PROJECT REPORT

Compiler Construction Lab (CSL-323)

PROJECT TITLE:

SYNTAX ANALYZER (PARSER)

BS (CS) – 5B

Group Members

Name	Enrollment
1. **Mohammad Soban**	02-134222-010
2. **Zeyad Baloch**	02-134222-020
3. **Sakeena Yamin**	02-134222-048

Submitted to:

Ma'am Mehwish Saleem

BAHRIA UNIVERSITY KARACHI CAMPUS

Department of Computer Science

Abstract

The Syntax Analyzer (Parser) project implements a comprehensive solution for analyzing programming code syntax using Python and PLY (Python Lex-Yacc). This implementation focuses on both lexical and syntactic analysis of source code, providing visual feedback through parse tree generation. The project integrates a graphical user interface built with PyQt5, enabling users to interact with the analysis tools effectively. The system is designed to assist students and developers in understanding compiler design fundamentals through practical application and visual representation of parsing processes.

TABLE OF CONTENTS

Introduction

Syntax analysis, also known as parsing, is a crucial phase in compiler construction that determines whether source code follows the correct syntactic structure according to the programming language's grammar rules. This project implements a syntax analyzer that combines lexical analysis, syntactic parsing, and visual representation of parse trees to create a comprehensive learning and development tool.

The implementation leverages modern technologies including:

- Python as the primary programming language
- PLY (Python Lex-Yacc) for lexical and syntactic analysis
- PyQt5 for the graphical user interface
- Networkx and Matplotlib for parse tree visualization

Problem statement

The development of compiler construction concepts presents significant challenges for students and developers in understanding the theoretical and practical aspects of syntax analysis. There is a need for an interactive tool that can:

- Demonstrate the process of lexical and syntactic analysis in real-time
- Provide visual representation of parse trees for better understanding
- Offer immediate feedback on syntax errors with detailed explanations
- Present a user-friendly interface for code analysis and learning

Methodology

The project follows a modular approach with three main components:

1. **Lexical Analysis Module**
    - Implementation of token definitions and patterns
    - Development of lexical analyzer using PLY's lex module
    - Error handling and reporting during tokenization
    - Token stream generation for parser input
2. **Syntactic Analysis Module**
    - Definition of grammar rules using PLY's yacc module
    - Implementation of syntax checking mechanisms
    - Error detection and reporting with line number references
    - Parse tree construction during analysis
3. **Visualization and Interface Module**
    - Development of PyQt5-based graphical interface
    - Integration of parse tree visualization using networkx
    - Implementation of interactive features for code input and analysis
    - Styling and theme implementation (dark orange and black)

Project Scope

The project encompasses the following key aspects:

1. **Code Analysis Capabilities**
    - Support for basic programming language constructs
    - Lexical analysis with comprehensive token recognition
    - Syntax verification based on defined grammar rules
    - Detailed error reporting and suggestions
2. **Visualization Features**
    - Interactive parse tree display
    - Token stream visualization
    - Error highlighting in source code
    - Real-time analysis feedback
3. **User Interface**
    - Code editor with syntax highlighting
    - Multiple view panels for different outputs
    - Tree visualization controls
    - Error and warning display area

Code

(GUI FOLDER):

COMPONENTS.PY

```
from PyQt5.QtWidgets import (QFrame, QVBoxLayout, QHBoxLayout,
                             QTextEdit, QPushButton, QLabel,
                             QTabWidget, QTableWidget, QWidget)
from PyQt5.QtGui import QFont

def create_input_section(window):
    """Create the input section of the GUI"""
    input_frame = QFrame()
    input_frame.setFrameStyle(QFrame.StyledPanel)
    input_layout = QVBoxLayout(input_frame)

    input_label = QLabel("Enter Code:")
    input_label.setFont(QFont("Arial", 12, QFont.Bold))

    window.input_text = QTextEdit()
    window.input_text.setPlaceholderText("Example: x = 2 * (3 + 4)")
    window.input_text.setFixedHeight(150)

    button_layout = QHBoxLayout()
    window.analyze_btn = QPushButton("Analyze Code")
    window.analyze_btn.clicked.connect(window.analyze_code)
    window.clear_btn = QPushButton("Clear")
    window.clear_btn.clicked.connect(window.clear_all)
    window.exit_btn = QPushButton("Exit")
    window.exit_btn.clicked.connect(window.close)

    button_layout.addWidget(window.analyze_btn)
    button_layout.addWidget(window.clear_btn)
    button_layout.addWidget(window.exit_btn)

    input_layout.addWidget(input_label)
    input_layout.addWidget(window.input_text)
    input_layout.addLayout(button_layout)

    return input_frame

def create_results_section(window):
    """Create the results section of the GUI"""
    results_tab = QTabWidget()

    # Token analysis tab
    window.token_table = QTableWidget()
    window.token_table.setColumnCount(4)
    window.token_table.setHorizontalHeaderLabels(["Type", "Value",
"Position", "Line"])
    results_tab.addTab(window.token_table, "Token Analysis")

    # Parse tree tab
    window.tree_widget = QWidget()
```

```
    window.tree_layout = QVBoxLayout(window.tree_widget)
    results_tab.addTab(window.tree_widget, "Parse Tree")

    # Error tab
    window.error_text = QTextEdit()
    window.error_text.setReadOnly(True)
    results_tab.addTab(window.error_text, "Errors")

    return results_tab
```

MAIN_WINDOW.PY

```
from PyQt5.QtWidgets import QMainWindow, QWidget, QVBoxLayout
from PyQt5.QtGui import QFont
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as
FigureCanvas
import matplotlib.pyplot as plt

from lexer.lexer import Lexer
from parser.parser import Parser
from .components import create_input_section, create_results_section
from .styles import apply_styles
from .tree_visualizer import ParseTreeVisualizer

class SyntaxAnalyzerGUI(QMainWindow):
    def __init__(self):
        super().__init__()
        self.lexer = Lexer()
        self.parser = Parser()
        self.visualizer = ParseTreeVisualizer()
        self.setup_ui()

    def setup_ui(self):
        """Initialize the user interface"""
        self.setWindowTitle("Advanced Code Analyzer")
        self.setGeometry(100, 100, 1200, 800)

        main_widget = QWidget()
        self.setCentralWidget(main_widget)
        layout = QVBoxLayout(main_widget)

        # Create UI components
        input_frame = create_input_section(self)
        self.results_tab = create_results_section(self)

        layout.addWidget(input_frame)
        layout.addWidget(self.results_tab)

        apply_styles(self)

    def analyze_code(self):
        """Analyze the input code and display results"""
        code = self.input_text.toPlainText().strip()
        if not code:
            self.show_error_message("Please enter code to analyze.")
            return
```

```
        self.clear_results()

        try:
            # Perform lexical analysis
            tokens = self.lexer.tokenize(code)
            self.update_token_table(tokens)

            if self.lexer.errors:
                self.show_errors(self.lexer.errors)
                return

            # Perform syntax analysis
            parse_tree = self.parser.parse(tokens)

            if self.parser.errors:
                self.show_errors(self.parser.errors)
            elif parse_tree:
                self.display_parse_tree(parse_tree)

        except Exception as e:
            self.show_errors([f"Error during analysis: {str(e)}"])

    def update_token_table(self, tokens):
        """Update the token analysis table"""
        from PyQt5.QtWidgets import QTableWidgetItem
        self.token_table.setRowCount(len(tokens))
        for i, token in enumerate(tokens):
            self.token_table.setItem(i, 0, QTableWidgetItem(token.type))
            self.token_table.setItem(i, 1,
QTableWidgetItem(str(token.value)))
            self.token_table.setItem(i, 2,
QTableWidgetItem(str(token.position)))
            self.token_table.setItem(i, 3, QTableWidgetItem(str(token.line)))
        self.token_table.resizeColumnsToContents()

    def show_errors(self, errors):
        """Display errors in the error tab"""
        self.error_text.setText("\n".join(errors))
        self.results_tab.setCurrentWidget(self.error_text)

    def show_error_message(self, message):
        """Display an error message dialog"""
        from PyQt5.QtWidgets import QMessageBox
        QMessageBox.warning(self, "Error", message)

    def display_parse_tree(self, parse_tree):
        """Display the parse tree visualization"""
        figure = self.visualizer.visualize(parse_tree)
        if figure:
            canvas = FigureCanvas(figure)
            self.tree_layout.addWidget(canvas)
            self.results_tab.setCurrentWidget(self.tree_widget)

    def clear_results(self):
        """Clear all result displays"""
        self.token_table.setRowCount(0)
```

```
        self.error_text.clear()

        for i in reversed(range(self.tree_layout.count())):
            widget = self.tree_layout.itemAt(i).widget()
            if isinstance(widget, FigureCanvas):
                plt.close(widget.figure)
            if widget:
                widget.setParent(None)
                widget.deleteLater()

    def clear_all(self):
        """Clear all input and results"""
        self.input_text.clear()
        self.clear_results()

    def closeEvent(self, event):
        """Handle window close event"""
        self.clear_results()
        super().closeEvent(event)
```

STYLES.PY

```
def apply_styles(window):
    """Apply styles to GUI components"""
    # Input styling
    window.input_text.setStyleSheet("""
        QTextEdit {
            background-color: #2c3e50;
            color: #ecf0f1;
            border: 2px solid #3498db;
            border-radius: 5px;
            padding: 10px;
            font-family: 'Consolas', monospace;
            font-size: 14px;
        }
    """)

    # Button styling
    button_style = """
        QPushButton {
            background-color: #3498db;
            color: white;
            border: none;
            border-radius: 4px;
            padding: 10px 20px;
            font-weight: bold;
            min-width: 120px;
            font-size: 12px;
        }
        QPushButton:hover {
            background-color: #2980b9;
        }
        QPushButton:pressed {
            background-color: #2472a4;
        }
    """
```

```
    window.analyze_btn.setStyleSheet(button_style)
    window.clear_btn.setStyleSheet(button_style)
    window.exit_btn.setStyleSheet(button_style)

    # Table styling
    window.token_table.setStyleSheet("""
        QTableWidget {
            background-color: #2c3e50;
            color: #ecf0f1;
            gridline-color: #34495e;
            border: none;
        }
        QHeaderView::section {
            background-color: #34495e;
            color: white;
            padding: 8px;
            border: 1px solid #2c3e50;
            font-weight: bold;
        }
        QTableWidget::item {
            padding: 8px;
        }
    """)

    # Error text styling
    window.error_text.setStyleSheet("""
        QTextEdit {
            background-color: #2c3e50;
            color: #e74c3c;
            border: none;
            font-family: 'Consolas', monospace;
            font-size: 14px;
            padding: 10px;
        }
    """)
```

TREE_VISUALIZER.PY

```
import networkx as nx
import matplotlib.pyplot as plt
from networkx.drawing.nx_pydot import graphviz_layout
from parser.parser import Node  # Fixed import path


class ParseTreeVisualizer:
    """Visualizes the parse tree using networkx and matplotlib"""

    def __init__(self):
        self.graph = nx.DiGraph()
        self.node_labels = {}
        self.node_colors = {}
        self.node_counter = 0

    def _get_node_id(self):
        """Generate a unique node identifier"""
        self.node_counter += 1
```

```
        return f"node_{self.node_counter}"

    def build_tree(self, node: Node, parent_id=None):
        """Build the networkx graph from the parse tree"""
        if not node:
            return

        current_id = self._get_node_id()

        # Set node properties
        self.node_labels[current_id] = f"{node.type}\n{node.value}"
        self.node_colors[current_id] = self._get_node_color(node.type)

        self.graph.add_node(current_id)
        if parent_id:
            self.graph.add_edge(parent_id, current_id)

        for child in node.children:
            self.build_tree(child, current_id)

    def _get_node_color(self, node_type: str) -> str:
        """Get the color for a node based on its type"""
        colors = {
            'operator': '#3498db',  # Blue
            'assign': '#e74c3c',    # Red
            'logical': '#2ecc71',   # Green
            'compare': '#f1c40f',   # Yellow
            'literal': '#9b59b6',   # Purple
            'group': '#95a5a6',     # Gray
        }
        return colors.get(node_type.lower(), '#34495e')  # Default dark blue

    def visualize(self, root_node: Node):
        """Create and return a matplotlib figure of the parse tree"""
        if not root_node:
            return None

        # Reset the graph state
        self.graph.clear()
        self.node_labels.clear()
        self.node_colors.clear()
        self.node_counter = 0

        # Build and draw the tree
        self.build_tree(root_node)

        plt.figure(figsize=(12, 8))
        pos = graphviz_layout(self.graph, prog="dot")

        nx.draw(self.graph, pos,
                labels=self.node_labels,
                node_color=[self.node_colors[node] for node in
self.graph.nodes()],
                node_size=2500,
                font_size=10,
                font_weight='bold',
                font_color='white',
```

```
                edge_color='#95a5a6',
                width=2,
                arrows=True,
                arrowsize=20)

        plt.title("Parse Tree Visualization", pad=20, fontsize=16,
fontweight='bold')
        plt.tight_layout()
        return plt.gcf()
```

(LEXER FOLDER)

LEXER.PY

```
import re
from dataclasses import dataclass
from typing import List

@dataclass
class Token:
    """Represents a token in the source code"""
    type: str
    value: str
    position: int
    line: int

class Lexer:
    """Performs lexical analysis of source code"""

    TOKEN_SPECS = [
        ('NUMBER',      r'\b\d+(\.\d+)?\b'),
        ('STRING',      r'"[^"]*"|\'[^\']*\''),
        ('KEYWORD',
r'\b(if|else|while|for|return|def|class|import|from)\b'),
        ('BOOL_OP',     r'\b(and|or|not)\b'),
        ('COMPARE_OP', r'<=|>=|==|!=|<|>'),
        ('ASSIGN_OP',  r'=|\+=|-=|\*=|/='),
        ('OPERATOR',    r'[+\-*/]'),
        ('IDENTIFIER', r'\b[a-zA-Z_][a-zA-Z_0-9]*\b'),
        ('LPAREN',      r'\('),
        ('RPAREN',      r'\)'),
        ('LBRACE',      r'\{'),
        ('RBRACE',      r'\}'),
        ('LBRACKET',    r'\['),
        ('RBRACKET',    r'\]'),
        ('SEMICOLON',   r';'),
        ('COMMA',       r','),
        ('DOT',         r'\.'),
        ('NEWLINE',     r'\n'),
        ('WHITESPACE', r'[ \t]+'),
        ('MISMATCH',    r'.'),
    ]

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

```
        self.errors = []

    def tokenize(self, code: str) -> List[Token]:
        """Convert source code into a list of tokens"""
        tokens = []
        line_num = 1

        for match in re.finditer(self.token_regex, code):
            kind = match.lastgroup
            value = match.group()
            position = match.start()

            if kind == 'WHITESPACE':
                continue
            elif kind == 'NEWLINE':
                line_num += 1
                continue
            elif kind == 'MISMATCH':
                self.errors.append(
                    f"Invalid character '{value}' at line {line_num},
position {position}")
                continue

            tokens.append(Token(kind, value, position, line_num))

        return tokens
```

(PARSER FOLDER)

PARSER.PY

```
from dataclasses import dataclass
from typing import List, Optional
from lexer.lexer import Token

@dataclass
class Node:
    """Represents a node in the parse tree"""
    type: str
    value: str
    children: List['Node']
    token: Token

class Parser:
    """Performs syntax analysis and builds parse tree"""

    def   init  (self):
        self.tokens: List[Token] = []
        self.current = 0
        self.errors = []

    def parse(self, tokens: List[Token]) -> Optional[Node]:
        """Parse the tokens and return the root node of the parse tree"""
        self.tokens = tokens
        self.current = 0
        self.errors = []
```

```
        if not tokens:
            return None

        try:
            return self.parse_expression()
        except Exception as e:
            self.errors.append(str(e))
            return None

    def parse_expression(self) -> Node:
        """Parse an expression"""
        return self.parse_assignment()

    def parse_assignment(self) -> Node:
        """Parse an assignment expression"""
        left = self.parse_logical()

        while self.match('ASSIGN_OP'):
            operator = self.previous()
            right = self.parse_logical()
            left = Node("assign", operator.value, [left, right], operator)

        return left

    def parse_logical(self) -> Node:
        """Parse a logical expression"""
        left = self.parse_comparison()

        while self.match('BOOL_OP'):
            operator = self.previous()
            right = self.parse_comparison()
            left = Node("logical", operator.value, [left, right], operator)

        return left

    def parse_comparison(self) -> Node:
        """Parse a comparison expression"""
        left = self.parse_term()

        while self.match('COMPARE_OP'):
            operator = self.previous()
            right = self.parse_term()
            left = Node('compare', operator.value, [left, right], operator)

        return left

    def parse_term(self) -> Node:
        """Parse a term"""
        left = self.parse_factor()

        while self.match('OPERATOR'):
            operator = self.previous()
            right = self.parse_factor()
            left = Node('operator', operator.value, [left, right], operator)

        return left
```

```
    def parse_factor(self) -> Node:
        """Parse a factor"""
        if self.match('NUMBER', 'STRING', 'IDENTIFIER'):
            return Node('literal', self.previous().value, [],
self.previous())

        if self.match('LPAREN'):
            expr = self.parse_expression()
            self.consume('RPAREN', "Expected ')' after expression")
            return Node('group', '()', [expr], self.previous())

        raise Exception(f"Unexpected token: {self.peek().value}")

    def match(self, *types) -> bool:
        """Check if current token matches any of the given types"""
        for type in types:
            if self.check(type):
                self.advance()
                return True
        return False

    def check(self, type: str) -> bool:
        """Check if current token is of the given type"""
        if self.is_at_end():
            return False
        return self.peek().type == type

    def advance(self) -> Token:
        """Advance to next token"""
        if not self.is_at_end():
            self.current += 1
        return self.previous()

    def is_at_end(self) -> bool:
        """Check if we've reached the end of tokens"""
        return self.current >= len(self.tokens)

    def peek(self) -> Token:
        """Look at current token"""
        return self.tokens[self.current]

    def previous(self) -> Token:
        """Get previous token"""
        return self.tokens[self.current - 1]

    def consume(self, type: str, message: str) -> Token:
        """Consume current token if it matches the type"""
        if self.check(type):
            return self.advance()
        raise Exception(message)
```

(MAIN CLASSES)

MAIN.PY

```
import sys
import os
import matplotlib

# Set backend before importing PyQt
matplotlib.use('Qt5Agg')

from PyQt5.QtWidgets import QApplication
from gui.main_window import SyntaxAnalyzerGUI

def main():
    app = QApplication(sys.argv)
    app.setStyle("Fusion")

    window = SyntaxAnalyzerGUI()
    window.show()
    sys.exit(app.exec_())

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

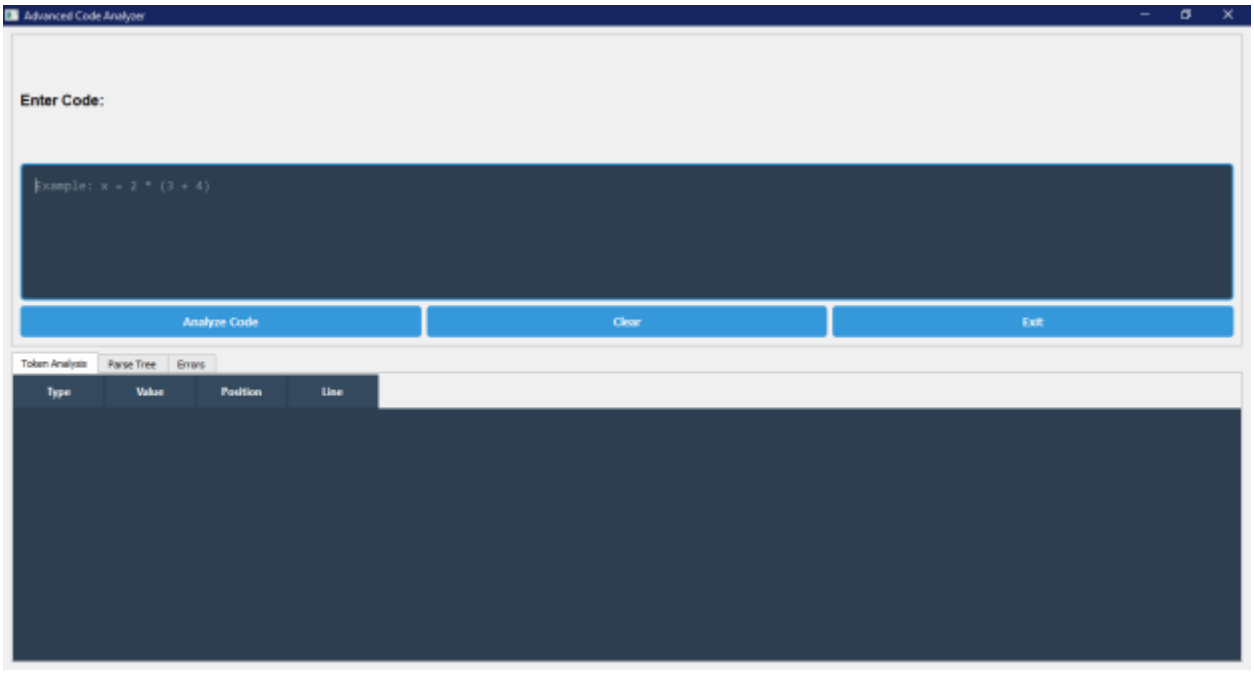
MAIN_WINDOW.PY

```
from PyQt5.QtWidgets import (QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
                             QTextEdit, QPushButton, QLabel, QFrame,
QMessageBox,
                             QTabWidget, QTableWidget, QTableWidgetItem)
from PyQt5.QtGui import QFont
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as
FigureCanvas
import matplotlib.pyplot as plt

class SyntaxAnalyzerGUI(QMainWindow):
    def clear_results(self):
        """Clear all result displays."""
        self.token_table.setRowCount(0)
        self.error_text.clear()

        # Properly clean up matplotlib figures
        for i in reversed(range(self.tree_layout.count())):
            widget = self.tree_layout.itemAt(i).widget()
            if isinstance(widget, FigureCanvas):
                plt.close(widget.figure)
            if widget:
                widget.setParent(None)
                widget.deleteLater()

    def analyze_code(self):
        """Analyze the input code and display results."""
        code = self.input_text.toPlainText().strip()
        if not code:
```

```
            QMessageBox.warning(self, "Input Error", "Please enter code to
analyze.")
            return

        self.clear_results()

        try:
            # Perform lexical analysis
            tokens = self.lexer.tokenize(code)
            self.update_token_table(tokens)

            if self.lexer.errors:
                self.show_errors(self.lexer.errors)
                return

            # Perform syntax analysis
            parse_tree = self.parser.parse(tokens)

            if self.parser.errors:
                self.show_errors(self.parser.errors)
            elif parse_tree:
                figure = self.visualizer.visualize(parse_tree)
                if figure:
                    canvas = FigureCanvas(figure)
                    self.tree_layout.addWidget(canvas)
                    self.results_tab.setCurrentWidget(self.tree_widget)
        except Exception as e:
            self.show_errors([f"Error during analysis: {str(e)}"])

    def closeEvent(self, event):
        """Clean up resources when closing the window."""
        self.clear_results()
        super().closeEvent(event)
```

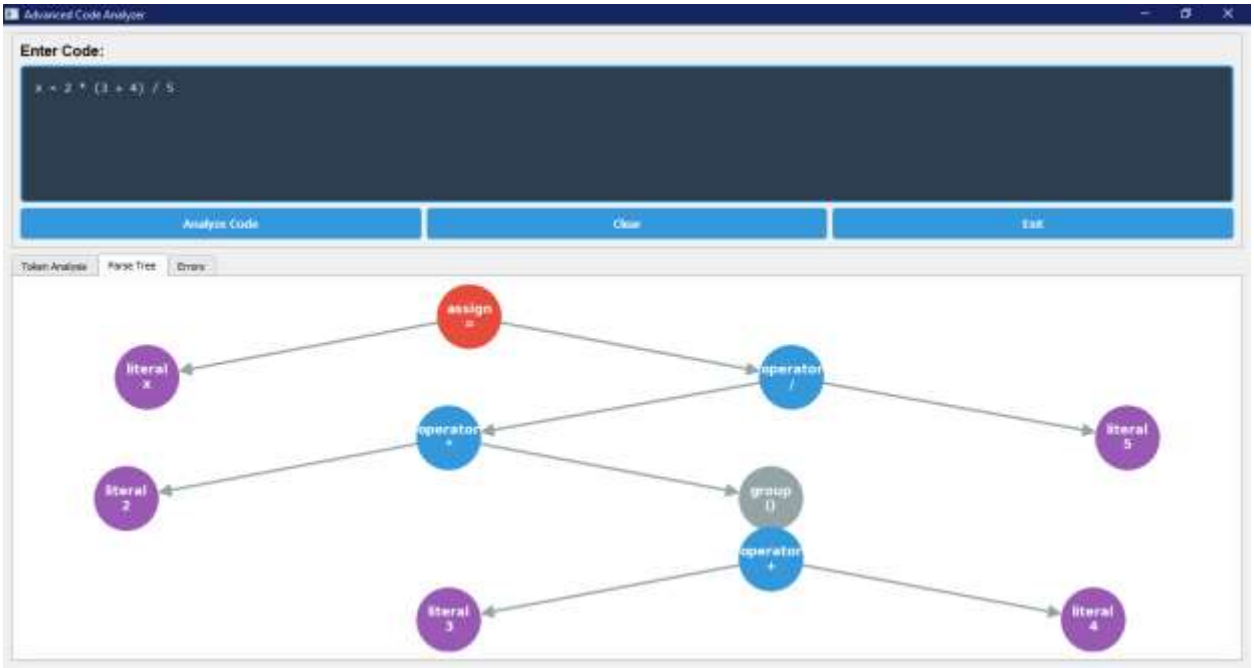
Output

MAIN GUI



TOKEN ANALYSIS

PARSE TREE


Advanced Code Analyzer
Enter Code:
x = 2 * (3 + 4) / 5
Analyze Code
Clear
Exit
Token Analysis
Parse Tree
Errors
assign
=
literal
x
operator
/
operator
*
literal
5
literal
2
group
()
operator
+
literal
3
literal
4

Future Development

1. **Enhanced Language Support**
    - Extension of grammar rules for additional language constructs
    - Support for multiple programming language parsing
    - Custom grammar definition capabilities
2. **Advanced Visualization**
    - Interactive parse tree manipulation
    - Animation of parsing process
    - Alternative tree visualization layouts
    - Export capabilities for generated diagrams
3. **Performance Optimization**
    - Improved parsing algorithms for larger code bases
    - Optimized memory usage for parse tree generation
    - Enhanced error recovery mechanisms

Conclusion

The Syntax Analyzer project successfully demonstrates the practical implementation of compiler construction concepts through an interactive and visual approach. The combination of PLY-based parsing, PyQt5 interface, and parse tree visualization provides a comprehensive tool for understanding syntax analysis. The modular design ensures maintainability and extensibility for future enhancements, while the user-friendly interface makes it accessible to both students and developers.

The project achieves its primary objectives of:

- Implementing a functional syntax analyzer
- Providing visual representation of parsing processes
- Creating an interactive learning tool for compiler construction
- Demonstrating practical application of theoretical concepts

Through this implementation, users can better understand the complexities of syntax analysis and compiler construction while having access to practical tools for code analysis and learning.