Arithmetic Expression Parser

Group Members:

Shahryar Mubashar 2020447

Sameer Arif Khan 2020430

<u>Compiler Lab CS424</u>

Course Instructor:

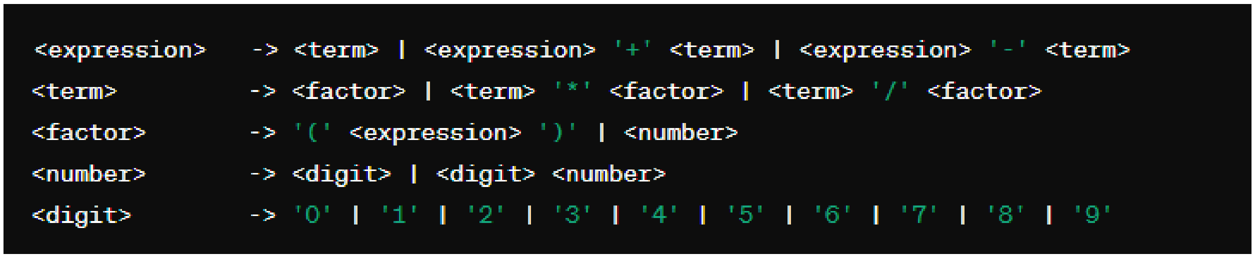
Sir Abrar

Introduction:

Arithmetic expression parsing is a fundamental task in computer science, with applications in various domains such as compilers, calculators, and symbolic computation. This report presents a comprehensive overview of an arithmetic expression parser developed through multiple milestones, including grammar specification, input handling, lexical analysis, and parsing algorithm implementation.

Context-Free Grammar (CFG):

The Context-Free Grammar (CFG) defines the syntax of the arithmetic expressions accepted by the parser. The grammar rules are as follows:



```
<expression>   -> <term> | <expression> '+' <term> | <expression> '-' <term>
<term>         -> <factor> | <term> '*' <factor> | <term> '/' <factor>
<factor>       -> '(' <expression> ')' | <number>
<number>       -> <digit> | <digit> <number>
<digit>        -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

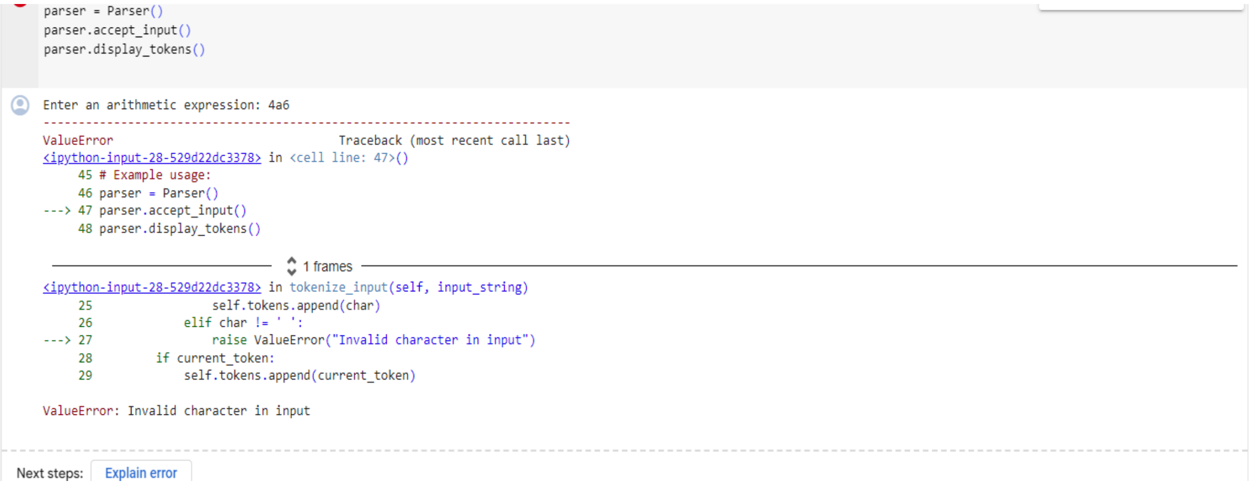
Grammar Rules:

1. Expression Rule: Represents an arithmetic expression composed of terms with addition or subtraction operators.
2. Term Rule: Represents a term composed of factors with multiplication or division operators.
3. Factor Rule: Represents a factor, which can be a subexpression or a number.
4. Number Rule: Represents a number composed of digits.
5. Digit Rule: Represents a single digit.

Milestone 1:

Understanding the Grammar and Implementing Input Handling

- Overview: This milestone focuses on understanding the grammar of the language and implementing basic input handling functionalities in the parser.
- Deliverables:
  - Documentation of the grammar rules with detailed explanations and examples.
  - Implementation of input handling functionality to tokenize the input string based on the grammar rules.



```
parser = Parser()
parser.accept_input()
parser.display_tokens()
```

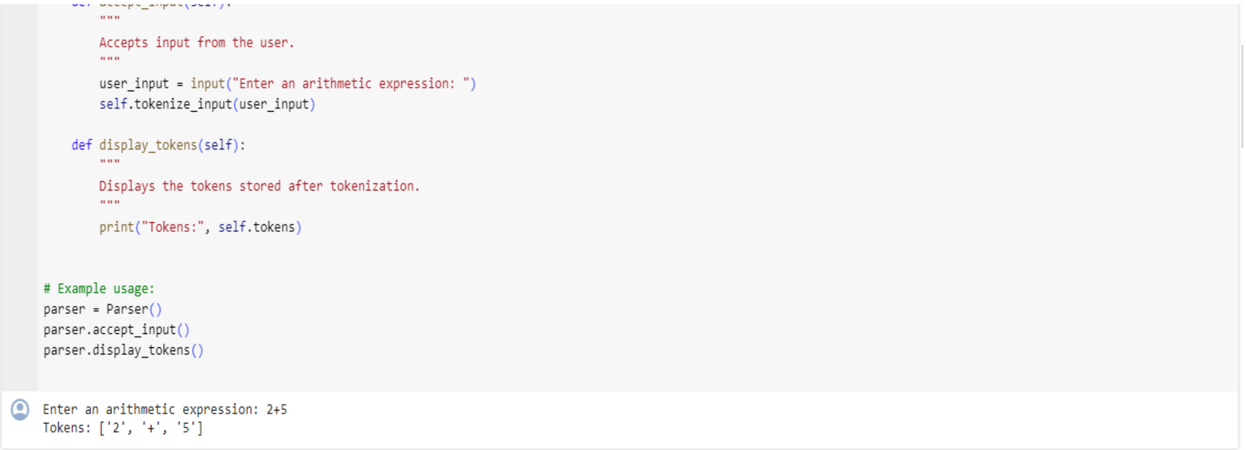
```
Enter an arithmetic expression: 4a6
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-529d22dc3378> in <cell line: 47>()
     45 # Example usage:
     46 parser = Parser()
---> 47 parser.accept_input()
     48 parser.display_tokens()

                              1 frames
<ipython-input-28-529d22dc3378> in tokenize_input(self, input_string)
     25                 self.tokens.append(char)
     26             elif char != ' ':
---> 27                 raise ValueError("Invalid character in input")
     28         if current_token:
     29             self.tokens.append(current_token)

ValueError: Invalid character in input
```

Input Handling (Milestone 1)

Milestone 2:

Lexical Analysis (Tokenization)

- Overview: This milestone involves implementing a lexical analyzer to break input strings into tokens based on grammar rules.
- Deliverables:
    - Implementation of a lexical analyzer that tokenizes input strings using regular expressions.
    - Handling whitespace, comments, and irrelevant characters gracefully.



```
    def accept_input(self):
        """
        Accepts input from the user.
        """
        user_input = input("Enter an arithmetic expression: ")
        self.tokenize_input(user_input)

    def display_tokens(self):
        """
        Displays the tokens stored after tokenization.
        """
        print("Tokens:", self.tokens)


# Example usage:
parser = Parser()
parser.accept_input()
parser.display_tokens()
```

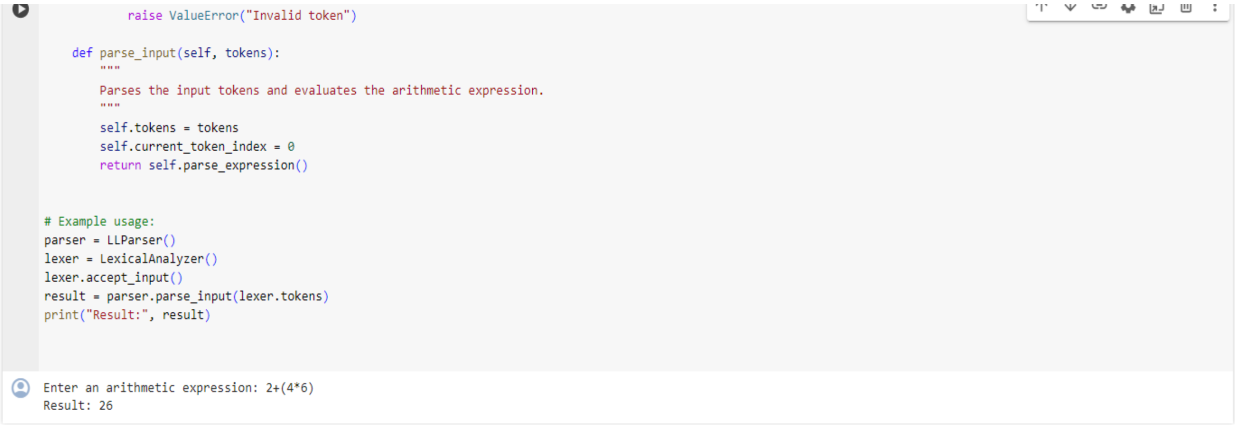
```
Enter an arithmetic expression: 2+5
Tokens: ['2', '+', '5']
```

Milestone 2

Milestone 3:

Parsing Algorithm

- Overview: This milestone focuses on choosing a parsing algorithm and implementing it using object-oriented design principles.
- Deliverables:
    - Selection of LL(1) parsing algorithm suitable for the grammar complexity.
    - Implementation of parsing algorithm using recursive descent approach.
    - Methods for parsing input tokens and constructing parse tree or generating parse results.



```
            raise ValueError("Invalid token")

    def parse_input(self, tokens):
        """
        Parses the input tokens and evaluates the arithmetic expression.
        """
        self.tokens = tokens
        self.current_token_index = 0
        return self.parse_expression()


# Example usage:
parser = LLParser()
lexer = LexicalAnalyzer()
lexer.accept_input()
result = parser.parse_input(lexer.tokens)
print("Result:", result)
```

```
Enter an arithmetic expression: 2+(4*6)
Result: 26
```

LL(1) Parser Milestone 3

Conclusion:

Arithmetic expression parsing is a foundational concept in computer science, and the development of an arithmetic expression parser requires understanding of grammar rules, lexical analysis, and parsing algorithms. Through the milestones outlined in this report, a complete parser has been developed capable of parsing and evaluating arithmetic expressions according to the specified grammar. This project serves as a valuable learning experience in language processing and compiler design concepts.