

**Simple Arithmetic Expression Compiler**



**Group Members**

Hanood (021-21-0001)

Parkash (021-21-0067)

**Simple Arithmetic Expression Compiler**

**Introduction:**

The purpose of this project was to design and implement a compiler capable of parsing and evaluating simple arithmetic expressions. This compiler was designed to accept input expressions containing arithmetic operators such as addition, subtraction, multiplication, and division. Additionally, it supports the use of parentheses to control operator precedence. The final output of the compiler is the computed result of the given arithmetic expression.

**Project Goals:**

The primary goal of this project was to design a lexer that tokenizes input arithmetic expressions. This involved specifying token types for numbers, operators, and parentheses, and using regular expressions to match each token type. Furthermore, the project aimed to implement a parser that generates an Abstract Syntax Tree (AST) from these tokens. The AST represents the hierarchical structure of the arithmetic expression. The next goal was to develop an evaluator to traverse the AST and compute the result of the arithmetic expression. Finally, the project included testing the compiler with various input expressions to ensure its correctness and reliability.

**Implementation:**

**Lexer (Tokenization):**

The lexer was responsible for breaking down the input arithmetic expression into a sequence of tokens. This was accomplished by defining token types for numbers, operators, and parentheses, and specifying regular expressions for each token type. The lexer function processed the input string, matched the regular expressions, and converted the input into a sequence of tokens, ignoring any whitespace.

**Parser (AST Generation):**

The parser's role was to convert the sequence of tokens generated by the lexer into an Abstract Syntax Tree (AST). The AST consists of nodes representing numbers and binary operations. To achieve this, classes were designed to represent these nodes. The parser class was implemented to construct the AST from the token sequence, with methods to handle the parsing of expressions, terms, and factors. This ensured that the hierarchical structure of the arithmetic expression was accurately represented.

**Evaluator (Expression Evaluation):**

The evaluator's purpose was to traverse the AST and compute the result of the arithmetic expression. This involved designing an evaluator class with methods to visit different types of nodes in the AST. The evaluator implemented logic to recursively evaluate subexpressions, ensuring that the arithmetic operations were performed correctly according to the structure of the AST.

**Compilation Process:**

The compilation process integrated the lexer, parser, and evaluator into a cohesive workflow. A main function was created to drive this process, which involved tokenizing the input expression, parsing the tokens into an AST, and evaluating the AST to compute the result. This function ensured that the entire process was automated and seamless, from input to output.

**Results:**

The compiler was tested with a variety of input expressions to validate its functionality. It successfully parsed and evaluated simple arithmetic expressions containing addition, subtraction, multiplication, division, and parentheses. The compiler produced correct results for all tested expressions, demonstrating its accuracy and reliability. The results confirmed that the compiler met the project goals and performed as expected.

**Conclusion:**

The project achieved its objective of creating a compiler capable of parsing and evaluating simple arithmetic expressions. By implementing a lexer, parser, and evaluator, the project demonstrated a fundamental understanding of the compilation process. The successful development and testing of the compiler showcased the ability to apply theoretical concepts to practical implementations. Future enhancements could include adding support for additional operators, improving error handling, and implementing optimization techniques.

**References:**

The project was guided by course materials and textbooks on compiler design and programming languages. Online resources and tutorials provided valuable insights into the implementation of lexers, parsers, and evaluators. Collaborative discussions and peer feedback during the project development process also contributed to the successful completion of the project.

**Appendix:**

The appendix includes source code file for the lexer, parser, and evaluator (main.py, history.txt) as well as test scripts used for testing the compiler. Sample input expressions and the corresponding output results obtained during testing are also provided. This documentation ensures that the project is well-documented and reproducible for future reference and development.

Overall, this project was a valuable learning experience that provided deep insights into compiler construction and language processing techniques. It successfully demonstrated the practical application of theoretical concepts, laying a solid foundation for future endeavors in software development and computer science.