



**Graphic Era
Hill University**
BHIMTAL CAMPUS

Term work of

**Project Based Learning (PBL)
of
Compiler Design**

Submitted in fulfillment of the requirement for the VI semester

Bachelor of Technology

By

Sumit Singh Rana

Vaibhav Chauhan

Himanshu Chilkoti

Adarsh Singh Bisht

Under the Guidance of

Mr Devesh Pandey

Assistant Professor

Dept. of CSE

GRAPHIC ERA HILL UNIVERSITY, BHIMTAL CAMPUS

SATTAL ROAD, P.O. BHOWALI

DISTRICT- NAINITAL-263132

2024 – 2025



**Graphic Era
Hill University**
BHIMTAL CAMPUS

CERTIFICATE

The term work of Project Based Learning, being submitted by Sumit Singh Rana (2261557) s/o Mr. Anand Singh Rana, Adarsh Singh Bisht (2261072) s/o Mr. Navin Singh Bisht, Vaibhav Chauhan (2261585) s/o Mr. CMS Chauhan and Himanshu Chilkoti (2261265) s/o Mr. Ramesh Chandra to Graphic Era Hill University Bhimtal Campus for the award of Bonafide work carried out by us. They had worked under my guidance and supervision and fulfilled the requirements for the submission of this work report.

(Mr. Devesh Pandey)

Assistant Professor

(Dr. Ankur Singh Bisht)

HOD, CSE Dept



**Graphic Era
Hill University**
BHIMTAL CAMPUS

STUDENT'S DECLARATION

We, Sumit Singh Rana, Adarsh Singh Bisht, Vaibhav Chauhan, Himanshu Chilkoti hereby declare the work, which is being presented in the report, entitled Syntax checker in c and python in fulfillment of the requirement for the award of the degree **Bachelor of Technology (Computer Science)** in the session **2024 - 2025** for semester VI, is an authentic record of our own work carried out under the supervision of **Mr. Devesh Pandey**.

(Graphic Era Hill University, Bhimtal)

The matter embodied in this project has not been submitted by us for the award of any other degree.

Date:

.....

(Full signature of students)

Table of Content

Chapter	Pages
1. Introduction.....	5
1.1 Project Overview.....	5
1.2 Objective.....	5
1.3 Technologies Used.....	6
1.4 GitHub Contribution.....	7
2. Compilation Workflow.....	8
2.1 Phases, Compilation and Execution.....	8
3. System Design.....	12
3.1 Flow Chart.....	12
3.2 Data Flow Diagram.....	13
4. Features of Compiler.....	14
4.1 Feature 1.....	14
4.2 Feature 2.....	15
5. Conclusion.....	16

Introduction

Project Overview

The AST-based Syntax Checker and Visualizer for C and Python is a compiler-inspired project aimed at analyzing and visualizing the structure of source code through its Abstract Syntax Tree (AST). The project offers support for both C and Python programming languages, enabling users to detect syntax errors and visualize the corresponding AST for improved code understanding.

This tool simulates the parsing phase of a compiler, where input code is first checked for syntax correctness and then translated into a hierarchical tree structure. For Python, it uses the built-in `ast` module, and for C, it leverages `pycparser`. The tool also integrates a professional, dark-themed graphical user interface (GUI) built with **PyQt6**, which makes the application user-friendly and interactive.

The project aims to help students and developers understand the internal structure of code beyond simple syntax validation by providing a real-time visual representation of ASTs. Visualization is done using **Graphviz**, which renders the AST into scalable graphical formats. With modular architecture and multi-language support, this project is not only an educational resource but also a foundation for future extensions such as semantic analysis, error explanation tooltips, and runtime validation for C programs.

Objectives

- 🔗 **To build a syntax checker for Python and C that identifies and reports syntax errors in real time.**
- ❑ **To visualize the Abstract Syntax Tree (AST) for both Python and C source code, aiding users in understanding the structural representation of their programs.**
- ❑ **To develop a professional, user-friendly GUI using PyQt6, allowing users to input code, view errors, and interact with AST visualizations in a seamless environment.**
- ❑ **To leverage parsing tools and libraries, including Python's `ast` module and `pycparser` for C, to perform syntax validation and AST generation.**
- ❑ **To integrate Graphviz for rendering clean, scrollable AST diagrams within the application.**

Technologies Used:

1. Programming Language:

2. Python 3 – Core development language for the syntax checker and visualizer.

3. C (via pycparser) – Support for analyzing C source code and generating ASTs.

4. GUI Framework:

5. PyQt6 – Used to develop a professional, dark-themed graphical user interface that allows users to input code, receive error feedback, and visualize ASTs interactively.

6. AST Generation Tools:

7. Python ast module – Built-in module to parse Python code and generate its Abstract Syntax Tree.

8. pycparser – A C parser in Python used to build and analyze ASTs from C source code.

9. Graphviz – Used to convert AST data into structured, graphical visualizations that are displayed within the GUI.

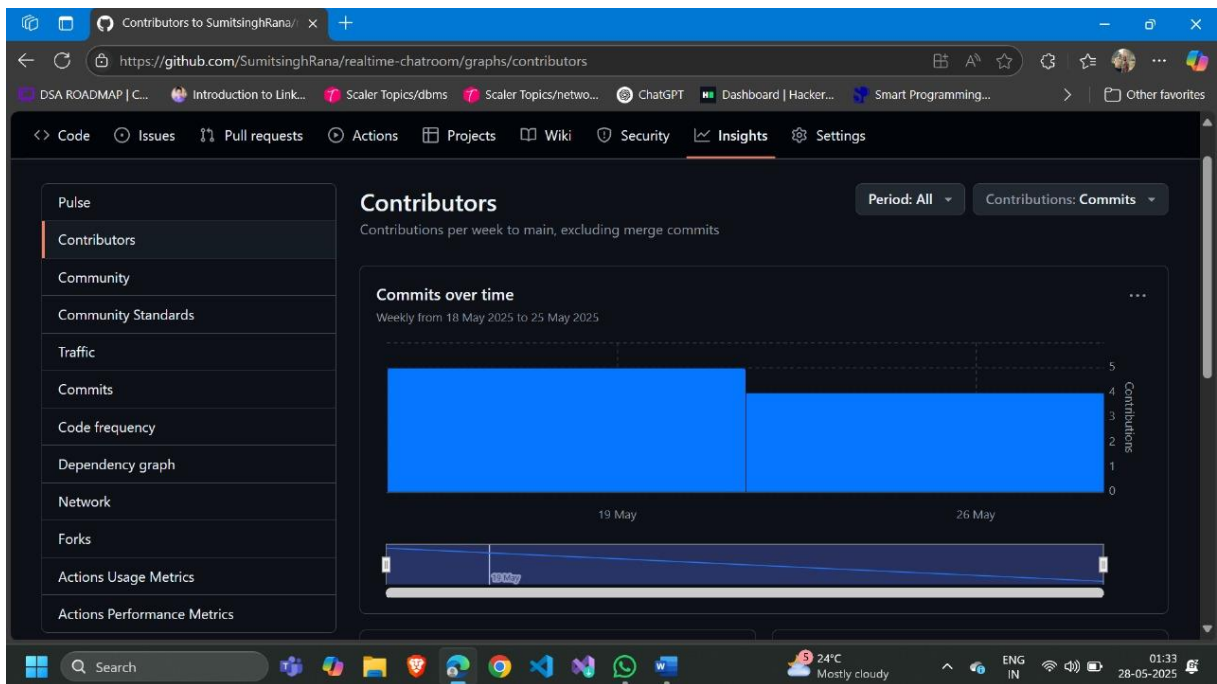
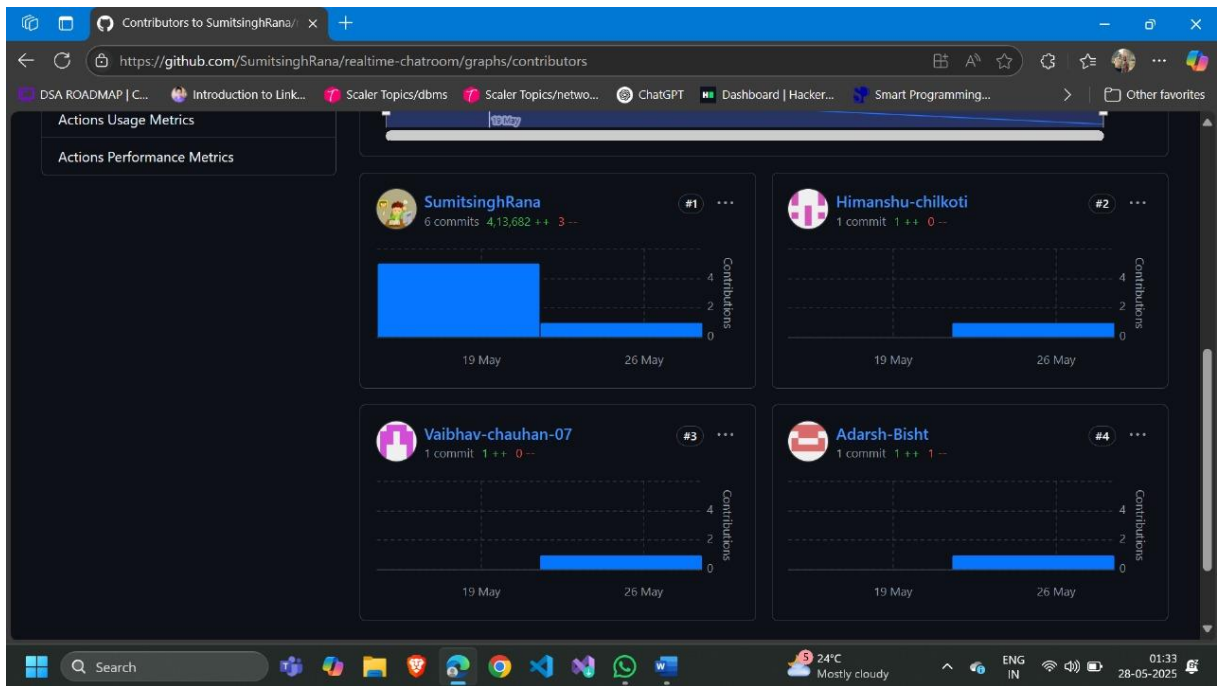
10. os and sys modules – Used for file operations, execution control, and system-level integration.

11. Git and GitHub – For version tracking, collaborative development, and maintaining project history.

12. Code Structure & Organization:

13. Modular project layout with separate directories for Python logic, C logic, GUI components, and output generation.

Github



Compilation Workflow

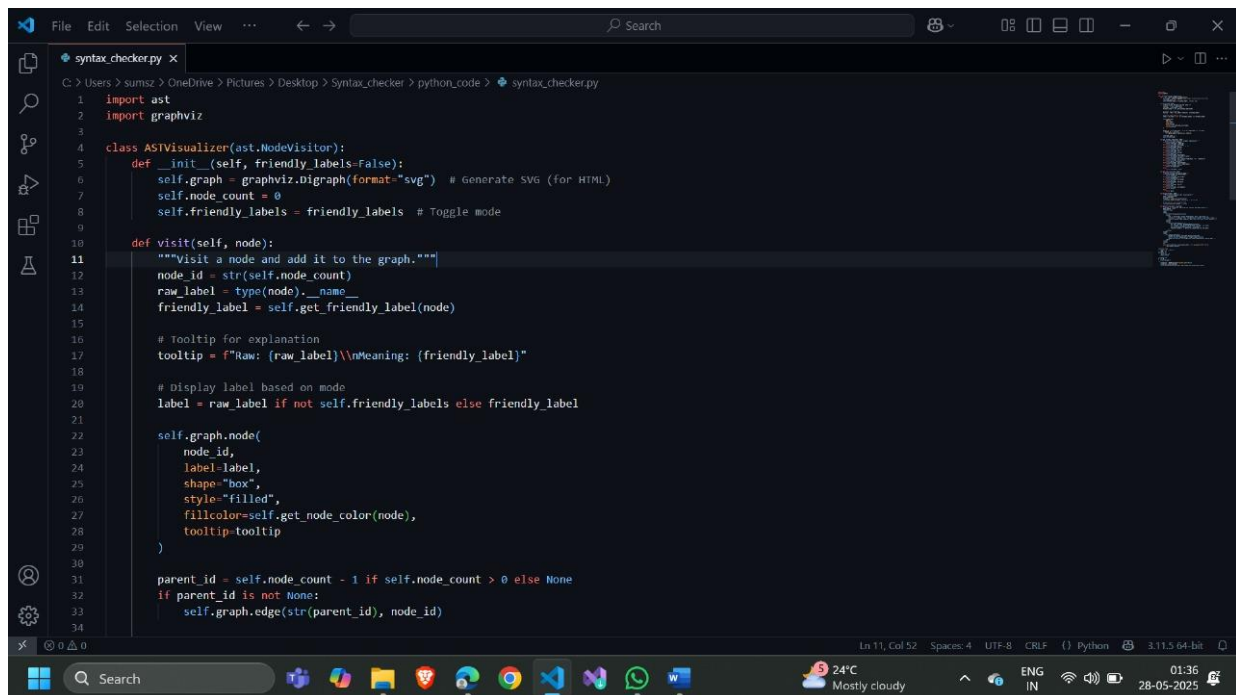
Phases, Compilation and Execution:

1. Python AST Generation – syntax_checker.py

This module defines the logic for traversing the Python Abstract Syntax Tree (AST) and visualizing it using Graphviz. It uses Python's ast module to visit nodes, assign labels, and generate tooltips that explain the meaning of each AST node.

🔑 Key Functionality:

- AST node traversal using visit()
- Dynamic label generation based on raw and friendly names
- Graphviz node and edge creation
- Tooltip support for enhanced clarity



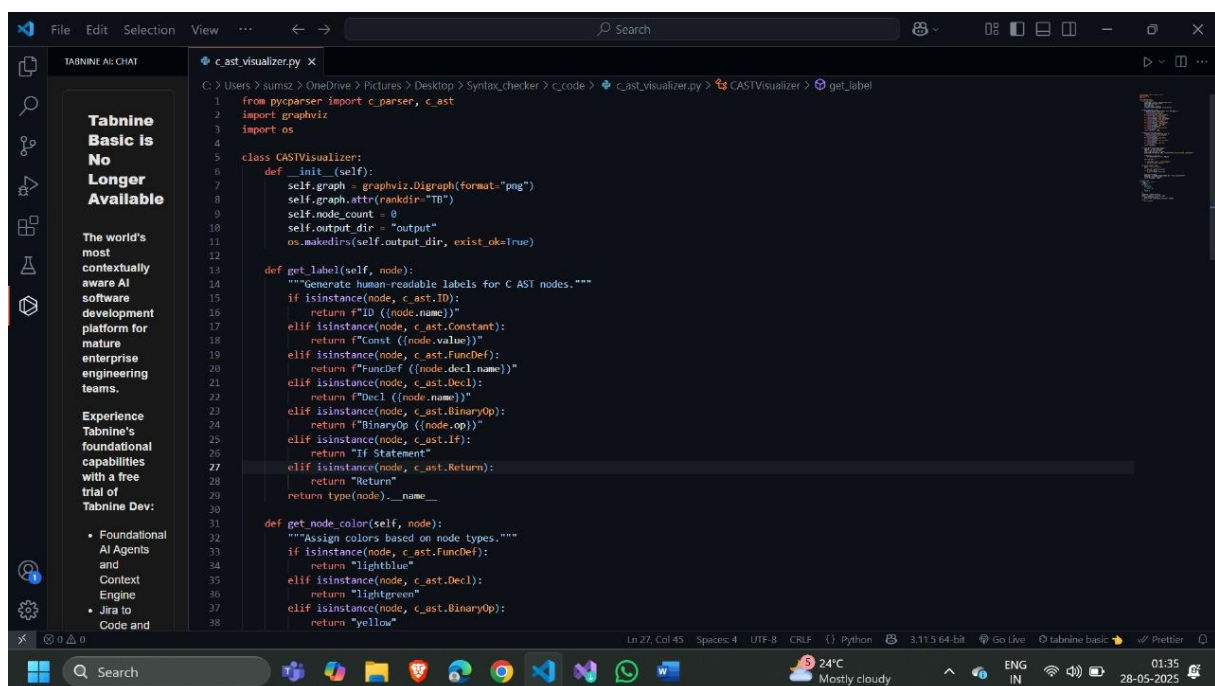
```
1 import ast
2 import graphviz
3
4 class ASIVisualizer(ast.NodeVisitor):
5     def __init__(self, friendly_labels=False):
6         self.graph = graphviz.Digraph(format="svg") # Generate SVG (for HTML)
7         self.node_count = 0
8         self.friendly_labels = friendly_labels # Toggle mode
9
10    def visit(self, node):
11        """Visit a node and add it to the graph."""
12        node_id = str(self.node_count)
13        raw_label = type(node).__name__
14        friendly_label = self.get_friendly_label(node)
15
16        # Tooltip for explanation
17        tooltip = f"Raw: {raw_label}\\nMeaning: {friendly_label}"
18
19        # Display label based on mode
20        label = raw_label if not self.friendly_labels else friendly_label
21
22        self.graph.node(
23            node_id,
24            label=label,
25            shape="box",
26            style="filled",
27            fillcolor=self.get_node_color(node),
28            tooltip=tooltip
29        )
30
31        parent_id = self.node_count - 1 if self.node_count > 0 else None
32        if parent_id is not None:
33            self.graph.edge(str(parent_id), node_id)
```


2. C AST Generation – c_ast_visualizer.py

This script processes C source code using pycparser, generates AST nodes, and creates a Graphviz-based visualization. It includes logic to assign human-readable labels to different C constructs such as declarations, functions, and return statements.

🔑 Key Functionality:

- Parsing C code into AST using pycparser
- Generating descriptive labels for C constructs
- Custom color coding based on node type
- Output saved in a dedicated directory



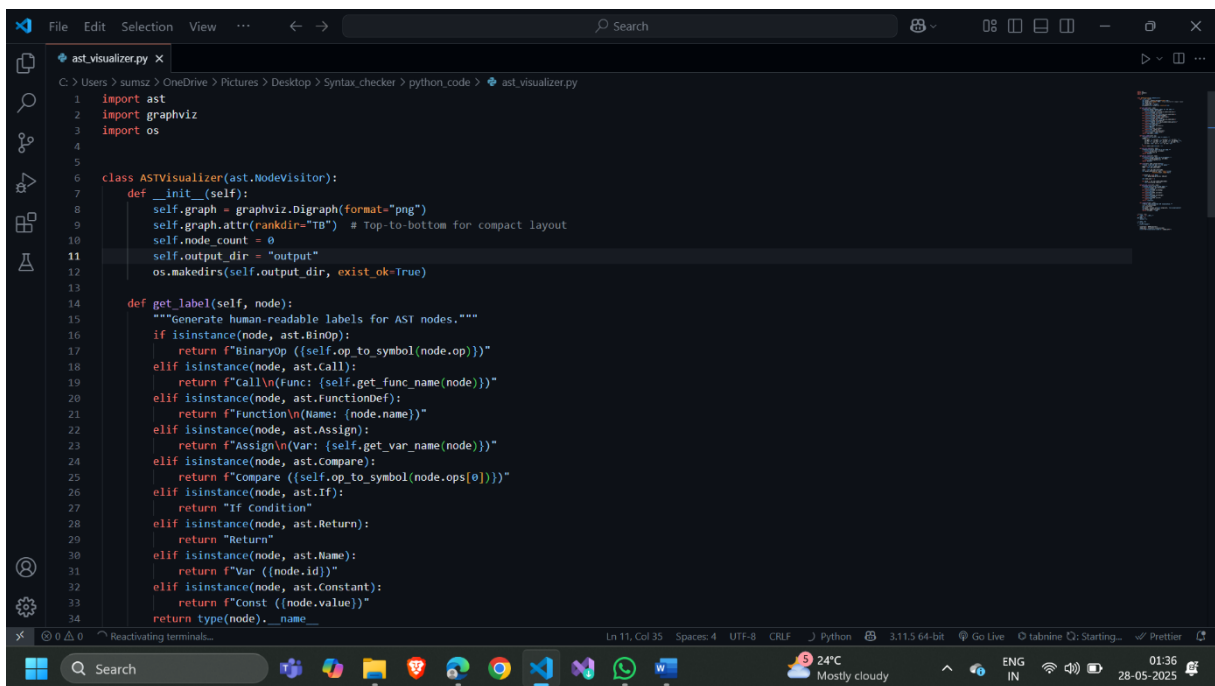
```
C:\Users\sumsz>OneDrive>Pictures>Desktop>Syntax_checker>c_code>c_ast_visualizer.py
1 from pycparser import c_parser, c_ast
2 import graphviz
3 import os
4
5 class CASTVisualizer:
6     def __init__(self):
7         self.graph = graphviz.Digraph(format='png')
8         self.graph.attr(rankdir="TB")
9         self.node_count = 0
10        self.output_dir = "output"
11        os.makedirs(self.output_dir, exist_ok=True)
12
13    def get_label(self, node):
14        """Generate human-readable labels for C AST nodes."""
15        if isinstance(node, c_ast.ID):
16            return f"ID ({node.name})"
17        elif isinstance(node, c_ast.Constant):
18            return f"Const ({node.value})"
19        elif isinstance(node, c_ast.FuncDef):
20            return f"FuncDef ({node.decl.name})"
21        elif isinstance(node, c_ast.Decl):
22            return f"Decl ({node.name})"
23        elif isinstance(node, c_ast.BinaryOp):
24            return f"BinaryOp ({node.op})"
25        elif isinstance(node, c_ast.If):
26            return "If Statement"
27        elif isinstance(node, c_ast.Return):
28            return "Return"
29        return type(node).__name__
30
31    def get_node_color(self, node):
32        """Assign colors based on node types."""
33        if isinstance(node, c_ast.FuncDef):
34            return "lightblue"
35        elif isinstance(node, c_ast.Decl):
36            return "lightgreen"
37        elif isinstance(node, c_ast.BinaryOp):
38            return "yellow"
```

◆ 3. Python AST Node Labeling – ast_visualizer.py

This Python module defines how AST nodes for Python code are converted into friendly, readable labels. Each node (like FunctionDef, Call, Assign, If, etc.) is processed and labeled to enhance educational value during visualization.

🔑 Key Functionality:

- Symbol-to-label mapping for AST nodes
- Custom display of binary operations, function calls, variable assignments, etc.
- Visual distinction of constructs in rendered AST images



```
1 import ast
2 import graphviz
3 import os
4
5
6 class ASTVisualizer(ast.NodeVisitor):
7     def __init__(self):
8         self.graph = graphviz.Digraph(format="png")
9         self.graph.attr(rankdir="TB") # Top-to-bottom for compact layout
10        self.node_count = 0
11        self.output_dir = "output"
12        os.makedirs(self.output_dir, exist_ok=True)
13
14    def get_label(self, node):
15        """Generate human-readable labels for AST nodes."""
16        if isinstance(node, ast.BinOp):
17            return f"BinaryOp ({self.op_to_symbol(node.op)})"
18        elif isinstance(node, ast.Call):
19            return f"Call\n(Func: {self.get_func_name(node)})"
20        elif isinstance(node, ast.FunctionDef):
21            return f"Function\n(Name: {node.name})"
22        elif isinstance(node, ast.Assign):
23            return f"Assign\n(Var: {self.get_var_name(node)})"
24        elif isinstance(node, ast.Compare):
25            return f"Compare ({self.op_to_symbol(node.ops[0])})"
26        elif isinstance(node, ast.If):
27            return "If Condition"
28        elif isinstance(node, ast.Return):
29            return "Return"
30        elif isinstance(node, ast.Name):
31            return f"Var ({node.id})"
32        elif isinstance(node, ast.Constant):
33            return f"Const ({node.value})"
34        return type(node).__name__
```

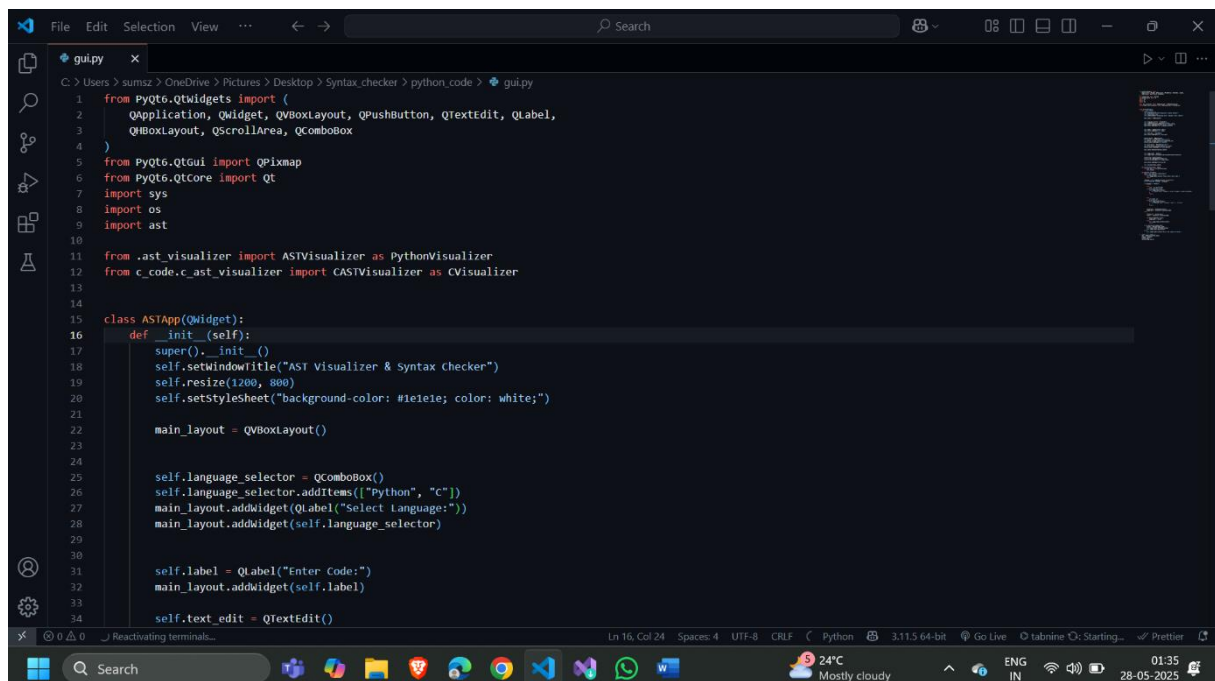
◆ 4. Graphical User Interface – gui.py

The GUI is built with PyQt6 and serves as the primary interface where users:

- Enter code
- Select the language (Python or C)
- Visualize ASTs
- View errors

🔑 Key Features:

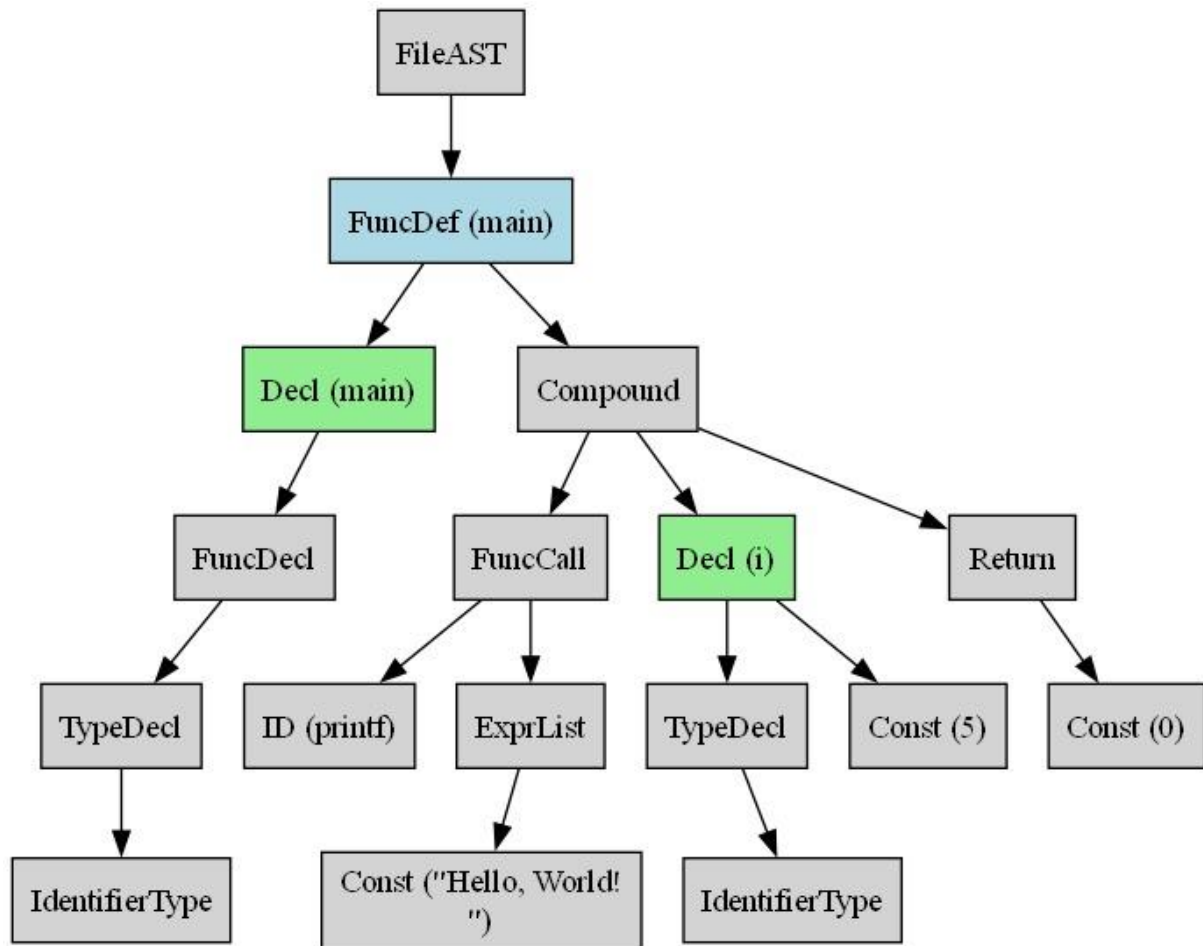
- Language dropdown (Python/C)
- Code editor area
- Integrated AST rendering with scrollable view
- Real-time syntax checking and error display

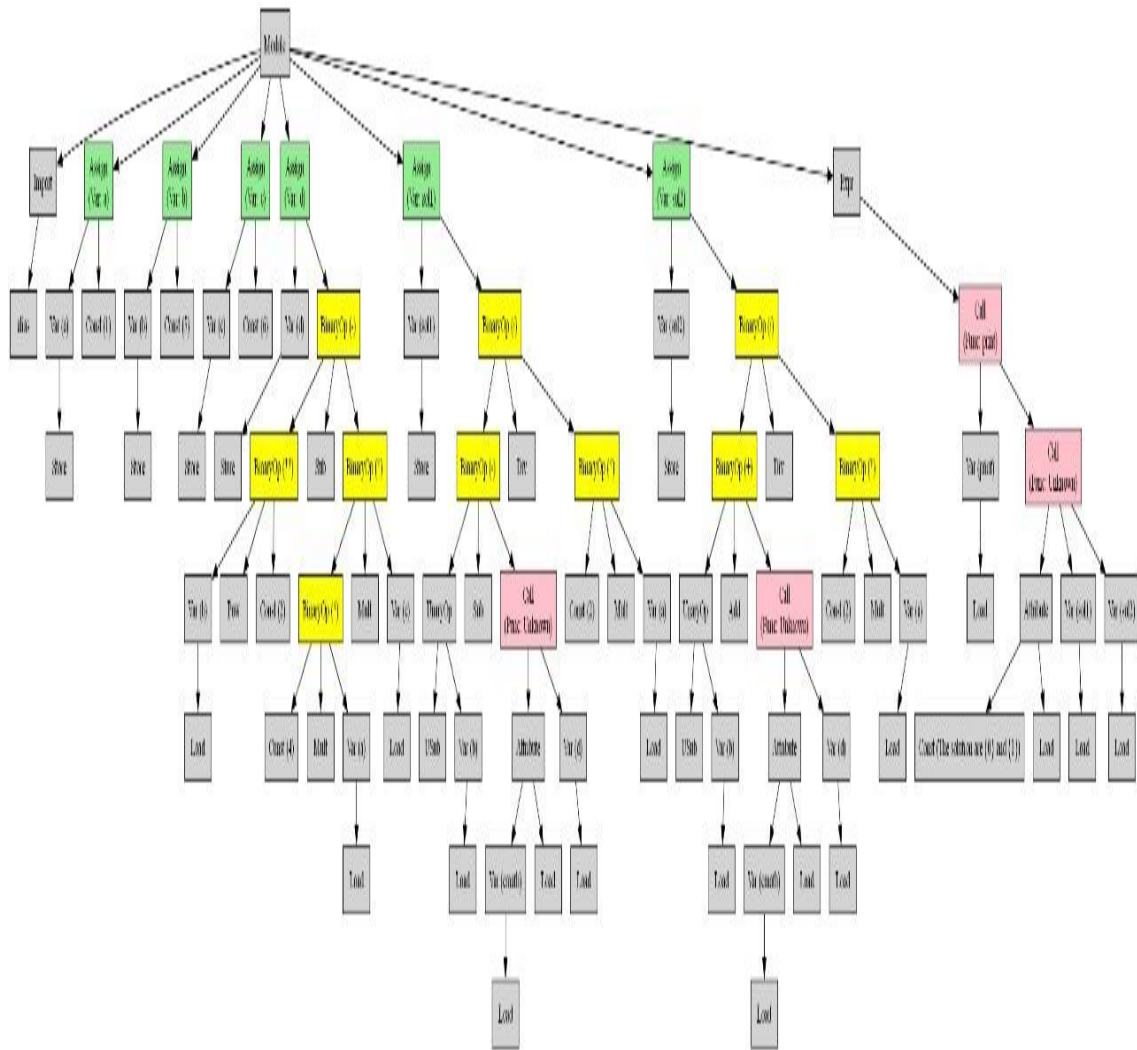


```
1 from PyQt6.QtWidgets import (
2     QApplication, QWidget, QVBoxLayout, QPushButton, QTextEdit, QLabel,
3     QHBoxLayout, QScrollArea, QComboBox
4 )
5 from PyQt6.QtGui import QPixmap
6 from PyQt6.QtCore import Qt
7 import sys
8 import os
9 import ast
10
11 from .ast_visualizer import ASTVisualizer as PythonVisualizer
12 from c_code.c_ast_visualizer import CASTVisualizer as CVisualizer
13
14
15 class ASTApp(QWidget):
16     def __init__(self):
17         super().__init__()
18         self.setWindowTitle("AST Visualizer & Syntax Checker")
19         self.resize(1200, 800)
20         self.setStyleSheet("background-color: #1e1e1e; color: white;")
21
22         main_layout = QVBoxLayout()
23
24
25         self.language_selector = QComboBox()
26         self.language_selector.addItems(["Python", "C"])
27         main_layout.addWidget(QLabel("Select Language:"))
28         main_layout.addWidget(self.language_selector)
29
30
31         self.label = QLabel("Enter Code:")
32         main_layout.addWidget(self.label)
33
34         self.text_edit = QTextEdit()
```

System Design

Flow Chart:





Features of Syntax Checker in C and Python

Feature 1: Converts C code to Pseudocode

This component takes source code written in the C programming language and transforms it into clear, structured pseudocode. It analyzes the syntax and semantics of the C code by parsing its statements, control structures, and expressions, then maps these elements into a simplified, human-readable format that preserves the original program's logic. The goal is to help users, especially beginners, understand the flow and functionality of C programs without getting bogged down by complex syntax, making debugging and learning more intuitive.

```
1  from pycparser import c_parser, c_ast
2  from pycparser.c_generator import CGenerator
3  import re
4
5  class CCodeVisitor(c_ast.NodeVisitor):
6      def __init__(self):
7          self.pseudo = []
8          self.indent_level = 0
9
10     def indent(self):
11         return "    " * self.indent_level
12
13     def visit_FuncDef(self, node):
14         name = node.decl.name
15         args = []
16         if isinstance(node.decl.type.args, c_ast.ParamList):
17             for param in node.decl.type.args.params:
18                 args.append(param.name)
19         self.pseudo.append(f"{self.indent()}Define function {name}({','.join(args)}) Begin")
20         self.indent_level += 1
21         self.visit(node.body)
22         self.indent_level -= 1
23         self.pseudo.append(f"{self.indent()}End")
24
25     def visit_Compound(self, node):
26         for stmt in node.block_items or []:
27             self.visit(stmt)
28
29
30     def visit_Decl(self, node):
31         if isinstance(node.type, c_ast.TypeDecl):
32             if node.init:
33                 self.pseudo.append(f"{self.indent()}Declare {node.name} ← {self._expr(node.init)}")
34             else:
```

Feature 2: Converts Python code to Pseudocode

This component translates Python code into structured pseudocode by analyzing its indentation-based syntax, control flow, and expressions. It simplifies Python constructs into clear, step-by-step logic, helping users visualize the program's execution in a more readable and educational format.

```
1 import ast
2
3 class PseudoCodeGenerator(ast.NodeVisitor):
4     def __init__(self):
5         self.pseudo = []
6         self.indent_level = 0
7
8     def indent(self):
9         return "    " * self.indent_level
10
11    def visit_FunctionDef(self, node):
12        args = ", ".join([arg.arg for arg in node.args.args])
13        self.pseudo.append(f"{self.indent()}Function {node.name}({args})")
14        self.pseudo.append(f"{self.indent()}Begin")
15        self.indent_level += 1
16        for stmt in node.body:
17            self.visit(stmt)
18        self.indent_level -= 1
19        self.pseudo.append(f"{self.indent()}End Function")
20
21    def visit_Assign(self, node):
22        targets = [ast.unparse(t) for t in node.targets]
23        value = ast.unparse(node.value)
24        self.pseudo.append(f"{self.indent()}Set {' = '.join(targets)} to {value}")
25
26    def visit_If(self, node):
27        test = ast.unparse(node.test)
28        self.pseudo.append(f"{self.indent()}If {test} Then")
29        self.pseudo.append(f"{self.indent()}Begin")
30        self.indent_level += 1
31        for stmt in node.body:
32            self.visit(stmt)
33        self.indent_level -= 1
34        self.pseudo.append(f"{self.indent()}End If")
```

Conclusion

The development of the AST-based Syntax Checker and Visualizer for C and Python has provided deep insight into the foundational concepts of compiler design, particularly lexical analysis, syntax analysis, and abstract syntax tree (AST) construction. By integrating tools such as Python's `ast` module and `pycparser` for C, the system effectively parses and validates code in both languages, providing structured feedback to users.

The implementation of an interactive, dark-themed GUI using **PyQt6** significantly enhances user experience, making it easy to write, analyze, and visualize code. The use of **Graphviz** to display ASTs adds a strong visual learning component, especially beneficial for students and beginners who are trying to understand the hierarchical nature of code execution.

This project not only meets its goal of checking and visualizing code structure but also lays the groundwork for future enhancements such as semantic analysis, runtime checking for C, and more advanced educational tools like tooltips for AST nodes. The modular architecture ensures that the system can be scaled and adapted as needed.

In summary, the project successfully bridges the gap between theoretical compiler principles and practical code analysis, offering an engaging and educational tool for both developers and learners.