FACULTY OF COMPUTER SCIENCE AND ENGINEERING.

Parser With GUI

Instructor: Abrar

Prepared by:

Ehsan Elahi
Ammar Bin Farrukh

Repo:

https://github.com/RaoEhsanElahi/Parser_with_GUI

https://github.com/ammar313-cs/compiler_construction_lab_oel/tree/main

MILESTONE 1

Grammar Rules:

The expression evaluator adheres to the following grammar rules:

- **Expressions:** Composed of **terms** separated by addition (+) or subtraction (-) operators.
- **Terms:** Composed of **factors** multiplied (*) or divided (/).
- **Factors:** Can be either numbers (decimals) or expressions enclosed in parentheses (**parentheses nesting** is allowed).
- **Numbers:** Sequences of digits, which can be positive, negative, or contain a decimal point.
- **Operators:** Supported operators include addition (+), subtraction (-), multiplication (*), and division (/). All operators are binary (operate on two operands).
- **Parentheses:** Used for grouping expressions to alter the order of operations.
- **Whitespace:** Spaces, tabs, and newlines are ignored during evaluation.
- **Comments:** Not currently supported.

Design Choices:

- **Simplicity and Efficiency:** The parser focuses on evaluating basic arithmetic expressions without complex parsing capabilities for advanced language features.
- **Recursive Descent Parsing:** The parser employs a recursive approach to handle nested expressions within parentheses. It breaks down the expression into smaller sub-expressions for evaluation.
- **Operator Precedence (Future):** The current implementation evaluates expressions from left to right. Future enhancements will incorporate rules for operator precedence (e.g., multiplication before addition).
- **Error Handling:** The parser raises exceptions to indicate invalid tokens, syntax errors, or division by zero.

MILESTONE 2 AND 3

Implementation Details:

1. Tokenizer Function (`tokenize(input_string)`)

- This function takes an expression string as input and breaks it down into a list of tokens.
- Each token is a tuple containing two elements:
  - **Type:** Denotes the token category (e.g., 'NUMBER', 'OPERATOR', 'PAREN').
  - **Value:** Represents the actual content of the token (e.g., the numerical value for a number, the operator symbol for an operator).
- The tokenizer utilizes regular expressions to match different token types in the input string.
- Whitespace characters (spaces, tabs, newlines) are ignored during tokenization.
- If an unrecognizable character sequence is encountered, a `ValueError` exception is raised.

2. Parser Class (`Parser`)

- The parser object is responsible for evaluating the expression based on the provided list of tokens.
- It maintains a pointer to the current token being processed within the token list.
- **`advance(self)`:** This method moves the pointer forward to the next token in the list.
- **`factor(self)`:** This method parses factors, which can be numbers or parenthesized expressions.
  - It checks if the current token represents a number and returns its float value after advancing the pointer.
  - If it encounters an opening parenthesis ("("), it performs the following steps:
    - Recursively calls `expression` to evaluate the nested expression within the parentheses.
    - Ensures a closing parenthesis (")") follows the expression.
    - Returns the evaluated result from the nested expression.
  - If none of the above conditions are met, it raises a `ValueError` for an invalid factor.
- **`term(self)`:** This method parses terms, which involve products or divisions of factors.
  - It calls `factor` to get the initial value and stores it in a variable.
  - It iterates as long as the current token is a multiplication or division operator.
    - It calls `factor` again to obtain the next factor's value.
    - Based on the encountered operator, it performs the multiplication or division operation and updates the stored value.
  - The final evaluated term value is returned.
- **`expression(self)`:** This method parses the entire expression, which consists of sums or differences of terms.
  - It calls `term` to get the initial value and stores it in a variable.
  - It iterates as long as the current token is an addition or subtraction operator.
    - It calls `term` again to obtain the next term's value.

        - Based on the encountered operator, it performs the addition or subtraction operation and updates the stored value.
    - The final evaluated expression value is returned.

Usage Instructions:

- Import the `tokenize` function and the `Parser` class from the relevant module.
- Use the `tokenize` function to convert your expression string into a list of tokens.
- Initialize a `Parser` object with the list of tokens.
- Call the `expression` method of the parser object to evaluate the expression.
- The `expression` method returns the calculated result.

Input Handling:

The parser assumes the input is a valid expression string written according to the defined grammar. It does not perform any validation or error correction on the input string itself beyond basic tokenization checks.

Variable Storage:

The parser does not handle variable storage or memory management. It operates on the provided expression string and calculates the final result.

Algebraic Operations:

Currently, the parser supports basic binary arithmetic operations: addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`).

Data Types:

The parser currently only recognizes numbers (decimals) as valid factors within expressions. It does not handle other data types like strings, booleans, or complex numbers.

GUI Components:

The provided code includes a basic GUI using PySimpleGUI. The user enters an expression in the input text box, clicks the "Evaluate" button, and the calculated result is displayed.

Assumptions:

- The input string is a well-formed expression without syntax errors.
- All tokens within the expression are valid according to the defined grammar.