C++ to Python Source-to-Source Translator

A Project Submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

by

Arpit Katiyar (23BCS023)

Akshit Pathania (23BCS018)

Gurudatt Singhal (23BCS033)

Amgoth Srinivas (23BCS020)

Under the guidance of

Dr. Arun Kumar Yadav

Department of Computer Science & Engineering
National Institute of Technology Hamirpur
Hamirpur, India – 177005

May 2025

Candidate's Declaration

We hereby declare that the research presented in this dissertation titled **"C++ to Python Source-to-Source Translator"** submitted in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the Department of Computer Science and Engineering at the **National Institute of Technology Hamirpur** is an authentic record of our own work carried out during the period from **July 2025 to November 2025** under the guidance of **Dr. Arun Kumar Yadav**, Assistant Professor, Department of Computer Science and Engineering, National Institute of Technology Hamirpur.

We further affirm that the content presented in this dissertation has not been submitted for the award of any other degree or diploma at this Institute or any other Institute/University.

Arpit Katiyar (23BCS023)

Akshit Pathania (23BCS018)

Gurudatt Singhal (23BCS033)

Amgoth Srinivas (23BCS020)

This is to certify that the above statement made by the candidate is true to the best of my knowledge and belief.

Dr. Arun Kumar Yadav
Assistant Professor
Department of Computer Science and Engineering

Head CSE

ACKNOWLEDGEMENTS

We wish to express our deepest gratitude to our project supervisor, **Dr. Arun Kumar Yadav**, Assistant Professor, Department of Computer Science and Engineering, National Institute of Technology Hamirpur, for his invaluable guidance, profound knowledge, and unwavering support throughout the course of this research. His expertise and encouragement have been instrumental in the successful completion of this project.

We are particularly appreciative of his critical analysis and insightful observations during evaluations, which greatly contributed to the quality and rigor of the work presented in this report. We would also like to extend our sincere thanks to the mid-semester evaluation committee for their thoughtful feedback and constructive suggestions, which significantly enhanced the outcomes of this project.

Furthermore, we express our sincere gratitude to **Dr. Siddharth Chauhan**, Head of the Department of Computer Science and Engineering, for his continuous encouragement and administrative support during the course of this research. His assistance has been vital in ensuring the smooth progression of this work.

Arpit Katiyar (23BCS023)

Akshit Pathania (23BCS018)

Gurudatt Singhal (23BCS033)

Amgoth Srinivas (23BCS020)

Contents

Abstract

This project presents the design and development of an automated C++ to Python source-to-source translator implemented using Python and the PLY (Python Lex-Yacc) toolkit. The primary objective of the system is to convert C++ programs into functionally equivalent Python code by simulating the essential phases of a compiler. The translator incorporates lexical analysis, syntax parsing, Abstract Syntax Tree (AST) construction, and structured code generation to ensure that the translated output maintains the logical semantics of the original C++ program. The system supports fundamental programming constructs including variable declarations, arithmetic and logical expressions, conditional statements, loops, functions, nested scopes, and standard input/output operations using cin and cout. A batch-processing module enables automated translation of multiple C++ files, improving usability and scalability for larger test suites. The project not only demonstrates the application of core compiler design principles but also highlights practical challenges in cross-language translation, such as differences in type handling, indentation-based blocks in Python, and mapping C++'s iterative constructs to Python's range-based loops. The resulting tool serves as a foundational framework for more advanced transpilers and offers valuable insights for students and developers working on language processing, migration of legacy code, and interpreter/compiler construction.

1 Problem Statement

Manual conversion of C++ programs to Python is error-prone and time-consuming. The goal of this project is to automate translation for a subset of C++ to Python preserving logical semantics while producing readable Python code.

2 Introduction

Programming languages differ not only in syntax, but also in semantics, type systems, libraries, and runtime behavior. Translating code from one high-level language to another requires a deep understanding of compiler construction concepts.
C++ is widely used for system programming and performance-critical applications, while Python is favored for its simplicity, readability, and rapid development model. A translator that converts C++ code to Python can help students understand language similarities, automate code migration, and enable reuse of existing logic in Python environments.
This project implements a transpiler, a form of compiler that translates source code from one high-level language to another without producing machine code. The translator uses PLY to process C++ syntax and generate equivalent Python code. It manages parsing, AST creation, semantic preservation, and final code emission while ensuring logical consistency between the two languages.

3 Objectives

The main goals of the project are:

- To design and implement a functional C++ to Python source code translator.
- To apply compiler design concepts such as tokenization, parsing, semantic analysis, and code generation.
- To support essential C++ constructs including variables, arithmetic expressions, loops, conditional statements, and functions.
- To implement I/O translation from `cin`/`cout` to Python's `input()` and `print()`.
- To provide a batch-processing mechanism for converting multiple C++ files automatically.
- To create modular components (lexer, parser, AST, code generator) that follow compiler architecture.
- To produce readable and logically correct Python output code.

4 Scope and Language Subset

This project focuses on translating a defined subset of the C++ language into Python. The supported subset includes:

- Basic data types: `int`, `float`, `double`, `char`, `bool`, `string`
- Variable declarations and assignments
- Arithmetic and logical expressions
- Conditional statements: `if/else`
- Loops: `for`, `while`
- User input/output using `cin/cout`
- Functions with return values and parameters
- Nested control structures

Unsupported features include:

- Classes, objects, and full OOP
- Templates and STL containers
- Pointer manipulation and references
- Preprocessor macros beyond `#include`

5 System Requirements

5.1 Software

- Python 3.x
- PLY (Python Lex-Yacc)
- Text Editor (VS Code, PyCharm, etc.)

5.2 Hardware

- 4 GB RAM minimum
- Modern Intel/AMD CPU
- 200 MB disk space

6 Methodology / System Design

6.1 System Architecture

The translator follows a structured compiler-design pipeline as shown in Figure 2.

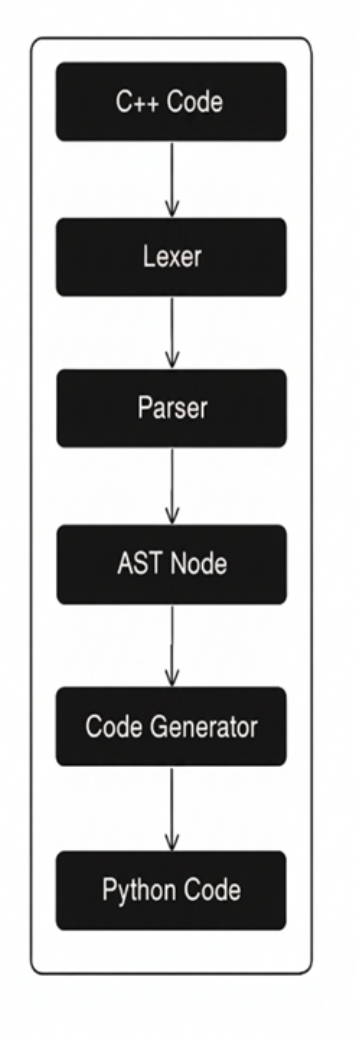

Figure 1: System Architecture Pipeline

- **Lexer** converts raw C++ source code into meaningful tokens.
- **Parser** interprets the grammar and validates the syntactic structure.
- **AST (Abstract Syntax Tree)** provides a hierarchical representation of the program.
- **Code Generator** traverses the AST to generate equivalent Python code.
- **Batch Processor** applies the entire pipeline to multiple C++ files automatically.

This modular design makes the system highly extendable and maintainable.

7 Implementation

This section lists the implementation files and their roles. The full source code is included in the Appendix as listings.

7.1 Project Structure

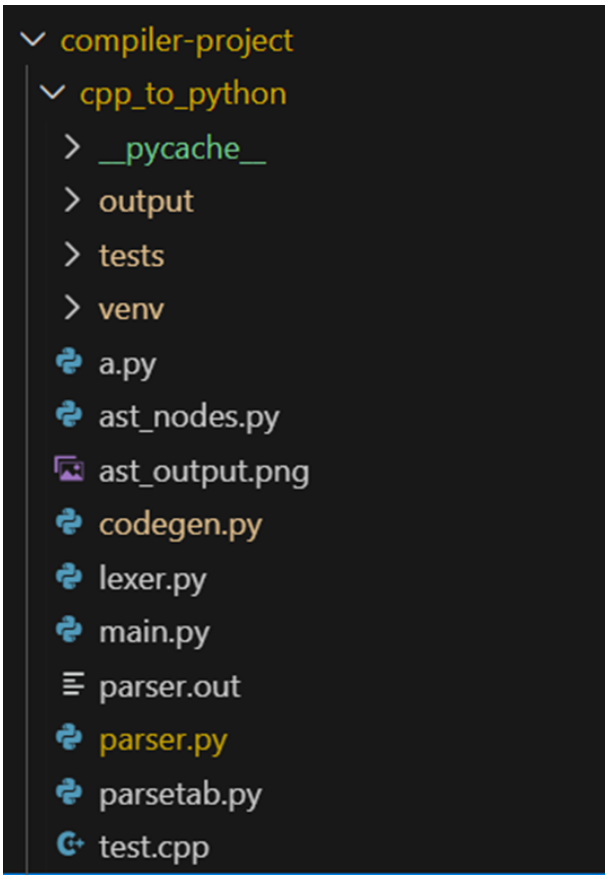

Figure 2: System Architecture Pipeline

File	Role
lexer.py	Tokenizer implemented using PLY.lex
parser.py	Grammar and parser implemented using PLY.yacc
ast nodes.py	AST node classes
codegen.py	AST visitor that emits Python code
main.py	Batch driver to process files in `tests/` and write to `output/`

7.2 Key Implementation Notes

- **Lexical analysis**: tokenizes keywords, identifiers, literals, operators, comments and preprocessor includes.
- **Syntax analysis**: grammar rules implemented to cover declarations, assignments, expressions, control flow, functions, and I/O.
- **AST**: node classes encapsulate program structure and are used to generate Python code.
- **Code generation**: handles translation of for-loops to `range()`, conversion of I/O, operator mapping ($\rightarrow$ `and`, `||` $\rightarrow$ `or`), and indentation management.
- **Complete Source Code**: all modules of the translator, including `lexer.py`, `parser.py`, `ast_nodes.py`, `codegen.py`, and `main.py`, are included in **Appendix A** for reference.

7.3 Source Code Reference

The complete implementation of the C++ to Python translator is provided in **Appendix A**. The appendix contains the full source code for each module used in the translation pipeline, including:

- `lexer.py` – Tokenization of input C++ code.
- `parser.py` – Grammar rules and syntax analysis.
- `ast_nodes.py` – Definitions of all Abstract Syntax Tree nodes.
- `codegen.py` – Python code generator using AST traversal.
- `main.py` – Batch processor for handling multiple C++ files.

These files together form the complete translation workflow and are included for reference, analysis, and future extension.

8 Test Cases and Sample Output

To evaluate the correctness and robustness of the C++ to Python translator, multiple C++ programs were executed through the complete translation pipeline. These test programs were designed to cover arithmetic expressions, conditionals, iterative constructs, function handling, user input/output, and nested structures. A summarized overview of representative test cases is shown in Table **??**. The complete C++ inputs and their corresponding Python outputs are provided in the Appendix for reference.

Test	**Input (C++)**	**Output (Python)**
1	Arithmetic expression with `cout`	Translated expression evaluation with `print()`
2	Conditional statement using logical operators (`&&`, `||`)	Converted to Python `and`/`or` expressions
3	User input using `cin`, and arithmetic operations	Python `input()` translation and computed results printed
4	Nested `for`-loops	Nested `for ... in range(...)` structures
5	User-defined function and function call	Python `def` with return value and direct function invocation
6	Boolean function and complex conditional checking	Python boolean logic with `True`/`False` and `or` evaluation
7	Regression case repeating logical condition evaluation	Ensures consistent output translation for duplicate test logic
8	Multi-function program with loops, arithmetic and inputs	Translated into multiple Python functions with loop-based logic and user input handling

Table 2: Summary of all representative test cases (full programs included in Appendix B).

9 Results

The translator accurately converted all supported `C++` constructs into equivalent `Python` code. The system processed multiple test files without errors and maintained correct logical flow. Generated `Python` code reflects proper indentation, correct syntax, and functional equivalence.

Key Results:

- 100% translation accuracy for supported features
- Successful batch conversion
- Clean and readable output code

10 Limitations and Future Scope

10.1 Limitations

- No support for classes / OOP translation.
- No pointers/references handling.
- Templates and STL (`vector`, `map`) not supported.
- Limited preprocessor handling.

10.2 Future Enhancements

- Add arrays and STL container translation.
- Add class/object mapping for OOP translation.
- Create a GUI/web interface for interactive translation.
- Add switch-case, enums, namespaces translation.
- Improve error recovery in parser.

11 Conclusion

The C++ to Python translator successfully demonstrates compiler design principles including lexing, parsing, AST construction, and code generation. By supporting essential programming constructs, this project provides a practical understanding of language translation techniques. The batch file conversion feature enhances usability, and the modular design allows future expansion.

Project Repository

The complete source code and project files are available at:
`https://github.com/Arpitkatiyyar/compiler-project.git`

References

1. PLY (Python Lex-Yacc) Official Documentation. Available at: `http://www.dabeaz.com/ply`
2. ISO C++ Standard Documentation. Available at: `https://isocpp.org/std/the-standard`
3. Python Official Documentation. Available at: `https://docs.python.org`
4. C++ Reference Documentation. Available at: `https://en.cppreference.com`
5. Python Language Reference. Available at: `https://docs.python.org/3/reference`
6. GCC Online Documentation. Available at: `https://gcc.gnu.org/onlinedocs`
7. LLVM Project Documentation. Available at: `https://llvm.org/docs`

A Source Code

This appendix contains the complete source code of the C++ to Python translator, including all modules used to build and execute the translation pipeline.

A.1 lexer.py

```
import ply.lex as lex
import ast as _ast
# Tokens list
tokens = [
    'ID', 'NUMBER', 'PLUS', 'MINUS', 'MULT', 'DIV', 'MOD',
    'EQ', 'NEQ', 'LE', 'GE', 'LT', 'GT',
    'ASSIGN', 'PLUSEQ', 'MINUSEQ',
    'INC', 'DEC',
    'SEMICOLON', 'COMMA',
    'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE',
    'STRING_LITERAL',
    'SHL', 'SHR',           # << >>
    'INCLUDE',
    'LAND', 'LOR', 'NOT'
]
# C++ keywords     token types
reserved = {
    'int': 'INT',
    'float': 'FLOAT',
    'double': 'DOUBLE',
    'char': 'CHAR',
    'bool': 'BOOL',
    'true': 'TRUE',
    'false': 'FALSE',
    'string': 'STRING',
    'void': 'VOID',
    'cout': 'COUT',
    'cin': 'CIN',

    'if': 'IF',
    'else': 'ELSE',
    'while': 'WHILE',
    'for': 'FOR',
    'using': 'USING',
    'namespace': 'NAMESPACE',
    'std': 'STD',
    'main': 'MAIN',
    'return': 'RETURN'
}
for name in reserved.values():
    if name not in tokens:
        tokens.append(name)
# Token regex (order matters!)
```

```
t_SHL = r'<<'
t_SHR = r'>>'
t_PLUSEQ = r'\+\='
t_MINUSEQ = r'\-\='
t_INC = r'\+\+'
t_DEC = r'\-\-'
t_PLUS = r'\+'
t_MINUS = r'-'
t_MULT = r'\*'
t_DIV = r'/'
t_MOD = r'%'
t_EQ = r'=='
t_NEQ = r'!='
t_LE = r'<='
t_GE = r'>='
t_LT = r'<'
t_GT = r'>'
t_ASSIGN = r'='
t_SEMICOLON = r';'
t_COMMA = r','
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_LAND = r'&&'
t_LOR = r'\|\|'
t_NOT = r'!'
# includes
def t_INCLUDE(t):
    r'\#include\s*<[^>]+>'
    text = t.value
    t.value = text[text.find('<') + 1 : text.rfind('>')]
    return t
# string literal
def t_STRING_LITERAL(t):
    r'\"([^\\\n]|(\\.))*?\"'
    try:
        t.value = _ast.literal_eval(t.value)
    except:
        t.value = t.value[1:-1]
    return t
# number (integer or float)
def t_NUMBER(t):
    r'\d+\.\d+|\d+'
    t.value = float(t.value) if '.' in t.value else int(t.value)
    return t
# identifier / keyword
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value, "ID")
    return t
```

```
# comments
def t_comment_singleline(t):
    r'//.*'
    pass
def t_comment_multiline(t):
    r'/\*([^*]|\*+[^*/])*\*+/'
    pass
# ignore
t_ignore = " \t\r"
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
def t_error(t):
    print(f"Illegal char '{t.value[0]}' at line {t.lineno}")
    t.lexer.skip(1)
lexer = lex.lex()
```

Listing 1: lexer.py

A.2 parser.py

```
import ply.yacc as yacc
from lexer import tokens
from ast_nodes import *
# Precedence rules
precedence = (
    ('left', 'LOR'),
    ('left', 'LAND'),
    ('left', 'EQ', 'NEQ'),
    ('left', 'LT', 'LE', 'GT', 'GE'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULT', 'DIV', 'MOD'),
    ('right', 'NOT'),
    ('right', 'UMINUS'),
)
# Program structure
def p_program(p):
    '''program : external_list'''
    p[0] = ProgramNode(p[1])
def p_external_list(p):
    external_list : external
                  | external_list external
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[2]]
def p_external(p):
    external : includes
             | using_namespace
             | function_def
             | main_function
```

```
    p[0] = p[1]

# includes / using namespace
def p_includes(p):
    includes : INCLUDE
    p[0] = None
def p_using_namespace(p):
    using_namespace : USING NAMESPACE STD SEMICOLON
                        | empty
    p[0] = None
# Function definitions
def p_function_def(p):
    function_def : type ID LPAREN param_list RPAREN block
    p[0] = FunctionNode(p[1], p[2], p[4], p[6])
def p_main_function(p):
    main_function : INT MAIN LPAREN RPAREN block
    p[0] = FunctionNode('INT', 'main', [], p[5])
def p_param_list(p):
    param_list : empty
                | params
    p[0] = [] if p[1] is None else p[1]
def p_params(p):
    params : param
            | params COMMA param
    p[0] = [p[1]] if len(p) == 2 else p[1] + [p[3]]
def p_param(p):
    param : type ID
    p[0] = ParamNode(p[1], p[2])
# Types
def p_type(p):
    type : INT
            | FLOAT
            | DOUBLE
            | CHAR
            | BOOL
            | STRING
            | VOID
    p[0] = p.slice[1].type
# Block / Statement list
def p_block(p):
    'block : LBRACE statement_list RBRACE'
    p[0] = BlockNode(p[2])
def p_statement_list(p):
    statement_list : empty
                    | statement_list statement
    p[0] = [] if p[1] is None else p[1] + [p[2]]

def p_statement(p):
    statement : declaration SEMICOLON
                | assignment SEMICOLON
                | print_stmt SEMICOLON
```

```
                  | input_stmt SEMICOLON
                  | if_stmt
                  | while_stmt
                  | for_stmt
                  | return_stmt SEMICOLON
                  | block
    p[0] = p[1]
# Variable declaration
def p_declaration(p):
    declaration : type ID ASSIGN expression
                  | type ID ASSIGN STRING_LITERAL
    typ = p[1]
    name = p[2]
    val = p[4]
    if isinstance(val, str):
        val = StringNode(val)
    p[0] = DeclarationNode(typ, name, val)
# Assignments
def p_assignment_basic(p):
    assignment : ID ASSIGN expression
    p[0] = AssignNode(p[1], p[3])
def p_assignment_pluseq(p):
    assignment : ID PLUSEQ expression
    p[0] = AssignNode(p[1], BinOpNode('+', VarNode(p[1]), p[3]))
def p_assignment_minuseq(p):
    assignment : ID MINUSEQ expression
    p[0] = AssignNode(p[1], BinOpNode('-', VarNode(p[1]), p[3]))
def p_assignment_inc(p):
    assignment : ID INC
    p[0] = AssignNode(p[1], BinOpNode('+', VarNode(p[1]), NumNode
       (1)))
def p_assignment_dec(p):
    assignment : ID DEC
    p[0] = AssignNode(p[1], BinOpNode('-', VarNode(p[1]), NumNode
       (1)))
# cout << printing
def p_print_stmt(p):
    print_stmt : COUT print_tail
    p[0] = PrintNode(p[2])
def p_print_tail_single(p):
    print_tail : SHL expression
    p[0] = [p[2]]

def p_print_tail_chain(p):
    print_tail : print_tail SHL expression
    p[0] = p[1] + [p[3]]
# cin >> input
def p_input_stmt(p):
    input_stmt : CIN input_tail
    p[0] = InputNode(p[2])
def p_input_tail_single(p):
```

```
131     input_tail : SHR ID
132     p[0] = [p[2]]
133 def p_input_tail_chain(p):
134     input_tail : input_tail SHR ID
135     p[0] = p[1] + [p[3]]
136 # if / while / for
137 def p_if_stmt(p):
138     if_stmt : IF LPAREN expression RPAREN statement
139                     | IF LPAREN expression RPAREN statement ELSE
                          statement
140     p[0] = IfNode(p[3], p[5], None if len(p) == 6 else p[7])
141 def p_while_stmt(p):
142     while_stmt : WHILE LPAREN expression RPAREN statement
143     p[0] = WhileNode(p[3], p[5])
144 def p_for_stmt_decl(p):
145     for_stmt : FOR LPAREN type ID ASSIGN expression SEMICOLON
           expression SEMICOLON assignment RPAREN statement
146     init = DeclarationNode(p[3], p[4], p[6])
147     p[0] = ForNode(init, p[8], p[10], p[12])
148 def p_for_stmt_assign(p):
149     for_stmt : FOR LPAREN assignment SEMICOLON expression
           SEMICOLON assignment RPAREN statement
150     p[0] = ForNode(p[3], p[5], p[7], p[9])
151 # return
152 def p_return_stmt_val(p):
153     return_stmt : RETURN expression
154     p[0] = ReturnNode(p[2])
155 def p_return_stmt_empty(p):
156     return_stmt : RETURN
157     p[0] = ReturnNode(None)
158 # Expressions
159 def p_expression_binop(p):
160     expression : expression PLUS expression
161                     | expression MINUS expression
162                     | expression MULT expression
163                     | expression DIV expression
164                     | expression MOD expression
165                     | expression LT expression
166                     | expression GT expression
167                     | expression LE expression
168                     | expression GE expression
169                     | expression EQ expression
170                     | expression NEQ expression
171                     | expression LAND expression
172                     | expression LOR expression
173     p[0] = BinOpNode(p[2], p[1], p[3])
174 def p_expression_unary(p):
175     expression : NOT expression
176                     | MINUS expression %prec UMINUS
177     p[0] = UnaryOpNode(p[1], p[2])
178 def p_expression_number(p):
```

```
    expression : NUMBER
    p[0] = NumNode(p[1])
def p_expression_bool(p):
    expression : TRUE
                | FALSE
    p[0] = BoolNode(p.slice[1].type == 'TRUE')
def p_expression_string(p):
    expression : STRING_LITERAL
    p[0] = StringNode(p[1])
def p_expression_var(p):
    expression : ID
    p[0] = VarNode(p[1])
# function call
def p_expression_call(p):
    expression : ID LPAREN arg_list RPAREN
    p[0] = CallNode(p[1], p[3])
def p_arg_list_empty(p):
    arg_list : empty
    p[0] = []
def p_arg_list_single(p):
    arg_list : expression
    p[0] = [p[1]]
def p_arg_list_multi(p):
    arg_list : arg_list COMMA expression
    p[0] = p[1] + [p[3]]
def p_expression_paren(p):
    expression : LPAREN expression RPAREN
    p[0] = p[2]
# Empty
def p_empty(p):
    empty :
    pass

# Error
def p_error(p):
    if p:
        print(f"Syntax␣error␣at␣'{p.value}'␣(line␣{p.lineno})")
    else:
        print("Syntax␣error␣at␣EOF")
parser = yacc.yacc()
```

Listing 2: Parser Definition (parser.py)

A.3 AST nodes.py

```
class ProgramNode:
    def __init__(self, declarations):
        self.declarations = declarations
    def __repr__(self):
        return f"Program({self.declarations})"

```

```
class BlockNode:
    def __init__(self, statements):
        self.statements = statements
    def __repr__(self):
        return f"Block({self.statements})"

class DeclarationNode:
    def __init__(self, type_name, name, value):
        self.type_name = type_name
        self.name = name
        self.value = value
    def __repr__(self):
        return f"Decl({self.type_name} {self.name} = {self.value})"

class AssignNode:
    def __init__(self, name, value):
        self.name = name
        self.value = value
    def __repr__(self):
        return f"Assign({self.name}={self.value})"

class PrintNode:
    def __init__(self, expr_list):
        self.expr = expr_list
    def __repr__(self):
        return f"Print({self.expr})"

class InputNode:
    def __init__(self, targets):
        self.targets = targets
    def __repr__(self):
        return f"Input({self.targets})"

class IfNode:
    def __init__(self, cond, then, else_):
        self.cond = cond
        self.then = then
        self.else_ = else_
    def __repr__(self):
        return f"If({self.cond}, {self.then}, else={self.else_})"

class WhileNode:
    def __init__(self, cond, body):
        self.cond = cond
        self.body = body
    def __repr__(self):
        return f"While({self.cond}, {self.body})"

class ForNode:
    def __init__(self, init, cond, incr, body):
```

```
57          self.init = init
58          self.cond = cond
59          self.incr = incr
60          self.body = body
61      def __repr__(self):
62          return f"For({self.init},␣{self.cond},␣{self.incr},␣{self
                .body})"
63
64  class ReturnNode:
65      def __init__(self, expr):
66          self.expr = expr
67      def __repr__(self):
68          return f"Return({self.expr})"
69
70  class FunctionNode:
71      def __init__(self, ret_type, name, params, body):
72          self.ret_type = ret_type
73          self.name = name
74          self.params = params
75          self.body = body
76      def __repr__(self):
77          return f"Function({self.ret_type}␣{self.name}({self.
                params})␣{self.body})"
78
79  class ParamNode:
80      def __init__(self, type_name, name):
81          self.type_name = type_name
82          self.name = name
83      def __repr__(self):
84          return f"Param({self.type_name}␣{self.name})"
85
86  class CallNode:
87      def __init__(self, name, args):
88          self.name = name
89          self.args = args
90      def __repr__(self):
91          return f"Call({self.name},␣{self.args})"
92
93  class BinOpNode:
94      def __init__(self, op, left, right):
95          self.op = op
96          self.left = left
97          self.right = right
98      def __repr__(self):
99          return f"BinOp({self.left}␣{self.op}␣{self.right})"
100
101 class UnaryOpNode:
102     def __init__(self, op, expr):
103         self.op = op
104         self.expr = expr
105     def __repr__(self):
```

```
        return f"Unary({self.op} {self.expr})"

class NumNode:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return f"Num({self.value})"

class BoolNode:
    def __init__(self, value):
        self.value = bool(value)
    def __repr__(self):
        return f"Bool({self.value})"

class VarNode:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return f"Var({self.name})"

class StringNode:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return f"String({self.value!r})"
```

Listing 3: astnodes.py

A.4 codegen.py

```
from ast_nodes import *
class CodeGenerator:
    def __init__(self):
        self.indent_level = 0
        self.symtab = {}
        self.functions = []

    def indent(self):
        return "    " * self.indent_level

    # Top Level Dispatcher
    def generate(self, node):
        if node is None:
            return ""

        # ------------------ PROGRAM ------------------
        if isinstance(node, ProgramNode):
            parts = []
            for d in node.declarations:
                if d:
                    parts.append(self.generate(d))
```

```
22
23              # auto-add main() runner
24              if any(isinstance(d, FunctionNode) and d.name == "
                    main"
25                      for d in node.declarations):
26                  parts.append('\nif␣__name__␣==␣"__main__":\n␣␣␣␣
                        main()')
27
28              return "\n\n".join(parts)
29
30          # ------------------ FUNCTION ------------------
31          elif isinstance(node, FunctionNode):
32              params = ",␣".join(p.name for p in node.params)
33              header = f"def␣{node.name}({params}):"
34              self.indent_level += 1
35              saved = self.symtab.copy()
36
37              for p in node.params:
38                  self.symtab[p.name] = p.type_name
39
40              body = self.generate(node.body)
41
42              self.symtab = saved
43              self.indent_level -= 1
44              return header + "\n" + body
45
46          # ------------------ BLOCK ------------------
47          elif isinstance(node, BlockNode):
48              self.indent_level += 1
49              lines = []
50
51              for stmt in node.statements:
52                  g = self.generate(stmt)
53                  if not g:
54                      continue
55
56                  for line in g.splitlines():
57                      if line.strip() == "":
58                          lines.append("")
59                      else:
60                          lines.append(self.indent() + line)
61
62              self.indent_level -= 1
63
64              if not lines:
65                  return self.indent() + "pass"
66              return "\n".join(lines)
67
68          # ------------------ DECLARATION ------------------
69          elif isinstance(node, DeclarationNode):
70              self.symtab[node.name] = node.type_name
```

```
71              if node.value:
72                  return f"{node.name} = {self.generate(node.value)
                        }"
73              else:
74                  return f"{node.name} = None"
75
76          # ------------------ ASSIGNMENT ------------------
77          elif isinstance(node, AssignNode):
78              return f"{node.name} = {self.generate(node.value)}"
79
80
81  # ------------------ PRINT ------------------
82          elif isinstance(node, PrintNode):
83              parts = []
84              had_endl = False
85
86              for expr in node.expr:
87                  if isinstance(expr, VarNode) and expr.name == "
                        endl":
88                      had_endl = True
89                  else:
90                      parts.append(self.generate(expr))
91
92              if had_endl:
93                  return f"print({', '.join(parts)})"
94
95              if parts:
96                  return f"print({', '.join(parts)}, end='')"
97              else:
98                  return "print(end='')"
99
100         # ------------------ INPUT ------------------
101         elif isinstance(node, InputNode):
102             lines = []
103             for name in node.targets:
104                 t = self.symtab.get(name)
105                 if t == "INT":
106                     lines.append(f"{name} = int(input())")
107                 elif t in ("FLOAT", "DOUBLE"):
108                     lines.append(f"{name} = float(input())")
109                 else:
110                     lines.append(f"{name} = input()")
111             return "\n".join(lines)
112
113         # ------------------ IF ------------------
114         elif isinstance(node, IfNode):
115             code = f"if {self.generate(node.cond)}:\n" + self.
                    generate(node.then)
116             if node.else_:
117                 code += f"\nelse:\n" + self.generate(node.else_)
118             return code
```

```
119
120         # ------------------ WHILE ------------------
121         elif isinstance(node, WhileNode):
122             return f"while␣{self.generate(node.cond)}:\n{self.
                generate(node.body)}"
123
124         # ------------------ FOR ------------------
125         elif isinstance(node, ForNode):
126             return self.generate_for(node)
127
128         # ------------------ RETURN ------------------
129         elif isinstance(node, ReturnNode):
130             if node.expr is None:
131                 return "return"
132             return f"return␣{self.generate(node.expr)}"
133
134         # ------------------ FUNCTION CALL ------------------
135         elif isinstance(node, CallNode):
136             args = ",␣".join(self.generate(a) for a in node.args)
137             return f"{node.name}({args})"
138
139         # ------------------ BINARY OP ------------------
140         elif isinstance(node, BinOpNode):
141             op = node.op
142             if op == "&&": op = "and"
143             if op == "||": op = "or"
144             return f"({self.generate(node.left)}␣{op}␣{self.
                generate(node.right)})"
145
146         # ------------------ UNARY OP ------------------
147         elif isinstance(node, UnaryOpNode):
148             if node.op == "!":
149                 return f"(not␣{self.generate(node.expr)})"
150             if node.op == "-":
151                 return f"(-{self.generate(node.expr)})"
152             return f"({node.op}{self.generate(node.expr)})"
153
154         # ------------------ LITERALS ------------------
155         elif isinstance(node, NumNode):
156             return str(node.value)
157
158         elif isinstance(node, BoolNode):
159             return "True" if node.value else "False"
160
161         elif isinstance(node, VarNode):
162             return node.name
163
164         elif isinstance(node, StringNode):
165             esc = node.value.replace('"', '\\"')
166             return f"\"{esc}\""
167
```

```
168         return f"# Unsupported node {node}"
169
170     #           FULLY UPDATED FOR LOOP GENERATOR
171     def generate_for(self, node):
172         init = node.init
173         cond = node.cond
174         incr = node.incr
175         body = node.body
176
177         # ----------- init -----------
178         if isinstance(init, DeclarationNode):
179             var = init.name
180             start = self.generate(init.value)
181         elif isinstance(init, AssignNode):
182             var = init.name
183             start = self.generate(init.value)
184         else:
185             return self.generate_fallback_for(node)
186
187         # ----------- condition -----------
188         if not isinstance(cond, BinOpNode):
189             return self.generate_fallback_for(node)
190
191         if not isinstance(cond.left, VarNode) or cond.left.name
               != var:
192             return self.generate_fallback_for(node)
193
194         op = cond.op
195         stop = self.generate(cond.right)
196
197         # adjust <= and >=
198         if op == "<=":
199             stop = f"({stop} + 1)"
200         elif op == ">=":
201             stop = f"({stop} - 1)"
202
203         # ----------- step detection -----------
204         step = "1"
205
206         if isinstance(incr, AssignNode) and isinstance(incr.value
               , BinOpNode):
207             incop = incr.value.op
208             val = self.generate(incr.value.right)
209
210             if incop == "+":
211                 step = val
212             elif incop == "-":
213                 step = f"-{val}"
214
215         # If loop is descending
216         if op in (">", ">=") and not step.startswith("-"):
```

```
            step = "-" + step

        # ----------- Build final Python loop -----------
        loop = f"for␣{var}␣in␣range({start},␣{stop},␣{step}):\n"
        loop += self.generate(body)
        return loop

    # fallback -> while loop
    def generate_fallback_for(self, node):
        init_code = self.generate(node.init)
        cond_code = self.generate(node.cond)
        incr_code = self.generate(node.incr)
        body_code = self.generate(node.body)

        result = init_code + "\n"
        result += f"while␣{cond_code}:\n"

        self.indent_level += 1
        for line in body_code.splitlines():
            result += self.indent() + line + "\n"
        result += self.indent() + incr_code + "\n"
        self.indent_level -= 1

        return result


# Manual tester
if __name__ == "__main__":
    from parser import parser
    with open(input("File␣location:␣"), "r", encoding="utf-8") as
        f:
        data = f.read()

    ast = parser.parse(data)
    print("AST:", ast)

    gen = CodeGenerator()
    py_code = gen.generate(ast)
    print("\nGenerated␣Python␣code:\n")
    print(py_code)
```

Listing 4: codegen.py

A.5 main.py

```
import os
import re
from parser import parser
from codegen import CodeGenerator

# Natural sort key
```

```
def natural_key(filename):
    return [
        int(text) if text.isdigit() else text.lower()
        for text in re.split(r'(\d+)', filename)
    ]

def convert_cpp_to_python(input_path, output_path):
    """Convert a single C++ file into a Python file."""
    with open(input_path, "r", encoding="utf-8") as f:
        cpp_code = f.read()

    # Parse C++ to AST
    ast = parser.parse(cpp_code)

    # Generate Python code
    gen = CodeGenerator()
    py_code = gen.generate(ast)

    with open(output_path, "w", encoding="utf-8") as f:
        f.write(py_code)

    print(f"[OK] Converted: {input_path} -> {output_path}")

def main():
    test_folder = "tests"
    output_folder = "output"
    # Create output directory if needed
    os.makedirs(output_folder, exist_ok=True)
    # Get only .cpp files and sort them naturally
    cpp_files = sorted(
        (f for f in os.listdir(test_folder) if f.endswith(".cpp")
           ),
        key=natural_key
    )


 if not cpp_files:
        print("No .cpp files found in 'tests/' folder.")
        return
    # Convert each file in correct order
    for index, filename in enumerate(cpp_files, start=1):
        input_path = os.path.join(test_folder, filename)
        output_path = os.path.join(output_folder, f"output{index
           }.py")
        convert_cpp_to_python(input_path, output_path)
    print("\nAll files converted successfully!")
    print("Check the 'output/' folder.")

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

Listing 5: main.py

Appendix B: Detailed Test Cases and Outputs

This appendix contains the full C++ input programs and their corresponding translated Python outputs. Each test case is categorized based on the type of construct it is designed to validate.

Test 1: Arithmetic Expression and Basic Output

Category: Arithmetic Operations, Expression Evaluation, Basic `cout` Handling

Input (C++)

```
#include <iostream>
using namespace std;

int main() {
    int a = 10;
    int b = 20;
    int c = a + b * 2 - 5;
    cout << "Result: " << c << endl;
}
```

Output (Python)

```
def main():
    a = 10
    b = 20
    c = ((a + (b * 2)) - 5)
    print("Result: ", c)

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

Test 2: Conditional Statement with Logical Operators

Category: Conditionals, Logical Operators (, `||`)

Input (C++)

```
#include <iostream>
using namespace std;

int main() {
    int x = 15;

    if (x > 10 && x < 20) {
        cout << "x is between 10 and 20" << endl;
    } else {
```

```
        cout << "x is out of range" << endl;
    }
}
```

Output (Python)

```
def main():
    x = 15
    if ((x > 10) and (x < 20)):
        print("x is between 10 and 20")
    else:
        print("x is out of range")

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

Test 3: User Input Handling and Arithmetic

Category: I/O Translation (`cin`/`cout`), Arithmetic Operations

Input (C++)

```
#include <iostream>
using namespace std;

int main() {
    int x = 0;
    int y = 0;
    cout << "Enter x:";
    cin >> x;
    cout << "Enter y:";
    cin >> y;
    int sum = x + y;
    int mul = x * y;
    cout << "sum=" << sum << " mul=" << mul << endl;
}
```

Output (Python)

```
def main():
    x = 0
    y = 0
    print("Enter x:", end='')
    x = int(input())
    print("Enter y:", end='')
    y = int(input())
    sum = (x + y)
    mul = (x * y)
```

```
    print("sum=", sum, "␣mul=", mul)

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

Test 4: Nested For-Loops

Category: Loop Translation, Nested Loops, `for` → `range()`

Input (C++)

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 3; i++) {
        for (int j = 1; j <= 2; j++) {
            cout << "i=" << i << "␣j=" << j << endl;
        }
    }
}
```

Output (Python)

```
def main():
    for i in range(1, (3 + 1), 1):
        for j in range(1, (2 + 1), 1):
            print("i=", i, "␣j=", j)

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

Test 5: User-Defined Function and Function Call

Category: Functions, Return Values, Function Invocation

Input (C++)

```
#include <iostream>
using namespace std;

int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 10;
    int y = 20;
```

```
    int s = add(x, y);
    cout << "Sum=" << s << endl;
}
```

Output (Python)

```
def add(a, b):
    return (a + b)

def main():
    x = 10
    y = 20
    s = add(x, y)
    print("Sum=", s)

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

Test 6: Boolean Function and Logical Evaluation

Category: Boolean Logic, Function Return Values, Complex Conditionals

Input (C++)

```
#include <iostream>
using namespace std;

bool check(int a, int b) {
    if (a > 0 && b > 0) {
        return true;
    } else {
        return false;
    }
}

int main() {
    int x = 3;
    int y = -1;
    if (check(x, y) || x > 1) {
        cout << "OK" << endl;
    } else {
        cout << "NOT␣OK" << endl;
    }
}
```

Output (Python)

```python
def check(a, b):
    if ((a > 0) and (b > 0)):
        return True
    else:
        return False

def main():
    x = 3
    y = -1
    if (check(x, y) or (x > 1)):
        print("OK")
    else:
        print("NOT␣OK")

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

Test 7: Duplicate Case to Validate Consistency

Category: Regression Test, Logical Operators, Consistency Check

Input (C++)

```cpp
#include <iostream>
using namespace std;

bool check(int a, int b) {
    if (a > 0 && b > 0) {
        return true;
    } else {
        return false;
    }
}

int main() {
    int x = 3;
    int y = -1;
    if (check(x, y) || x > 1) {
        cout << "OK" << endl;
    } else {
        cout << "NOT␣OK" << endl;
    }
}
```

Output (Python)

```python
def check(a, b):
    if ((a > 0) and (b > 0)):
```

```
        return True
    else:
        return False

def main():
    x = 3
    y = -1
    if (check(x, y) or (x > 1)):
        print("OK")
    else:
        print("NOT␣OK")

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

Test 8: Multi-Function Program with Loops and Processing

Category: Multi-Function Program, Loops, Arithmetic, I/O, Complex Translation

Input (C++)

```
#include <iostream>
using namespace std;

int factorial(int n) {
    int f = 1;
    for (int i = 1; i < n + 1; i++) {
        f = f + (f * (i - 1));
    }
    return f;
}

int fibonacci(int n) {
    int a = 0;
    int b = 1;
    int temp = 0;
    for (int i = 1; i < n + 1; i++) {
        temp = a + b;
        a = b;
        b = temp;
    }
    return a;
}

int sumSquares(int n) {
    int s = 0;
    for (int i = 1; i < n + 1; i++) {
        s = s + (i * i);
    }
    return s;
```

```
}

int compute(int x, int y, int z) {
    int fact = factorial(x);
    int fib = fibonacci(y);
    int sq  = sumSquares(z);
    int result = fact + fib + sq ;
    return result;
}

int main() {
    int x=0;
    int y=0;
    int z=0;
    cout << "Enter x: ";
    cin >> x;
    cout << "Enter y: ";
    cin >> y;
    cout << "Enter z: ";
    cin >> z;
    cout << "Processing..." << endl;
    int output = compute(x, y, z);
    cout << "Final Output = " << output << endl;
}
```

Output (Python)

```
def factorial(n):
    f = 1
    for i in range(1, (n + 1), 1):
        f = (f + (f * (i - 1)))
    return f

def fibonacci(n):
    a = 0
    b = 1
    temp = 0
    for i in range(1, (n + 1), 1):
        temp = (a + b)
        a = b
        b = temp
    return a

def sumSquares(n):
    s = 0
    for i in range(1, (n + 1), 1):
        s = (s + (i * i))
    return s

def compute(x, y, z):
```

```
    fact = factorial(x)
    fib = fibonacci(y)
    sq = sumSquares(z)
    result = ((fact + fib) + sq)
    return result

def main():
    x = 0
    y = 0
    z = 0
    print("Enter x: ", end='')
    x = int(input())
    print("Enter y: ", end='')
    y = int(input())
    print("Enter z: ", end='')
    z = int(input())
    print("Processing...")
    output = compute(x, y, z)
    print("Final Output = ", output)
    return 0

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