Term work of

Project Based Learning (PBL) of Compiler Design

Submitted in fulfillment of the requirement for the VI semester

Bachelor of Technology

By

Rohit Routela

Manish Rawat

Nikhil Singh Rautela

Akshat Joshi

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

STUDENT'S DECLARATION

We, Rohit Routela, Manish Rawat, Nikhil Singh Rautela and Aksha Joshi, hereby declare the work, which is being presented in the report, entitled **Term work of Project Based Learning of Compiler Design** 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

Introduction

Project Overview

This project is a mini-compiler that translates Python code into equivalent C++ code. It takes basic Python constructs such as variable declarations, expressions, conditional statements, loops, functions, and input/output operations, and converts them into their corresponding C++ syntax. The system is divided into several modules including lexical analysis, parsing, intermediate representation (IR) generation, and code generation. A Tkinter-based GUI is also provided for easy code input, conversion, and visualization. The main goal is to demonstrate the fundamental principles of compiler design and automation of language translation at a basic level..

This Converter is modularly built and is structured around the following core functionalities:

Key Features

- The project implements a complete conversion pipeline that takes Python code as input, processes it through lexing, parsing, AST generation, and intermediate representation (IR), and finally generates valid, executable C++ code, supporting all basic Python constructs including variables, loops, conditionals, and functions.
- **Real-Time Web Interface**:

  The interface built with **Streamlit** allows users to:

  - Input or paste their Python code
  - Click a button to instantly analyze the code
  - View results in a clean and interactive format

- **Clean Architecture**:
    - `app.py` handles the Streamlit frontend and connects all backend analysis logic.
    - `analyzer_utils.py` performs line-by-line analysis, calculates statistics, and handles text processing.
    - `ast_utils.py` processes the code using AST and extracts deeper structural insights.

Objectives

The primary objectives are as follows:

1. Create a GUI application that converts Python code to C++ code with a modern and user-friendly interface.
2. Implement a robust code analysis pipeline that includes lexical analysis, parsing, and semantic analysis to properly understand and process Python code before conversion.
3. Generate valid, well-formatted C++ code through a sophisticated code generation system that handles different Python language features and constructs
4. Provide immediate visual feedback through a dual-panel interface where users can see their Python code and the converted C++ code side by side with syntax highlighting.
5. Ensure comprehensive error handling and reporting that helps users identify issues in their Python code, unsupported features, and conversion problems.
6. Support practical file operations including loading Python files, saving C++ output, maintaining a recent files history, and copying converted code to clipboard for a smooth user experience.

Technologies Used

- **Programming Language**

  Python (primary implementation language)

- **GUI Framework**

  Tkinter (tk) - Python's standard GUI library

  Ttk – Themed Tkinter widgets for morden look

- **GUI Framework**

  AST (Abstract Syntax Tree) module - for Python code parsing.

- **Visualization Library**:

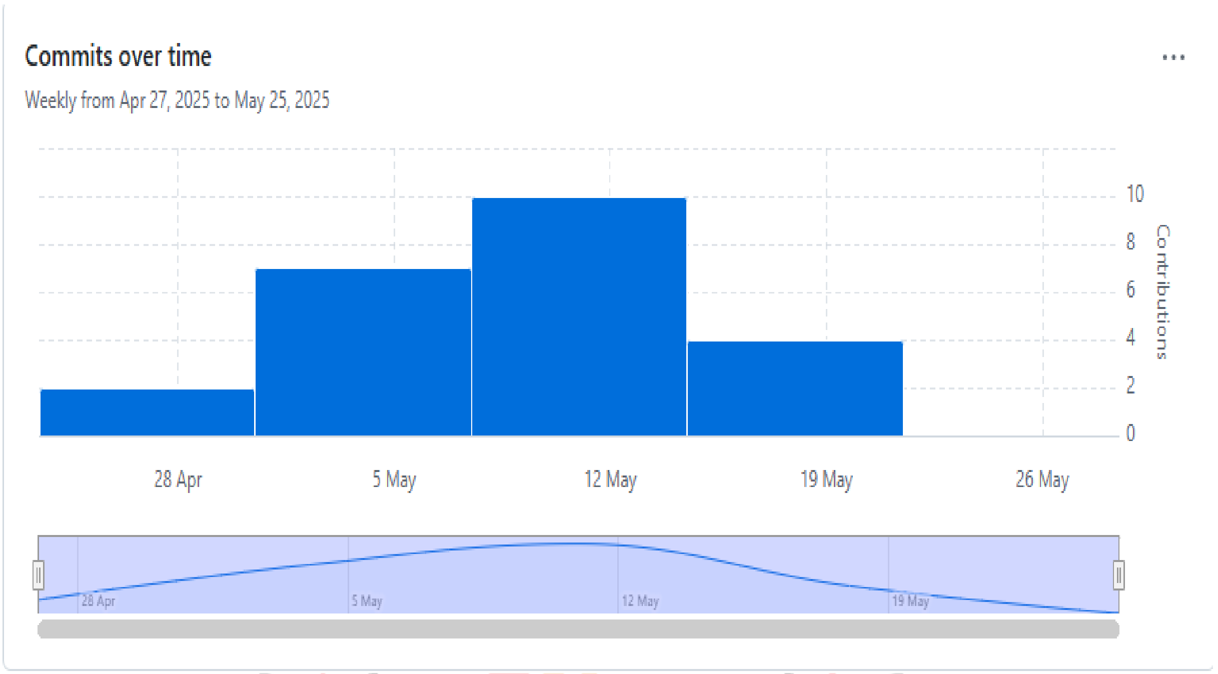
  Graphviz (optional, if used for AST)

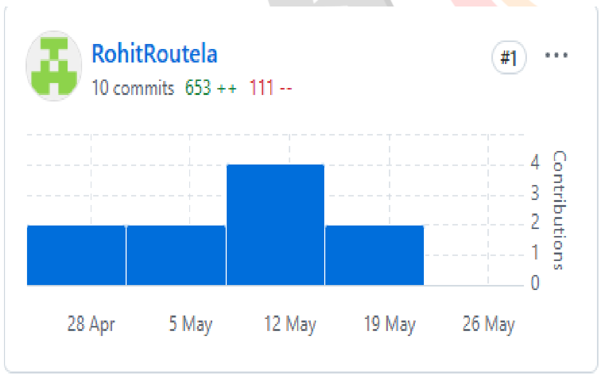
  Used for generating graphical representations of the Abstract Syntax Tree (AST)..

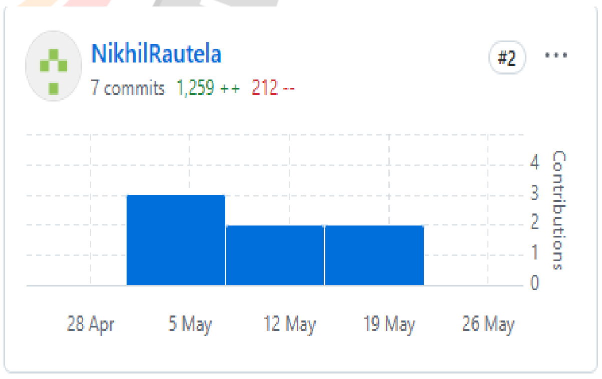
Key Components:

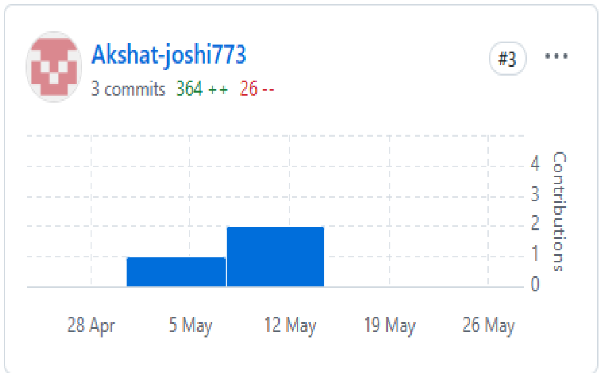
- **Lexer (Lexical Analyzer)** – Breaks down the input Python code into tokens such as keywords, identifiers, literals, and symbols.
- **Parser** – Analyzes the token sequence to build a syntax tree according to Python grammar rules.
- **Abstract Syntax Tree (AST) Generator** – Creates a structured, tree-based representation of the parsed code to reflect its hierarchical syntax.
- **Intermediate Representation (IR)** – A simplified, language-agnostic form of the AST used to ease the translation into C++.

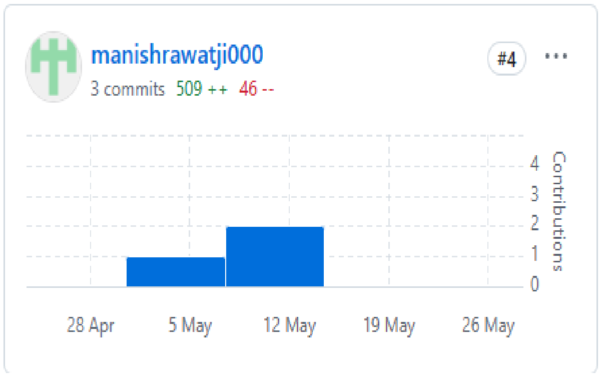
Github Contribution











Compilation Workflow

Phases, Compilation and Execution:

1. Lexical Analyzer

In our project, the **Lexical Analyzer** serves as the **first phase of code analysis**. Its role is to **break down the input source code into tokens** — these are atomic units such as keywords, identifiers, operators, literals, and delimiters that serve as the basic building blocks of code syntax.

Purpose in the Project :

- Input: Raw Python code
- Process: Breaks code into token
- Output: List of tokens
- Purpose: Identifies language elements like keywords, identifiers, operators.

```
# From test.py
tokens = tokenize(code)  # Converts source code into tokens
```

2. Parsing AST Generation

- Input: Token stream.
- Process: Builds AST according to Python grammar.
- Output: Python AST.
- Purpose: Creates hierarchical representation of code structure.

```
parser = Parser(tokens)
ast = parser.parse()  # Generates Python AST
```

2. Custom AST Conversion Phase

Transforms Python's AST into a custom AST format that's easier to work with for C++ conversion.

Purpose in our Project

- Input: Python AST.
- Process: Converts to custom AST nodes.
- Output: Custom AST.
- Purpose: Creates language-agnostic representation.

```
class CustomNodeConverter(ast.NodeVisitor):
    def visit_Module(self, node):
        return Program([self.visit(stmt) for stmt in node.body])
```

```python

    def visit_Assign(self, node):
        target = self.visit(node.targets[0])
        value = self.visit(node.value)
        return AssignmentNode(target.name, value)
```

4. IR Generation Phase

Creates an intermediate representation that bridges the gap between Python and C++ constructs.

Purpose in our Project

- Input: Custom AST.
- Process: Generates intermediate instructions.
- Output: IR instructions.
- Purpose: Creates platform-independent representation.

```python
class IRGenerator:
    def generate(self, node):
        if isinstance(node, AssignmentNode):
            expr = self.generate(node.expression)
            self.instructions.append(('assign', node.identifier, expr))
```

5. Code Generation Phase

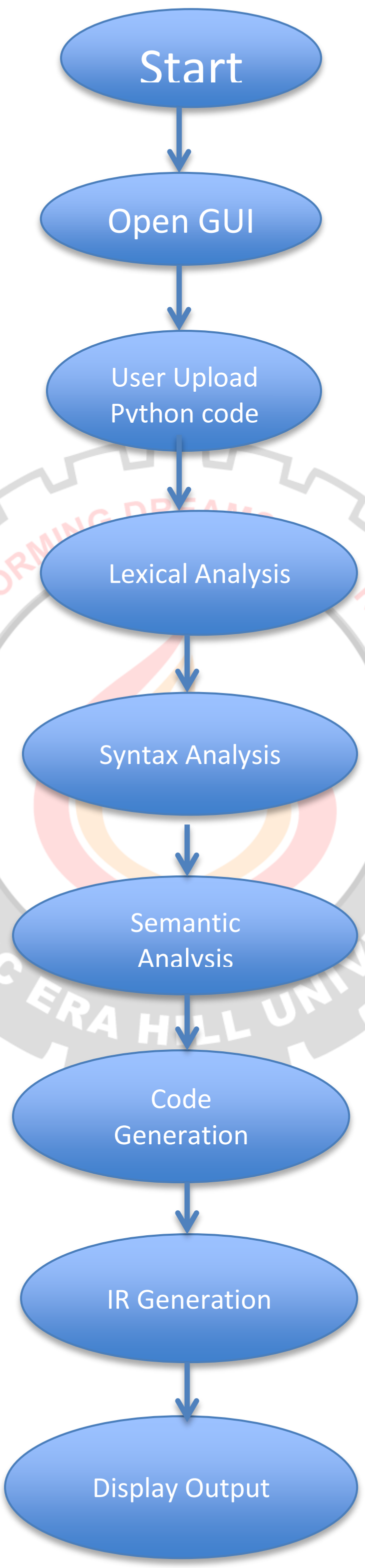
Produces the final C++ code from the intermediate representation.

- Input: IR instructions.
- Process: Converts IR to C++ code.
- Output: C++ source code.
- Purpose: Produces final target language code.

```python
class CodeGenerator:
    def generate(self, ir):
        # Process IR instructions
        if isinstance(ir, tuple):
            instruction_type = ir[0]
            if instruction_type == 'assign':
                var_name = ir[1]
                expr = self.generate_expr(ir[2])
                var_type = self.infer_var_type(i
```

System Design

Flow Chart:



Features of Compiler

1. Lexical Analysis (Tokenization)

Purpose: Lexical analysis is the first phase of a compiler, where the source code is converted into tokens.

```
# From lexer.py
class Token:
    def __init__(self, type_, value, line=None):
        self.type = type_
        self.value = value
        self.line = line

def tokenize(code):
    tokens = []
    i = 0
    length = len(code)
    line_number = 1

    while i < length:
        c = code[i]

        # Handle identifiers and keywords
        if c.isalpha() or c == '_':
            start = i
            while i < length and (code[i].isalnum() or code[i] == '_'):
                i += 1
            word = code[start:i]

            if word in KEYWORDS:
                tokens.append(Token('KEYWORD', word, line_number))
            elif word in DATA_TYPES:
                tokens.append(Token('DATA_TYPE', word, line_number))
            else:
                tokens.append(Token('IDENTIFIER', word, line_number))

        # Handle numbers
        elif c.isdigit():
            start = i
            while i < length and (code[i].isdigit() or code[i] == '.'):
                i += 1
            number = code[start:i]
            tokens.append(Token('NUMBER', float(number) if '.' in number else int(number),
line_number))

        # Handle strings
        elif c in '"\'':
            quote = c
            i += 1
            start = i
            while i < length and code[i] != quote:
                i += 1
            string = code[start:i]
            tokens.append(Token('STRING', string, line_number))
            i += 1
```

2. Parser Implementation

Purpose:
Creates Abstract Syntax Tree (AST) from tokens

```
# From parser.py
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def parse(self):
        return self.program()

    def program(self):
        statements = []
        while self.current_token():
            statements.append(self.statement())
        return Program(statements)

    def statement(self):
        token = self.current_token()

        if token.type == "IDENTIFIER":
            return self.assignment()
        elif token.type == "KEYWORD":
            if token.value == "if":
                return self.if_statement()
            elif token.value == "while":
                return self.while_statement()
            elif token.value == "for":
                return self.for_statement()

    def if_statement(self):
        self.eat("KEYWORD")  # Consume 'if'
        condition = self.expression()
        self.eat("SYMBOL")  # Expect ':'
        true_branch = self.block()
        false_branch = None

        if self.current_token() and self.current_token().type == "KEYWORD" and
self.current_token().value == "else":
            self.eat("KEYWORD")
            self.eat("SYMBOL")  # Expect ':'
            false_branch = self.block()

        return IfNode(condition, true_branch, false_branch)
```

3. Semantic Analyzer Implementation

Purpose:
Verifies program meaning and type checking.

```python
# From semantic_analyzer.py
class SemanticAnalyzer:
    def __init__(self):
        self.symbol_table = ScopedSymbolTable()

    def analyze(self, node):
        if isinstance(node, AssignmentNode):
            expr_type = self.analyze(node.expression)
            self.symbol_table.declare(node.identifier, expr_type)

        elif isinstance(node, BinaryOpNode):
            left_type = self.analyze(node.left)
            right_type = self.analyze(node.right)
            if left_type != right_type:
                raise SemanticError(f"Type mismatch: {left_type} vs {right_type}")
            return left_type

        elif isinstance(node, IfNode):
            cond_type = self.analyze(node.condition)
            if cond_type != 'bool':
                raise SemanticError(f"Condition must be boolean, got {cond_type}")
            self.analyze(node.true_branch)
            if node.false_branch:
                self.analyze(node.false_branch)
```

4. IR Generator Implementation

Purpose:
Creates intermediate code between Python and C++.

```python
# From ir_generator.py
class IRGenerator:
    def __init__(self):
        self.instructions = []
        self.label_count = 0

    def generate(self, node):
        if isinstance(node, AssignmentNode):
            expr = self.generate(node.expression)
            self.instructions.append(('assign', node.identifier, expr))

        elif isinstance(node, IfNode):
            condition_ir = self.generate(node.condition)
            true_branch_ir = self._generate_block(node.true_branch)
            false_branch_ir = self._generate_block(node.false_branch) if node.false_branch
else None
            self.instructions.append(('if', condition_ir, true_branch_ir, false_branch_ir))

        elif isinstance(node, BinaryOpNode):
            left = self.generate(node.left)
            right = self.generate(node.right)
            return ('binop', node.operator, left, right)
```

5. Code Generator Implementation

Purpose:
Produces final C++ code.

```
# From code_generator.py
class CodeGenerator:
    def __init__(self):
        self.code = []
        self.functions = []
        self.declared_vars = {}
        self.indentation_level = 1

    def generate(self, ir):
        if isinstance(ir, tuple):
            instruction_type = ir[0]

            if instruction_type == 'assign':
                var_name = ir[1]
                expr = self.generate_expr(ir[2])
                var_type = self.infer_var_type(ir[2])

                if var_name not in self.declared_vars:
                    self.code.append(self._indent(f"{var_type} {var_name} = {expr};"))
                    self.declared_vars[var_name] = var_type
                else:
                    self.code.append(self._indent(f"{var_name} = {expr};"))

            elif instruction_type == 'if':
                condition = self.generate_expr(ir[1])
                true_branch = self.generate_block(ir[2])
                false_branch = self.generate_block(ir[3]) if ir[3] else []

                self.code.append(self._indent(f"if ({condition}) {{"))
                self.indentation_level += 1
                self.code.extend(true_branch)
                self.indentation_level -= 1
                self.code.append(self._indent("}"))

                if false_branch:
                    self.code.append(self._indent("else {"))
                    self.indentation_level += 1
                    self.code.extend(false_branch)
                    self.indentation_level -= 1
                    self.code.append(self._indent("}"))

    def get_cpp_code(self):
        header_code = "#include <bits/stdc++.h>\nusing namespace std;\n\n"
        functions_code = "\n".join(self.functions) + "\n\n" if self.functions else ""
        main_code = "int main() {\n"
        main_code += "\n".join(self.code)
        main_code += "\n    return 0;\n}"
        return header_code + functions_code + main_code
```

Future Scope:

This serves as a foundational tool and can be enhanced further by:

- Support for More Python Features
- Enhanced Code Optimization..
- IDE Integration
- Support for Python libraries and their C++ equivalents

Final Statement

The Python to C++ Code Converter stands as a testament to innovative software engineering, offering a comprehensive solution for developers seeking to bridge the gap between Python and C++ programming languages. At its core, the project excels through its sophisticated architecture, featuring a meticulously designed modular system that separates concerns across lexing, parsing, and code generation components. The user experience is elevated through a modern, intuitive GUI interface that incorporates essential features like syntax highlighting, seamless file operations, and real-time error feedback. The converter's robust handling of Python constructs, including accurate type conversions, string manipulations, and control flow structures, ensures reliable and maintainable C++ output. The project's commitment to error handling is evident in its comprehensive error detection and recovery mechanisms, providing clear, actionable feedback to users. This culmination of features, combined with its extensible codebase and user-centric design, makes the converter an invaluable tool for developers looking to port Python applications to C++ while maintaining code quality and functionality. The project not only achieves its primary goal of code conversion but does so in a way that promotes best practices in software development, making it a significant contribution to the developer toolset.

Conclusion

Based on the project files and README, this appears to be a Python to C++ Code Converter project. The project is a GUI application that converts Python code to C++ code with a modern and user-friendly interface.

The project successfully integrates the following core features:

- A lexer (lexer.py) for tokenizing Python code.
- A parser (parser.py) for creating an AST.
- A semantic analyzer (semantic_analyzer.py) for code analysis**.**
- Code generation components (code_generator.py, ir_generator.py)..
- A visualizer (visualizer.py) for the GUI interface.

The project appears to be a complete solution for converting Python code to C++, with a focus on user experience through its GUI interface. The code is well-organized into separate modules handling different aspects of the conversion process, from lexical analysis to code generation.

The project successfully implements a practical tool that can help developers convert Python code to C++, which can be particularly useful for:

- Porting Python applications to C++.
- Learning the differences between Python and C++ syntax.
- Quick prototyping in Python with the ability to convert to C++ for performance improvements.