Compiler or Interpreter: Create a basic compiler or interpreter for a simple programming language, demonstrating knowledge of lexical analysis and parsing.

INTRODUCTION:

This assihmnet entails the creation of a basic Python compiler/interpreter in C++, emphasizing the integral phases of lexical analysis and parsing. Leveraging C++ for its efficiency and versatility, the implementation involves a lexer to tokenize the Python source code and a parser to construct the Abstract Syntax Tree (AST). The AST serves as an intermediate representation, facilitating subsequent code generation and interpretation. The primary objective is to showcase a robust comprehension of compiler design principles, grammar recognition, and the seamless coordination of lexical and syntactic analysis. Through this endeavor, we aim to contribute insights into language processing, providing a foundation for future exploration in the realms of compiler construction and system-level programming.

STEPS:

1)In the first step I have add the necessary header which are required for my assignment, those are **<iostream>** which is used for input and output operations, and **<cctype>** which is used for character classification functions like **isspace** and **isdigit**.

2)In the next step I had written a line **using namespace as std; ,** which brings the entire **std** (Standard C++ Library) namespace into scope, allowing the use of standard C++ functions and objects without the **std::** prefix.

3)In next step I had defined a enumeration(means user defined datatype) called **TokenType,** which consists of **INTEGER**{Represents an integer literal in the arithmetic expression.}**, PLUS**{Represents the addition operator}**, MINUS**{Represents the subtraction operator}**, MULTIPLY**{Represents the multiplication operator}**, DIVIDE**{Represents the division operator }**, LPAREN**{Represents a left parenthesis }**, RPAREN**{Represents a right parenthesis}**, EOF_TOKEN**{Stands for "End of File Token" and might be used to signify the end of the input stream or expression.}

4)in the next step I had defined a stuct(user defined datatype that allows to group different types of variabbles under a single name) named **Token,** which consists of **TokenType** which define enum and **value** which is of datatype integer.

5)Then I created a class named **Lexer** where then lexer(a component of a compiler or interpreter that breaks down the source code into a sequence of tokens) is initialized with the input text, and it sets the initial position (**pos**) to 0 and the current character (**current_char**) to the first character in the input text. Then In private access specifier I have created data memers named **text**(string), **pos**(size_t), **current_char**(char).In the public access specifier I have initialized several member functions:

- **void error()** : This function called when the lexer encounters an invalid character. It throws a **runtime_error** with the message "Invalid character".
- **void advance()** : This function advances the lexer to the next character in the input text. If the end of the input text is reached, it sets **current_char** to '\0' to indicate the end of the input.
- **void skip_whitespace()** : This function skips over whitespace characters until a non-whitespace character is encountered.
- **int integer()** : This function extracts a sequence of digits from the input and converts them into an integer.
- **Token get_next_token()** : This function is the main driver for extracting tokens from the input. It uses a series of **if** statements to check the current character and return the corresponding token. If none of the expected characters is found, it calls the **error** method.

6)In the next step I created another class named **Paser,** where The constructor initializes the parser with a reference to a **Lexer** object. It also initializes the **current_token** member variable by getting the first token from the lexer using **lexer.get_next_token()**. The private members has a reference to the lexer (**Lexer& lexer**) and the current token (**Token current_token**). The lexer is used to obtain tokens, and the current token keeps track of the parser's position in the token stream. In the public members I have initialized several member functions:

- **void error()** : The function is called when a syntax error is encountered.
- **void eat(TokenType expected_type)** : This function is used to consume tokens. It checks if the type of the **current_token** matches the expected type. If it does, the function advances to the next token by calling **lexer.get_next_token()**. If there is a type mismatch, it calls the **error** function.
- **int factor()** : This function parses a factor in the expression. It can be either an integer or an expression enclosed in parentheses.
- **int term()** : This function parses a term in the expression, which is a sequence of factors separated by multiplication or division operators.
- **int expr()** : This function parses an entire expression, which is a sequence of terms separated by addition or subtraction operators.
- **int parse()** : This function initiates the parsing process by calling the **expr** function. It returns the result of parsing the entire expression.

7)the main function for a basic Python-like compiler or interpreter designed for handling arithmetic operations. It prompts the user for input, creates instances of a Lexer and a Parser to analyze and parse the input, attempts to perform the parsing, and prints the result to the console if successful. In case of a parsing error, it catches the exception, displays an error message, and exits with a non-zero return code. The program follows a structure typical for a simple interactive interpreter, where the user inputs arithmetic expressions, and the program evaluates and outputs the result.

CONCLUSION/SUMMARY:

This C++ assignment implements a basic Python-like compiler/interpreter for arithmetic expressions. The code comprises a **lexer** and **parser**, collectively enabling the conversion of user-inputted arithmetic expressions into executable code. The **lexer** tokenizes the input, recognizing integers, arithmetic operators, and parentheses, while the recursive descent parser interprets the syntax and executes the arithmetic operations following standard precedence rules. The program includes error-handling mechanisms for identifying invalid characters, syntax errors, and division by zero. The user interacts with the program through a console interface, entering expressions in a Python-like format. Although the current implementation is tailored for educational purposes and simplicity, there is potential for expansion to support more complex language features in the future.