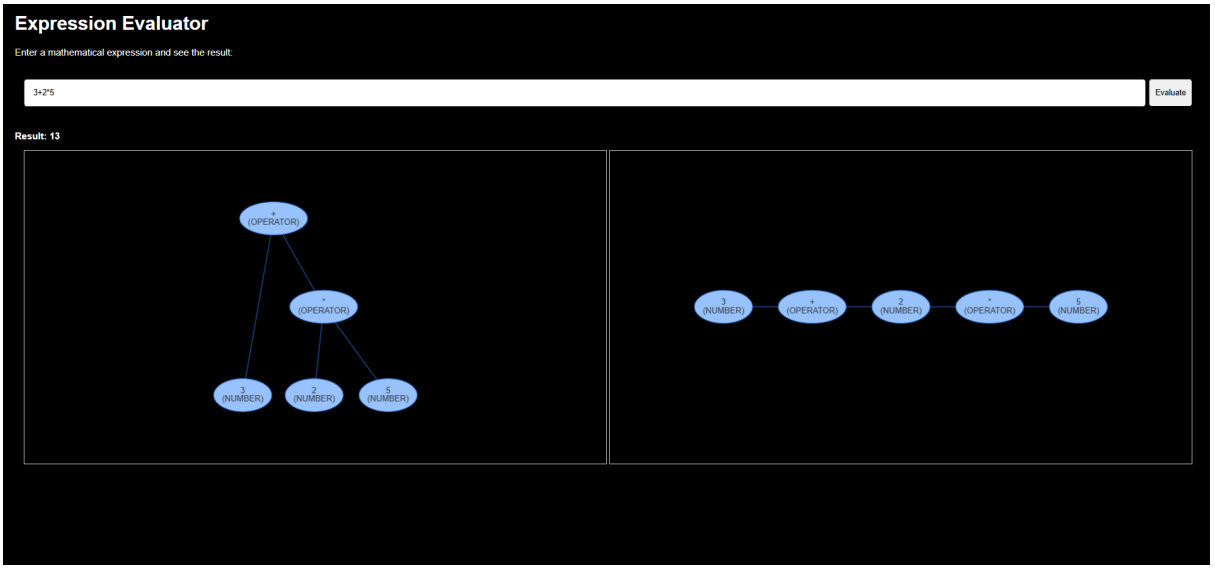
1. **Start (E):**
   - The `E` function is called and begins by calling `parseTerm()` (`T`).
2. **First Term (T):**
   - `parseTerm()` is called and starts with `parseFactor()` (`F`).
   - `parseFactor()` matches `3` (NUMBER) from the token stream and creates an `F` node.
3. **Operator +:**
   - The `E` function sees that the next token is `+` (OPERATOR) and calls `parseTerm()` (`T`) to evaluate the right-hand side.
4. **Second Term (T):**
   - `parseTerm()` is called again and starts with `parseFactor()` (`F`).

- `parseFactor()` matches `2` (NUMBER) from the token stream and creates an `F` node.
- `T` sees that the next token is ` ` (OPERATOR) and calls `parseFactor()` again.

5. **Third Term (F):**
   - `parseFactor()` matches `5` (NUMBER) from the token stream and creates an `F` node.
   - `T` combines these two `F` nodes (`2` and `5`) with the ` ` operator to form a `T` node.
6. **Result (E):**
   - `E` combines the earlier `F` node (`3`) and the `T` node (`2 * 5`) with the `+` operator to form an `E` node.

Parse Tree

The resulting parse tree is structured as follows:



Step 3: Evaluation Phase

The `evaluate` function traverses the parse tree in a depth-first manner to compute the result:

1. The ` ` operation (`2 * 5`) is computed first → Result: `10`.
2. The `+` operation (`3 + 10`) is computed next → Result: `13`.

Final Result: The result of the expression is `13`.

Visualization

For visualizing the token sequence and parse tree, the **vis.js** library is used. It provides a graphical representation of the structure of tokens and the hierarchical relationships in the parse tree.