Advanced Calculator with Variable Storage and Syntax Error Reporting

Course Title: Compiler Construction

Instructor Name: Sir Shahbaz Qadeer

Semester: Spring 2024

Group Member:

M.Faran Malik	**F2021266559**
Esha Aslam	**F2021266597**

University of Management and Technology (UMT)

C-II Johar Town, Lahore 54770, Pakistan

Table of Contents

1. Introduction

This project report provides an in-depth analysis of a Python-based calculator designed to handle basic arithmetic operations and variable assignments. The program processes input either directly from the user or from a file, tokenizes the input, parses it into an abstract syntax tree (AST), evaluates the AST, and stores the results. This document details the code, explains each component, discusses error handling, and offers insights into potential future enhancements.

2. Project Objectives

The primary objectives of this project are:

- To develop a robust calculator that supports basic arithmetic operations (+, -, *, /).
- To enable variable assignments and usage within expressions.
- To provide a user-friendly interface for entering expressions and managing calculations.
- To ensure persistence of variable values across sessions through file operations.
- To handle errors gracefully and provide meaningful feedback to users.

3. Overview of the Code

The calculator comprises several classes and functions, each responsible for a specific part of the process:

- **Tokenizer**: Converts input strings into tokens.
- **Parser**: Parses tokens into an AST.
- **Evaluator**: Evaluates the AST.
- **SymbolTable**: Manages variable assignments.
- **process_expression**: Integrates tokenization, parsing, and evaluation.
- **main**: The entry point of the program, handling user interaction and file operations.

The following sections provide a detailed explanation of each component, its purpose, and how it works.

4. Detailed Explanation of Each Component

Tokenizer Class

The `Tokenizer` class is responsible for converting input strings into a list of tokens. It uses regular expressions to identify different types of tokens, such as numbers, operators, parentheses, identifiers, and assignment operators.

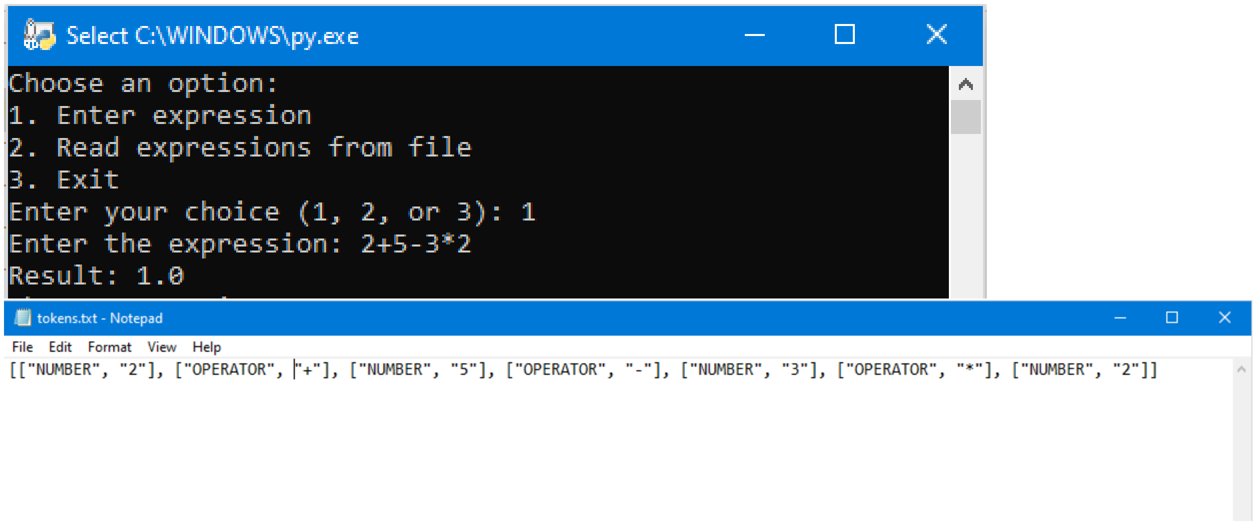
Code:

```
import re
import json

class Tokenizer:
    token_patterns = {
        'NUMBER': r'\d+(\.\d+)?',
        'OPERATOR': r'[+\-*/]',
        'PAREN': r'[()]',
        'IDENTIFIER': r'[a-zA-Z]\w*',
        'ASSIGN': r'='
    }

    def __init__(self):
        self.token_regex = re.compile('|'.join(f'(?P<{name}>{pattern})' for
name, pattern in self.token_patterns.items()))

    def tokenize(self, input_string):
        tokens = []
        for match in self.token_regex.finditer(input_string):
            token_type = match.lastgroup
            token_value = match.group(token_type)
            tokens.append((token_type, token_value))
        return tokens

    def tokenize_to_file(self, input_string, filename):
        tokens = self.tokenize(input_string)
        with open(filename, 'w') as file:
            file.write(json.dumps(tokens))
        return tokens
```

Explanation:

- **Token Patterns**: The `token_patterns` dictionary maps token types to their corresponding regular expression patterns.
    - `NUMBER`: Matches integer and floating-point numbers.
    - `OPERATOR`: Matches arithmetic operators (+, -, *, /).
    - `PAREN`: Matches parentheses ((),)).
    - `IDENTIFIER`: Matches variable names (e.g., x, var1).
    - `ASSIGN`: Matches the assignment operator (=).
- **Initialization**: The `__init__` method compiles the regular expressions into a single regex pattern.

- **Tokenization**: The `tokenize` method scans the input string for matches, categorizes them into tokens, and returns a list of tokens.
- **File Tokenization**: The `tokenize_to_file` method tokenizes the input string and writes the tokens to a specified file.



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: 2+5-3*2
Result: 1.0
```

```
[["NUMBER", "2"], ["OPERATOR", "+"], ["NUMBER", "5"], ["OPERATOR", "-"], ["NUMBER", "3"], ["OPERATOR", "*"], ["NUMBER", "2"]]
```

Parser Class

The `Parser` class converts a list of tokens into an AST, enabling the program to understand and evaluate expressions.

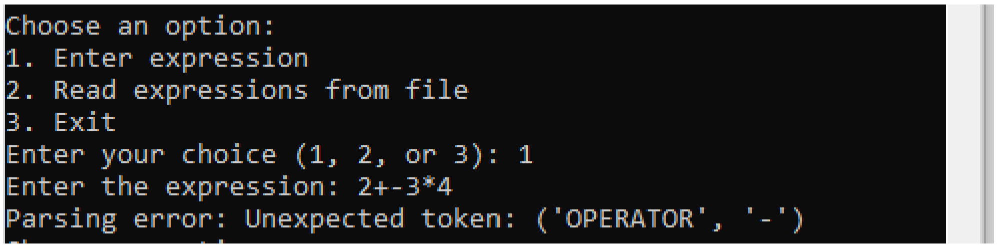
Code:

```
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0
        self.error = None

    def peek(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos]
        return None

    def consume(self):
        self.pos += 1

    def parse_expression(self):
        try:
            node = self.parse_term()
            while self.peek() and self.peek()[1] in '+-':
                token = self.peek()
                self.consume()
                right = self.parse_term()
                node = ('bin_op', token[1], node, right)
            return node
        except SyntaxError as se:
            self.error = str(se)
            return None

    def parse_term(self):
        node = self.parse_factor()
        while self.peek() and self.peek()[1] in '*/':
            token = self.peek()
            self.consume()
            right = self.parse_factor()
            node = ('bin_op', token[1], node, right)
        return node

    def parse_factor(self):
        token = self.peek()
        if token[0] == 'NUMBER':
            self.consume()
            return ('number', token[1])
        elif token[0] == 'IDENTIFIER':
            self.consume()
            if self.peek() and self.peek()[0] == 'ASSIGN':
                self.consume()
                value = self.parse_expression()
                return ('assign', token[1], value)
            else:
                return ('identifier', token[1])
        elif token[1] == '(':
            self.consume()
            expr = self.parse_expression()
            if self.peek() and self.peek()[1] == ')':
                self.consume()
                return expr
            else:
                raise SyntaxError("Missing closing parenthesis")
        else:
            raise SyntaxError(f"Unexpected token: {token}")
```

Explanation:

- **Initialization**: The `__init__` method initializes the parser with a list of tokens, setting the current position to 0 and the error attribute to None.
- **Peeking and Consuming Tokens**:
  - `peek`: Returns the current token without advancing the position.
  - `consume`: Advances to the next token.
- **Parsing Expressions**:
  - `parse_expression`: Parses terms separated by `+` or `-` operators.

    - `parse_term`: Parses factors separated by `*` or `/` operators.
    - `parse_factor`: Parses numbers, identifiers, assignments, and parenthesized expressions.
- **Error Handling**: Raises `SyntaxError` for unexpected tokens or missing parentheses.



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: 2+-3*4
Parsing error: Unexpected token: ('OPERATOR', '-')
```

Evaluator Class

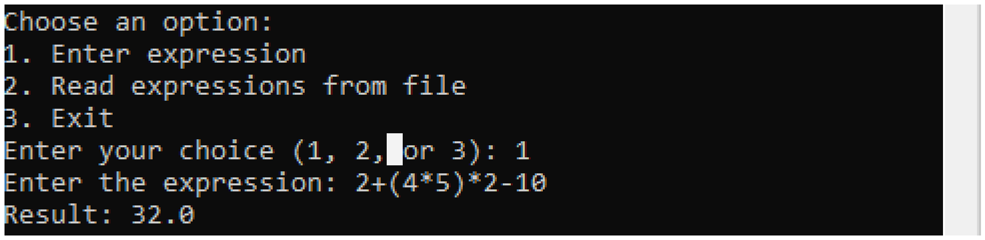
The `Evaluator` class evaluates the AST, performing the necessary arithmetic operations and managing variable assignments.

Code:

```
class Evaluator:
    def __init__(self, symbol_table):
        self.symbol_table = symbol_table

    def evaluate(self, node):
        if node[0] == 'number':
            return float(node[1])
        elif node[0] == 'identifier':
            return self.symbol_table.get(node[1])
        elif node[0] == 'assign':
            value = self.evaluate(node[2])
            self.symbol_table.set(node[1], value)
            return value
        elif node[0] == 'bin_op':
            left = self.evaluate(node[2])
            right = self.evaluate(node[3])
            if node[1] == '+':
                return left + right
            elif node[1] == '-':
                return left - right
            elif node[1] == '*':
                return left * right
            elif node[1] == '/':
                if right == 0:
                    raise ValueError("Division by zero")
                return left / right
```

Explanation:

- **Initialization**: The `__init__` method initializes the evaluator with a symbol table.
- **Evaluating Nodes**:
  - `number`: Converts the token value to a float.
  - `identifier`: Retrieves the value from the symbol table.
  - `assign`: Evaluates the right-hand side and assigns the value to the identifier.
  - `bin_op`: Evaluates the left and right operands and performs the arithmetic operation.
- **Error Handling**: Raises `ValueError` for division by zero.



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: 2+(4*5)*2-10
Result: 32.0
```

SymbolTable Class

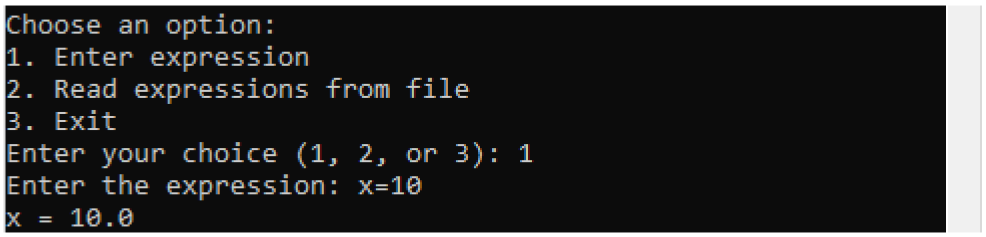
The `SymbolTable` class manages variable assignments and stores their values, ensuring persistence across sessions.

Code:

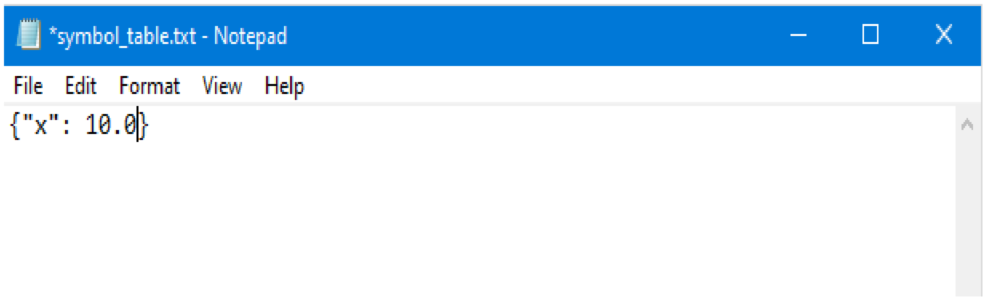
```
class SymbolTable:
    def __init__(self, filename):
        self.filename = filename
        self.symbols = {}
        self.load()

    def load(self):
        if os.path.exists(self.filename):
            with open(self.filename, 'r') as file:
                self.symbols = json.load(file)

    def save(self):
        with open(self.filename, 'w') as file:
            json.dump(self.symbols, file)

    def get(self, name):
        return self.symbols.get(name, 0)

    def set(self, name, value):
        self.symbols[name] = value
        self.save()
```

Explanation:

- **Initialization**: The `__init__` method initializes the symbol table with a filename and loads existing symbols from the file.
- **Loading Symbols**: The `load` method reads symbols from a file if it exists.
- **Saving Symbols**: The `save` method writes symbols to a file.
- **Getting and Setting Symbols**:
    - `get`: Retrieves the value of a symbol, defaulting to 0 if not found.
    - `set`: Sets the value of a symbol and saves the table.



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: x=10
x = 10.0
```



```
{"x": 10.0}
```

Process Expression Function

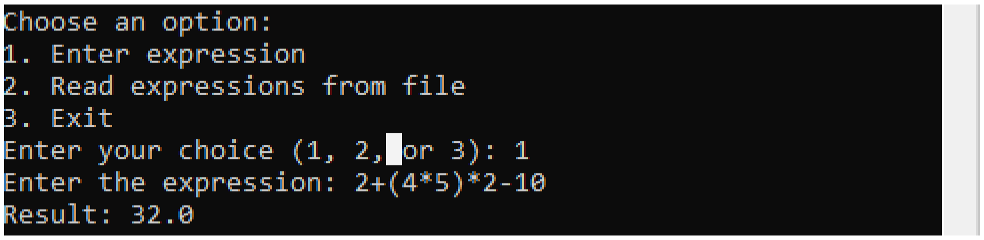
The `process_expression` function integrates tokenization, parsing, and evaluation into a single process, handling errors and returning results.

Code:

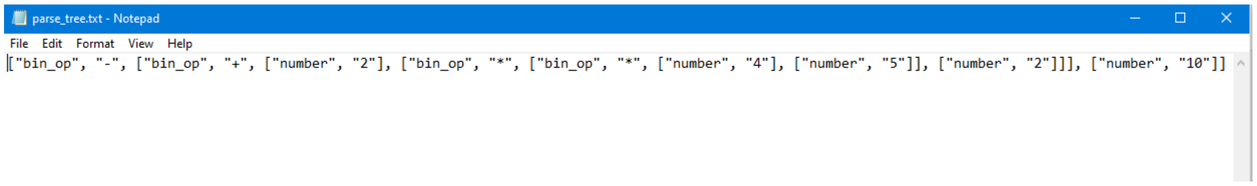
```
def process_expression(input_string, tokenizer, symbol_table,
error_file='errors.txt'):
    try:
        tokens = tokenizer.tokenize(input_string)
        parser = Parser(tokens)
        parse_tree = parser.parse_expression()
        if parser.error:
            with open(error_file, 'w') as file:
                file.write(f"Parsing error: {parser.error}\n")
            return f"Parsing error: {parser.error}"

        evaluator = Evaluator(symbol_table)
        result = evaluator.evaluate(parse_tree)
        return f"Result: {result}"
    except Exception as e:
        with open(error_file, 'w') as file:
            file.write(f"Evaluation error: {e}\n")
        return f"Evaluation error: {e}"
```

Explanation:

- **Tokenization**: Converts the input string into tokens using the tokenizer.
- **Parsing**: Converts tokens into an AST using the parser.
- **Evaluation**: Evaluates the AST using the evaluator.
- **Error Handling**: Catches and writes errors to a file, returning an error message.



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: 2+(4*5)*2-10
Result: 32.0
```



```
["bin_op", "-", ["bin_op", "+", ["number", "2"], ["bin_op", "*", ["bin_op", "*", ["number", "4"], ["number", "5"]], ["number", "2"]]], ["number", "10"]]
```

Main Function

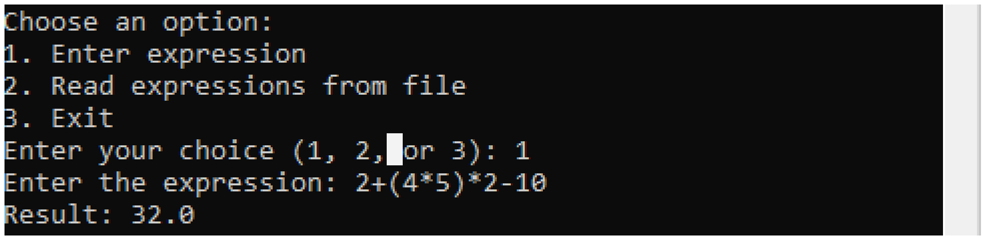
The `main` function handles user interaction, providing options for direct input or file input, and manages the overall program flow.

Code:

```
def main():
    # Setup file names
    symbol_table_file = "symbol_table.txt"
    output_file = "results.txt"

    # Initialize classes
    tokenizer = Tokenizer()
    symbol_table = SymbolTable(symbol_table_file)

    while True:
        # Input options
        print("Choose an option:")
        print("1. Enter expression")
        print("2. Read expressions from file")
        print("3. Exit")
        input_source = input("Enter your choice (1, 2, or 3): ")

        if input_source == '1':
            input_string = input("Enter the expression: ")
            result = process_expression(input_string, tokenizer,
symbol_table)
            print(result)
        elif input_source == '2':
            input_filename = input("Enter the input file name: ")
            try:
                with open(input_filename, 'r') as infile:
                    expressions = infile.readlines()

                with open(output_file, 'w') as outfile:
                    for expression in expressions:
                        expression = expression.strip()
                        if expression:
                            print(f"Processing: {expression}")
                            result = process_expression(expression,
tokenizer, symbol_table)
                            print(result)
                            outfile.write(f"{expression}: {result}\n")
                print(f"Results have been written to {output_file}")
            except FileNotFoundError:
                print("File not found.")
        elif input_source == '3':
            print("Exiting...")
            break
        else:
            print("Invalid input source.")

if __name__ == "__main__":
    main()
```

Explanation:

- **Initialization**: Initializes the tokenizer and symbol table.
- **User Input Options**: Prompts the user to enter an expression directly or read from a file.
- **Processing Expressions**:
    - Direct Input: Processes a single expression entered by the user.
    - File Input: Reads and processes multiple expressions from a specified file, writing results to an output file.
- **Error Handling**: Catches file not found errors and invalid input options.
- **Exit Handling**: Provides an option to exit the program.



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: 2+(4*5)*2-10
Result: 32.0
```

5. Flow of Execution

The flow of execution in the calculator program follows these steps:

1. **Initialization**: The `main` function initializes the tokenizer and symbol table.
2. **User Input**:
   - The user selects between entering an expression directly or reading from a file.
   - Based on the user's choice, the program either prompts for an expression or reads expressions from a file.
3. **Processing**:
   - The `process_expression` function tokenizes the input string, parses the tokens into an AST, and evaluates the AST.
   - Results are either displayed to the user or written to an output file.
4. **Loop**: The process repeats, allowing the user to enter more expressions or read from another file until they choose to exit.

6. Error Handling

Effective error handling is crucial for a robust calculator program. This section details the various error handling mechanisms implemented in the code:

Syntax Errors

Syntax errors occur when the input string contains invalid syntax, such as mismatched parentheses or unexpected tokens. The parser raises `SyntaxError` exceptions for such cases.

Example:

```
if token[1] == '(':
    self.consume()
    expr = self.parse_expression()
    if self.peek() and self.peek()[1] == ')':
        self.consume()
        return expr
    else:
        raise SyntaxError("Missing closing parenthesis")
else:
    raise SyntaxError(f"Unexpected token: {token}")
```

Evaluation Errors

Evaluation errors occur during the evaluation of the AST, such as division by zero. The evaluator raises `ValueError` exceptions for such cases.

Example:

```
if node[1] == '/':
    if right == 0:
        raise ValueError("Division by zero")
    return left / right
```

File Errors

File errors occur during file operations, such as when a file is not found. The main function catches `FileNotFoundError` exceptions and provides feedback to the user.

Example:

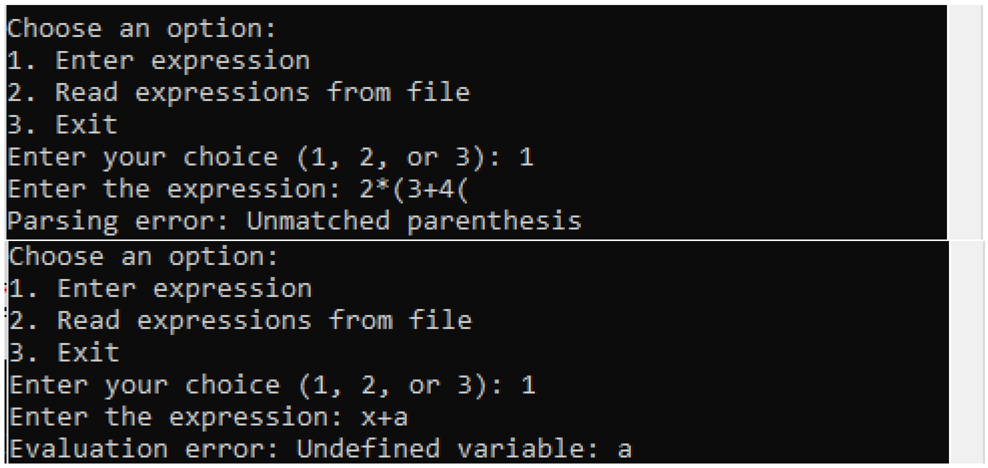
```
try:
    with open(input_filename, 'r') as infile:
        expressions = infile.readlines()
except FileNotFoundError:
    print("File not found.")
```

General Error Handling

The `process_expression` function includes a try-except block to catch any exceptions that occur during tokenization, parsing, or evaluation. Errors are written to a file, and an error message is returned.

Example:

```
try:
    tokens = tokenizer.tokenize(input_string)
    parser = Parser(tokens)
    parse_tree = parser.parse_expression()
    if parser.error:
        with open(error_file, 'w') as file:
            file.write(f"Parsing error: {parser.error}\n")
        return f"Parsing error: {parser.error}"

    evaluator = Evaluator(symbol_table)
    result = evaluator.evaluate(parse_tree)
    return f"Result: {result}"
except Exception as e:
    with open(error_file, 'w') as file:
        file.write(f"Evaluation error: {e}\n")
    return f"Evaluation error: {e}"
```



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: 2*(3+4(
Parsing error: Unmatched parenthesis
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the expression: x+a
Evaluation error: Undefined variable: a
```

7. File Operations

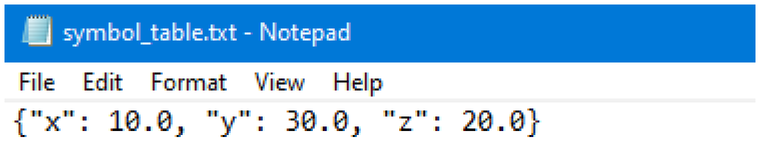
File operations are an essential part of this calculator program, enabling persistence of variable values and handling input/output from files.

Symbol Table File

The symbol table is saved to and loaded from a file to ensure that variable assignments persist across sessions.

Example:

```
class SymbolTable:
    def __init__(self, filename):
        self.filename = filename
        self.symbols = {}
        self.load()

    def load(self):
        if os.path.exists(self.filename):
            with open(self.filename, 'r') as file:
                self.symbols = json.load(file)
    def save(self):
        with open(self.filename, 'w') as file:
            json.dump(self.symbols, file)
```



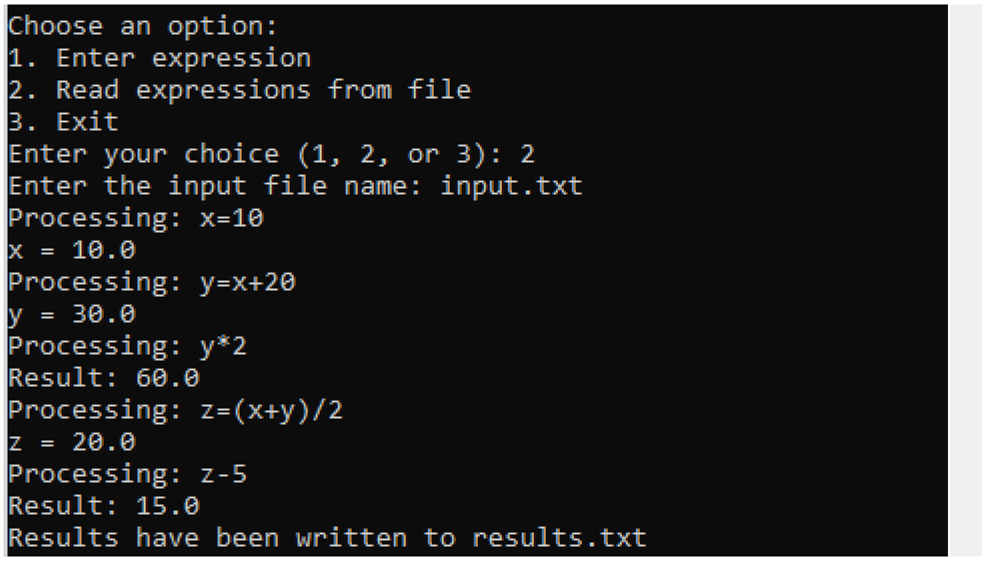
```
{"x": 10.0, "y": 30.0, "z": 20.0}
```

Input and Output Files

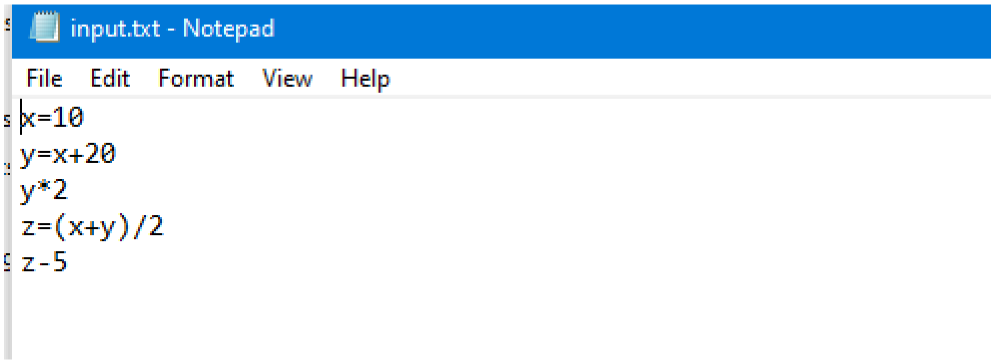
The program supports reading expressions from an input file and writing results to an output file. This feature is particularly useful for batch processing of multiple expressions.

Example:

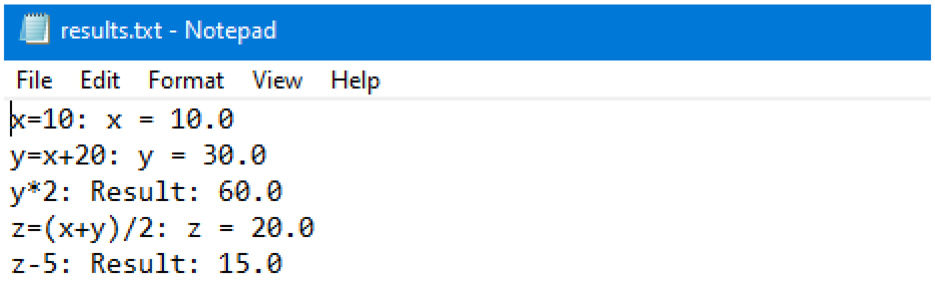
```
try:
    with open(input_filename, 'r') as infile:
        expressions = infile.readlines()

    with open(output_file, 'w') as outfile:
        for expression in expressions:
            expression = expression.strip()
            if expression:
                print(f"Processing: {expression}")
                result = process_expression(expression, tokenizer, 
symbol_table)
                print(result)
                outfile.write(f"{expression}: {result}\n")
    print(f"Results have been written to {output_file}")
except FileNotFoundError:
    print("File not found.")
```



```
Choose an option:
1. Enter expression
2. Read expressions from file
3. Exit
Enter your choice (1, 2, or 3): 2
Enter the input file name: input.txt
Processing: x=10
x = 10.0
Processing: y=x+20
y = 30.0
Processing: y*2
Result: 60.0
Processing: z=(x+y)/2
z = 20.0
Processing: z-5
Result: 15.0
Results have been written to results.txt
```



```
x=10
y=x+20
y*2
z=(x+y)/2
z-5
```



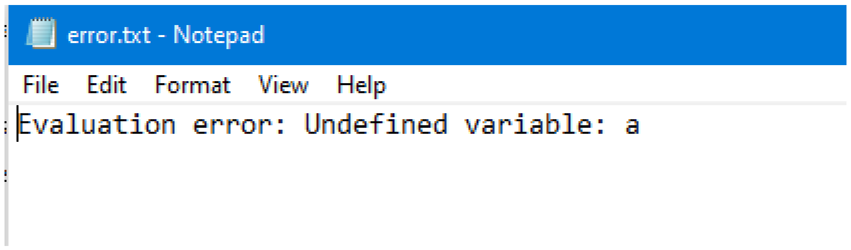
```
x=10: x = 10.0
y=x+20: y = 30.0
y*2: Result: 60.0
z=(x+y)/2: z = 20.0
z-5: Result: 15.0
```

Error File

Errors encountered during parsing or evaluation are written to a specified error file, providing a log for debugging and analysis.

Example:

```
except Exception as e:
    with open(error_file, 'w') as file:
        file.write(f"Evaluation error: {e}\n")
    return f"Evaluation error: {e}"
```



```
Evaluation error: Undefined variable: a
```

8. User Interaction

The main function provides a user-friendly interface for interacting with the calculator. Users can choose to enter expressions directly or process expressions from a file.

Direct Input

Users can enter expressions directly, which are then processed and the results displayed immediately.

Example:

```
if input_source == '1':
    input_string = input("Enter the expression: ")
    result = process_expression(input_string, tokenizer, symbol_table)
    print(result)
```

File Input

Users can specify an input file containing multiple expressions. The program processes each expression and writes the results to an output file.

Example:

```
elif input_source == '2':
    input_filename = input("Enter the input file name: ")
    try:
        with open(input_filename, 'r') as infile:
            expressions = infile.readlines()

        with open(output_file, 'w') as outfile:
            for expression in expressions:
                expression = expression.strip()
                if expression:
                    print(f"Processing: {expression}")
                    result = process_expression(expression, tokenizer,
symbol_table)
                    print(result)
                    outfile.write(f"{expression}: {result}\n")
        print(f"Results have been written to {output_file}")
    except FileNotFoundError:
        print("File not found.")
```

Exiting the Program

Users can exit the program by selecting the appropriate option, which breaks the loop and ends the execution.

Example:

```
elif input_source == '3':
    print("Exiting...")
    break
else:
    print("Invalid input source.")
```

9. Performance Considerations

The performance of the calculator program is influenced by several factors, including the efficiency of the tokenizer, parser, and evaluator, as well as the handling of file operations. This section discusses key performance considerations and optimizations.

Tokenization Efficiency

The `Tokenizer` class uses regular expressions to identify tokens in the input string. Regular expressions are efficient for pattern matching, but complex expressions can impact performance. Optimizing regex patterns and minimizing the number of regex operations can improve tokenization speed.

Parsing Efficiency

The `Parser` class converts tokens into an AST using recursive descent parsing. This approach is suitable for simple arithmetic expressions but can become inefficient for deeply nested expressions. Optimizing the parsing logic and minimizing recursive calls can enhance performance.

Evaluation Efficiency

The `Evaluator` class evaluates the AST, performing arithmetic operations and managing variable assignments. Efficient evaluation requires minimizing redundant calculations and optimizing arithmetic operations. Memoization and caching techniques can be used to improve performance for repeated expressions.

File Operations

File operations, such as reading and writing symbols and expressions, can impact performance, especially for large files. Using buffered I/O operations and optimizing file access patterns can reduce overhead and improve performance.

Scalability

The program is designed to handle individual expressions and batch processing of multiple expressions. Scalability considerations include optimizing the handling of large input files, minimizing memory usage, and ensuring efficient management of the symbol table.

10. Conclusion

This project report provides a comprehensive overview of a Python-based calculator designed to handle basic arithmetic operations and variable assignments. The program's modular design, robust error handling, and file operations ensure maintainability and extensibility. The detailed explanations of each component, along with the flow of execution, error handling, and file operations, provide a thorough understanding of the code. The testing and validation strategies ensure the correctness and reliability of the program. Future enhancements offer potential avenues for improving functionality and user experience.

11. References

- Python Documentation: https://docs.python.org/3/
- Regular Expressions: https://docs.python.org/3/library/re.html
- JSON Module: https://docs.python.org/3/library/json.html
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.
- Flex: Fast Lexical Analyzer Generator
- Python Regular Expressions