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 (23BCS023)

Akshit Pathania (23BCS018)

Gurudutt 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

November 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 (23BCS023)

Akshit Pathania (23BCS018)

Gurudutt 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

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.
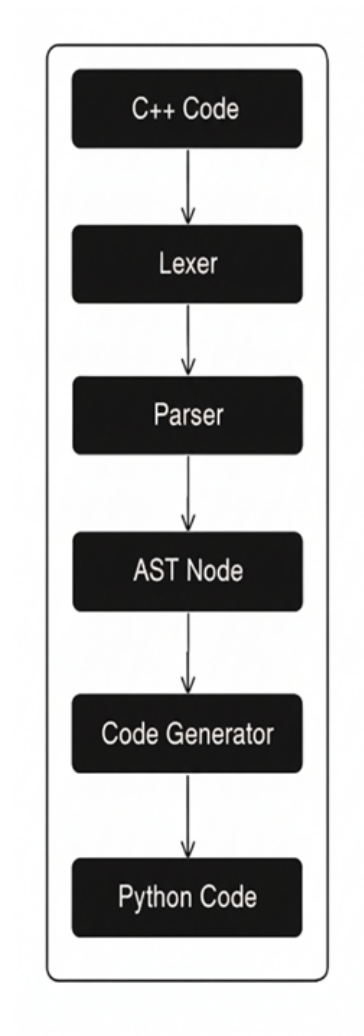
Figure 1: System Architecture Pipeline

- **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
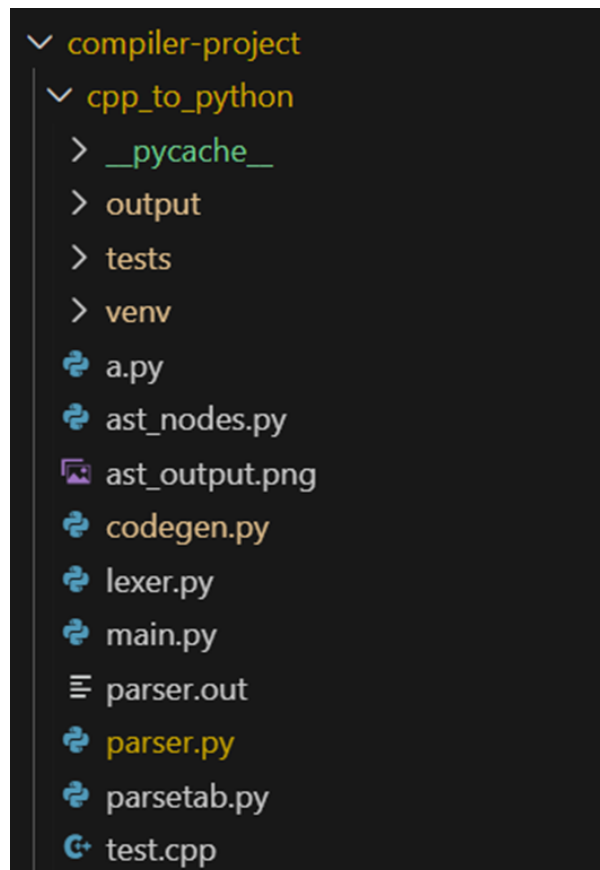
Figure 2: System Architecture Pipeline

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 2. 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 in-
cluding 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

```python
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!)
```

```
44  t_SHL = r'<<'
45  t_SHR = r'>>'
46  t_PLUSEQ = r'\+\='
47  t_MINUSEQ = r'\-\='
48  t_INC = r'\+\+'
49  t_DEC = r'\-\-'
50  t_PLUS = r'\+'
51  t_MINUS = r'-'
52  t_MULT = r'\*'
53  t_DIV = r'/'
54  t_MOD = r'%'
55  t_EQ = r'=='
56  t_NEQ = r'!='
57  t_LE = r'<='
58  t_GE = r'>='
59  t_LT = r'<'
60  t_GT = r'>'
61  t_ASSIGN = r'='
62  t_SEMICOLON = r';'
63  t_COMMA = r','
64  t_LPAREN = r'\('
65  t_RPAREN = r'\)'
66  t_LBRACE = r'\{'
67  t_RBRACE = r'\}'
68  t_LAND = r'&&'
69  t_LOR = r'\|\|'
70  t_NOT = r'!'
71  # includes
72  def t_INCLUDE(t):
73      r'\#include\s*<[^>]+>'
74      text = t.value
75      t.value = text[text.find('<') + 1 : text.rfind('>')]
76      return t
77  # string literal
78  def t_STRING_LITERAL(t):
79      r'\"([^\\\n]|(\\.))*?\"'
80      try:
81          t.value = _ast.literal_eval(t.value)
82      except:
83          t.value = t.value[1:-1]
84      return t
85  # number (integer or float)
86  def t_NUMBER(t):
87      r'\d+\.\d+|\d+'
88      t.value = float(t.value) if '.' in t.value else int(t.value)
89      return t
90  # identifier / keyword
91  def t_ID(t):
92      r'[a-zA-Z_][a-zA-Z0-9_]*'
93      t.type = reserved.get(t.value, "ID")
94      return t
```

```
95  # comments
96  def t_comment_singleline(t):
97      r'//.*'
98      pass
99  def t_comment_multiline(t):
100     r'/\*([^*]|\*+[^*/])*\*+/'
101     pass
102 # ignore
103 t_ignore = " \t\r"
104 def t_newline(t):
105     r'\n+'
106     t.lexer.lineno += len(t.value)
107 def t_error(t):
108     print(f"Illegal char '{t.value[0]}' at line {t.lineno}")
109     t.lexer.skip(1)
110 lexer = lex.lex()
```

Listing 1: lexer.py

A.2   parser.py

```
1   import ply.yacc as yacc
2   from lexer import tokens
3   from ast_nodes import *
4   # Precedence rules
5   precedence = (
6       ('left', 'LOR'),
7       ('left', 'LAND'),
8       ('left', 'EQ', 'NEQ'),
9       ('left', 'LT', 'LE', 'GT', 'GE'),
10      ('left', 'PLUS', 'MINUS'),
11      ('left', 'MULT', 'DIV', 'MOD'),
12      ('right', 'NOT'),
13      ('right', 'UMINUS'),
14  )
15  # Program structure
16  def p_program(p):
17      '''program : external_list'''
18      p[0] = ProgramNode(p[1])
19  def p_external_list(p):
20      external_list : external
21                    | external_list external
22      if len(p) == 2:
23          p[0] = [p[1]]
24      else:
25          p[0] = p[1] + [p[2]]
26  def p_external(p):
27      external : includes
28               | using_namespace
29               | function_def
30               | main_function
```

```python
31       p[0] = p[1]

32

33  # includes / using namespace
34  def p_includes(p):
35      includes : INCLUDE
36      p[0] = None
37  def p_using_namespace(p):
38      using_namespace : USING NAMESPACE STD SEMICOLON
39                                | empty
40      p[0] = None
41  # Function definitions
42  def p_function_def(p):
43      function_def : type ID LPAREN param_list RPAREN block
44      p[0] = FunctionNode(p[1], p[2], p[4], p[6])
45  def p_main_function(p):
46      main_function : INT MAIN LPAREN RPAREN block
47      p[0] = FunctionNode('INT', 'main', [], p[5])
48  def p_param_list(p):
49      param_list : empty
50                      | params
51      p[0] = [] if p[1] is None else p[1]
52  def p_params(p):
53      params : param
54              | params COMMA param
55      p[0] = [p[1]] if len(p) == 2 else p[1] + [p[3]]
56  def p_param(p):
57      param : type ID
58      p[0] = ParamNode(p[1], p[2])
59  # Types
60  def p_type(p):
61      type : INT
62              | FLOAT
63              | DOUBLE
64              | CHAR
65              | BOOL
66              | STRING
67              | VOID
68      p[0] = p.slice[1].type
69  # Block / Statement list
70  def p_block(p):
71      'block : LBRACE statement_list RBRACE'
72      p[0] = BlockNode(p[2])
73  def p_statement_list(p):
74      statement_list : empty
75                          | statement_list statement
76      p[0] = [] if p[1] is None else p[1] + [p[2]]

77

78  def p_statement(p):
79      statement : declaration SEMICOLON
80                  | assignment SEMICOLON
81                  | print_stmt SEMICOLON
```

14

```python
                        | 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):
```

16

```
179        expression : NUMBER
180        p[0] = NumNode(p[1])
181 def p_expression_bool(p):
182        expression : TRUE
183                   | FALSE
184        p[0] = BoolNode(p.slice[1].type == 'TRUE')
185 def p_expression_string(p):
186        expression : STRING_LITERAL
187        p[0] = StringNode(p[1])
188 def p_expression_var(p):
189        expression : ID
190        p[0] = VarNode(p[1])
191 # function call
192 def p_expression_call(p):
193        expression : ID LPAREN arg_list RPAREN
194        p[0] = CallNode(p[1], p[3])
195 def p_arg_list_empty(p):
196        arg_list : empty
197        p[0] = []
198 def p_arg_list_single(p):
199        arg_list : expression
200        p[0] = [p[1]]
201 def p_arg_list_multi(p):
202        arg_list : arg_list COMMA expression
203        p[0] = p[1] + [p[3]]
204 def p_expression_paren(p):
205        expression : LPAREN expression RPAREN
206        p[0] = p[2]
207 # Empty
208 def p_empty(p):
209        empty :
210        pass
211
212 # Error
213 def p_error(p):
214        if p:
215            print(f"Syntax error at '{p.value}' (line {p.lineno})")
216        else:
217            print("Syntax error at EOF")
218 parser = yacc.yacc()
```

Listing 2: Parser Definition (`parser.py`)

A.3   AST nodes.py

```
1 class ProgramNode:
2     def __init__(self, declarations):
3         self.declarations = declarations
4     def __repr__(self):
5         return f"Program({self.declarations})"
6
```

```python
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):
```

```python
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):
```

```
106          return f"Unary({self.op}␣{self.expr})"
107
108  class NumNode:
109      def __init__(self, value):
110          self.value = value
111      def __repr__(self):
112          return f"Num({self.value})"
113
114  class BoolNode:
115      def __init__(self, value):
116          self.value = bool(value)
117      def __repr__(self):
118          return f"Bool({self.value})"
119
120  class VarNode:
121      def __init__(self, name):
122          self.name = name
123      def __repr__(self):
124          return f"Var({self.name})"
125
126  class StringNode:
127      def __init__(self, value):
128          self.value = value
129      def __repr__(self):
130          return f"String({self.value!r})"
```

Listing 3: astnodes.py

A.4   codegen.py

```
1   from ast_nodes import *
2   class CodeGenerator:
3       def __init__(self):
4           self.indent_level = 0
5           self.symtab = {}
6           self.functions = []
7
8       def indent(self):
9           return "␣␣␣␣" * self.indent_level
10
11      # Top Level Dispatcher
12      def generate(self, node):
13          if node is None:
14              return ""
15
16          # ----------------- PROGRAM ------------------
17          if isinstance(node, ProgramNode):
18              parts = []
19              for d in node.declarations:
20                  if d:
21                      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
```

22

```
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("-"):
```

```
217            step = "-" + step
218
219        # ----------- Build final Python loop -----------
220        loop = f"for {var} in range({start}, {stop}, {step}):\n"
221        loop += self.generate(body)
222        return loop
223
224    # fallback -> while loop
225    def generate_fallback_for(self, node):
226        init_code = self.generate(node.init)
227        cond_code = self.generate(node.cond)
228        incr_code = self.generate(node.incr)
229        body_code = self.generate(node.body)
230
231        result = init_code + "\n"
232        result += f"while {cond_code}:\n"
233
234        self.indent_level += 1
235        for line in body_code.splitlines():
236            result += self.indent() + line + "\n"
237        result += self.indent() + incr_code + "\n"
238        self.indent_level -= 1
239
240        return result
241
242
243 # Manual tester
244 if __name__ == "__main__":
245     from parser import parser
246     with open(input("File location: "), "r", encoding="utf-8") as
           f:
247         data = f.read()
248
249     ast = parser.parse(data)
250     print("AST:", ast)
251
252     gen = CodeGenerator()
253     py_code = gen.generate(ast)
254     print("\nGenerated Python code:\n")
255     print(py_code)
```

Listing 4: codegen.py

A.5   main.py

```
1 import os
2 import re
3 from parser import parser
4 from codegen import CodeGenerator
5
6 # Natural sort key
```

```python
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

26

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++)

```cpp
#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)

```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++)

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

int main() {
    int x = 15;

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

```
10              cout << "x␣is␣out␣of␣range" << endl;
11          }
12    }
```

Output (Python)

```
1    def main():
2        x = 15
3        if ((x > 10) and (x < 20)):
4            print("x␣is␣between␣10␣and␣20")
5        else:
6            print("x␣is␣out␣of␣range")
7
8    if __name__ == "__main__":
9        main()
```

Test 3: User Input Handling and Arithmetic

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

Input (C++)

```
1    #include <iostream>
2    using namespace std;
3
4    int main() {
5        int x = 0;
6        int y = 0;
7        cout << "Enter␣x:";
8        cin >> x;
9        cout << "Enter␣y:";
10        cin >> y;
11        int sum = x + y;
12        int mul = x * y;
13        cout << "sum=" << sum << "␣mul=" << mul << endl;
14    }
```

Output (Python)

```
1    def main():
2        x = 0
3        y = 0
4        print("Enter␣x:", end='')
5        x = int(input())
6        print("Enter␣y:", end='')
7        y = int(input())
8        sum = (x + y)
9        mul = (x * y)
```

```
10        print("sum=", sum, "␣mul=", mul)
11
12 if __name__ == "__main__":
13        main()
```

Test 4: Nested For-Loops

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

Input (C++)

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      for (int i = 1; i <= 3; i++) {
6          for (int j = 1; j <= 2; j++) {
7              cout << "i=" << i << "␣j=" << j << endl;
8          }
9      }
10 }
```

Output (Python)

```
1  def main():
2      for i in range(1, (3 + 1), 1):
3          for j in range(1, (2 + 1), 1):
4              print("i=", i, "␣j=", j)
5
6  if __name__ == "__main__":
7      main()
```

Test 5: User-Defined Function and Function Call

Category: Functions, Return Values, Function Invocation

Input (C++)

```
1  #include <iostream>
2  using namespace std;
3
4  int add(int a, int b) {
5      return a + b;
6  }
7
8  int main() {
9      int x = 10;
10     int y = 20;
```

```
11      int s = add(x, y);
12      cout << "Sum=" << s << endl;
13  }
```

Output (Python)

```
1   def add(a, b):
2       return (a + b)
3
4   def main():
5       x = 10
6       y = 20
7       s = add(x, y)
8       print("Sum=", s)
9
10  if __name__ == "__main__":
11      main()
```

Test 6: Boolean Function and Logical Evaluation

Category: Boolean Logic, Function Return Values, Complex Conditionals

Input (C++)

```
1   #include <iostream>
2   using namespace std;
3
4   bool check(int a, int b) {
5       if (a > 0 && b > 0) {
6           return true;
7       } else {
8           return false;
9       }
10  }
11
12  int main() {
13      int x = 3;
14      int y = -1;
15      if (check(x, y) || x > 1) {
16          cout << "OK" << endl;
17      } else {
18          cout << "NOT␣OK" << endl;
19      }
20  }
```

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)):
```

```
3            return True
4        else:
5            return False
6
7    def main():
8        x = 3
9        y = -1
10       if (check(x, y) or (x > 1)):
11           print("OK")
12       else:
13           print("NOT␣OK")
14
15   if __name__ == "__main__":
16       main()
```

Test 8: Multi-Function Program with Loops and Processing

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

Input (C++)

```
1    #include <iostream>
2    using namespace std;
3
4    int factorial(int n) {
5        int f = 1;
6        for (int i = 1; i < n + 1; i++) {
7            f = f + (f * (i - 1));
8        }
9        return f;
10   }
11
12   int fibonacci(int n) {
13       int a = 0;
14       int b = 1;
15       int temp = 0;
16       for (int i = 1; i < n + 1; i++) {
17           temp = a + b;
18           a = b;
19           b = temp;
20       }
21       return a;
22   }
23
24   int sumSquares(int n) {
25       int s = 0;
26       for (int i = 1; i < n + 1; i++) {
27           s = s + (i * i);
28       }
29       return s;
```

```
30  }
31
32  int compute(int x, int y, int z) {
33      int fact = factorial(x);
34      int fib = fibonacci(y);
35      int sq  = sumSquares(z);
36      int result = fact + fib + sq ;
37      return result;
38  }
39
40  int main() {
41      int x=0;
42      int y=0;
43      int z=0;
44      cout << "Enter␣x:␣";
45      cin >> x;
46      cout << "Enter␣y:␣";
47      cin >> y;
48      cout << "Enter␣z:␣";
49      cin >> z;
50      cout << "Processing..." << endl;
51      int output = compute(x, y, z);
52      cout << "Final␣Output␣=␣" << output << endl;
53  }
```

Output (Python)

```python
1   def factorial(n):
2       f = 1
3       for i in range(1, (n + 1), 1):
4           f = (f + (f * (i - 1)))
5       return f
6
7   def fibonacci(n):
8       a = 0
9       b = 1
10      temp = 0
11      for i in range(1, (n + 1), 1):
12          temp = (a + b)
13          a = b
14          b = temp
15      return a
16
17  def sumSquares(n):
18      s = 0
19      for i in range(1, (n + 1), 1):
20          s = (s + (i * i))
21      return s
22
23  def compute(x, y, z):
```

```python
    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()
```