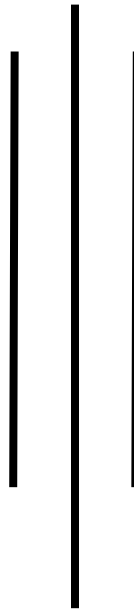




School of Engineering & Technology

Asian Institute of Technology

AT70.07 - Programming Languages and Compilers



Date: 11 May 2025

Report: PLC Final Project

Submitted To

Prof. Phan Minh Dung

Mr. Akaradet Sinsamersuk

Submitted By:

Aman Bhardwaj - st125713

Huy Nguyen - st124724

Aung Htet Lwin - st125773

Table of Contents

1. Introduction	4
2. Detailed Description of the Language Grammar	5
2.1. Lexical Grammar	5
2.2. Syntactic Grammar	8
3. Language Features in Detail	13
3.1. Types: int, float, boolean, and string	13
3.2. Dynamic Typing: Runtime type checking and error handling	13
3.3. Arithmetic Expressions: +, -, , / with correct precedence	14
3.4. Boolean Expressions: ==, != between arithmetic expressions	14
3.5. String Operations: Concatenation using + operator	14
3.6. Control Flow	15
3.6.1. if... then... else... end	15
3.6.2. while... do... end	15
3.7. Functions	16
3.7.1. Definitional parameter passing (no recursion)	16
3.7.2. User-defined functions with def... end	16
3.7.3. Built-in Function: print(...)	16
4. Compiler Architecture and Implementation	17
4.1. Lexical Analysis	17
4.2. Syntactic Analysis	18
4.3. Abstract Syntax Tree (AST)	18
4.4. Semantic Analysis and Runtime Environment	19
5. Graphical User Interface (GUI)	21
6. Test Cases and Execution Results	22
6.1 Arithmetic and String Operations	22
6.2 If-Else Control Flow with Booleans	23
6.3 While Loop with Accumulation	24
6.4 Function Definition and Call	25
6.5 Type Operations and Concatenation	26

6.6 Boolean Logic and Nested Conditionals.....	27
6.7 While Loop and Arithmetic	27
6.8 Function with Parameters	28
6.9 Float Calculation and Equality Check	29
7. Project Setup and Execution Guide.....	29
8. Team Members and Responsibilities	30
9. Conclusion and Future Work.....	31

1. Introduction

This document presents a comprehensive analysis of the PLC (Programming Language and Compiler) project, undertaken as the final submission for the Programming Languages and Compilers course at AIT. This endeavor involved the design and implementation of a custom programming language, accompanied by a Python-based compiler. The primary objective was to create a functional compiler for a small yet expressive language incorporating essential programming paradigms and features. The resulting language supports fundamental data types such as integers, floating-point numbers, booleans, and strings. It also features dynamic typing, allowing for runtime type checking and error handling. The language enables the construction of arithmetic and boolean expressions, including standard operators with appropriate precedence. Furthermore, it provides string manipulation capabilities through concatenation. Control flow mechanisms, specifically if-then-else and while-do statements, are integral to the language's structure. The design includes support for user-defined functions with definitional parameter passing, alongside a built-in print function for output. To enhance user interaction, a graphical user interface (GUI) was developed using PyQt6, providing an integrated development environment with code input, an Abstract Syntax Tree (AST) view, and an output display. The compiler's architecture is based on Python and leverages tools such as SLY (Sly Lexing and Parsing) for the lexical and syntactic analysis phases. This project serves as a practical demonstration of the principles and concepts learned throughout the Programming Languages and Compilers course, showcasing the application of theoretical knowledge in a tangible implementation.

2. Detailed Description of the Language Grammar

The grammar of the PLC language defines the rules that govern the structure and syntax of valid programs written in this language. It can be broadly categorized into two main components: the lexical grammar, which describes the individual symbols or tokens that make up the language, and the syntactic grammar, which specifies how these tokens can be combined to form meaningful statements and programs.

2.1. Lexical Grammar

The lexical grammar of the PLC language is responsible for identifying and classifying the basic units of the language, known as tokens [lexica.py]. The MyLexer class, implemented using the SLY library, performs this task by defining a set of regular expressions that match different patterns in the source code. The tokens recognized by the lexer include fundamental elements such as NUMBER (representing both integer and floating-point literals), STRING (representing text enclosed in double quotes), and NAME (representing identifiers for variables and functions). Additionally, the lexer identifies various operators, including arithmetic operators (PLUS, MINUS, TIMES, DIVIDE, MOD), comparison operators (EQEQ, NOTEQ, LT, LE, GT, GE), and the assignment operator (EQUAL). Punctuation marks like parentheses (LPAREN, RPAREN) and the comma (,) are also recognized as tokens. Furthermore, the language incorporates several keywords that have specific meanings, such as IF, THEN, ELSE, END, WHILE, DO, DEF, and PRINT. Boolean literals TRUE and FALSE are also identified as distinct tokens, with their corresponding boolean values being assigned during the lexical analysis phase [lexica.py]. Characters like spaces and tabs are explicitly ignored, as defined by the ignored attribute. The lexer also handles single-line comments, which start with // and continue to the end of the line, by effectively discarding them. Newline characters are counted to maintain line numbers for error reporting but are otherwise ignored. In cases where the lexer encounters characters that do not match any defined token pattern, it raises a LexerError, providing information about the illegal character and its location [lexica.py]. The use of regular expressions for defining token patterns is a standard and efficient method in lexical analysis, allowing for precise and unambiguous identification of the language's basic building blocks. Handling keywords directly within the rule for identifiers simplifies the overall grammar definition by avoiding the need for separate rules for each keyword. The lexer's ability to differentiate between integer and floating-point numbers, recognize string literals by removing the surrounding quotes, and handle boolean literals and comments demonstrates its capability to correctly interpret various fundamental elements of the

language. The inclusion of a custom error handling mechanism ensures that the compiler can provide feedback to the user when encountering invalid lexical elements.

```

compiler-starter-project > components > lexica.py > MyLexer
1  from sly import Lexer
2
3  class LexerError(Exception):
4      pass
5
6  class MyLexer(Lexer):
7      tokens = [
8          'NUMBER', 'STRING', 'NAME',
9          'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'MOD',
10         'EQUAL', 'EQEQ', 'NOTEQ',
11         'LT', 'LE', 'GT', 'GE',
12         'LPAREN', 'RPAREN', 'COMMA',
13         'IF', 'THEN', 'ELSE', 'END', 'WHILE', 'DO',
14         'DEF', 'PRINT', 'TRUE', 'FALSE']
15
16
17     ignore = ' \t' # ignore spaces and tabs
18
19     # Operators and punctuation
20     PLUS = r'\+'
21     MINUS = r'\-'
22     TIMES = r'\*'
23     DIVIDE = r'\/'
24     EQEQ = r'=='
25     NOTEQ = r'!='
26     EQUAL = r'='
27     LPAREN = r'\('
28     RPAREN = r'\)'
29     COMMA = r','
30     MOD = r'%'
31     LE = r'<='
32     GE = r'>='
33     LT = r'<'
34     GT = r'>'
35
36
37     # String literals (double quotes)
38     @_(r'\("[^"]*"')
39     def STRING(self, token):
40         token.value = token.value[1:-1] # remove surrounding quotes
41         return token
42
43     # Number literal (integer or float)
44     @_(r'\d+\.\d+')
45     def NUMBER(self, token):
46         token.value = float(token.value)
47         return token
48
49     @_(r'\d+(\.\d+)?')
50     def NUMBER(self, t):
51         t.value = float(t.value) if '.' in t.value else int(t.value)
52         return t

```

```

compiler-starter-project > components > lexica.py > MyLexer
6  class MyLexer(Lexer):
7
8      # Identifiers and keywords
9      @_(r'[a-zA-Z_][a-zA-Z0-9_]*')
10     def NAME(self, token):
11         # Keywords
12         if token.value == 'if':
13             token.type = 'IF'
14         elif token.value == 'then':
15             token.type = 'THEN'
16         elif token.value == 'else':
17             token.type = 'ELSE'
18         elif token.value == 'end':
19             token.type = 'END'
20         elif token.value == 'while':
21             token.type = 'WHILE'
22         elif token.value == 'do':
23             token.type = 'DO'
24         elif token.value == 'def':
25             token.type = 'DEF'
26         elif token.value == 'print':
27             token.type = 'PRINT'
28         elif token.value == 'true':
29             token.type = 'TRUE'
30             token.value = True
31         elif token.value == 'false':
32             token.type = 'FALSE'
33             token.value = False
34         else:
35             token.type = 'NAME'
36         return token
37
38     # Single-line comments (// to end of line)
39     @_(r'//.*')
40     def COMMENT(self, t):
41         pass
42
43     # Newlines (for line counting, but otherwise ignored)
44     @_(r'\n+')
45     def ignore_newline(self, t):
46         self.lineno += t.value.count('\n')
47
48     # Illegal characters
49     def error(self, token):
50         raise LexerError(f"Illegal character {token.value[0]!r} at line {self.lineno}")

```

2.2. Syntactic Grammar

The syntactic grammar of the PLC language specifies how the tokens identified by the lexer can be combined to form valid program structures, such as statements and expressions [parsers.py]. The MyParser class, built using the SLY library, defines these grammatical rules. A crucial aspect of the

syntactic grammar is the definition of operator precedence, which dictates the order in which operations are performed in expressions. In the PLC language, multiplication (TIMES), division (DIVIDE), and modulo (MOD) have higher precedence than addition (PLUS) and subtraction (MINUS). Comparison operators (EQEQ, NOTEQ, LT, LE, GT, GE) have a lower precedence than arithmetic operators [parsers.py]. The grammar rules are defined using a set of methods within the MyParser class, each decorated with @_. The program rule serves as the entry point of the parser, defining a program as a sequence of statements. The statements rule is defined recursively, allowing for a program to consist of one or more statements. The stmt rule defines the various types of statements supported by the language, including variable assignment (NAME EQUAL expr), if statements (both with and without an else clause), while loops, function definitions (DEF NAME LPAREN parameters RPAREN THEN statements END), print statements (PRINT LPAREN expr RPAREN), and function calls (NAME LPAREN arguments RPAREN) [parsers.py]. The expr rule defines different types of expressions, encompassing binary operations (combining two expressions with an operator), comparisons (comparing two expressions), parenthesized expressions (used to override default precedence), numeric literals, string literals, boolean literals (TRUE, FALSE), and variable names (NAME). The grammar also includes rules for defining function parameters (parameters) and passing arguments during function calls (arguments). The parser, upon successfully analyzing the token stream, constructs an Abstract Syntax Tree (AST) by instantiating classes defined in the components.ast.statement module. This AST serves as a hierarchical representation of the program's structure, making it easier for subsequent stages of the compiler to process the code. The defined operator precedence ensures that complex expressions are parsed and evaluated according to standard mathematical rules. The inclusion of common control flow structures like if-then-else and while-do enables the creation of programs with conditional execution and iterative behavior. The support for user-defined functions, along with the built-in print function, enhances the language's expressiveness and practicality. The generation of an AST is a fundamental step in the compilation process, providing a structured and abstract representation of the source code that facilitates further analysis and execution.

```

15     @_('statements')
16     def program(self, p):
17         return p.statements
18
19     @_('stmt')
20     def statements(self, p):
21         return [p.stmt]
22
23     @_('stmt statements')
24     def statements(self, p):
25         return [p.stmt] + p.statements
26
27     @_('NAME EQUAL expr')
28     def stmt(self, p):
29         return ast_module.StatementAssign(p.NAME, p.expr)
30
31     @_('IF expr THEN statements END')
32     def stmt(self, p):
33         return ast_module.StatementIf(p.expr, ast_module.StatementBlock(p.statements))
34
35     @_('IF expr THEN statements ELSE statements END')
36     def stmt(self, p):
37         return ast_module.StatementIf(p.expr, ast_module.StatementBlock(p.statements0), ast_module.StatementBlock(p.statements1))
38
39     @_('WHILE expr DO statements END')
40     def stmt(self, p):
41         return ast_module.StatementWhile(p.expr, ast_module.StatementBlock(p.statements))
42
43     @_('DEF NAME LPAREN parameters RPAREN THEN statements END')
44     def stmt(self, p):
45         return ast_module.StatementFunctionDef(p.NAME, p.parameters, ast_module.StatementBlock(p.statements))
46
47     @_('PRINT LPAREN expr RPAREN')
48     def stmt(self, p):
49         return ast_module.StatementPrint(p.expr)
50
51     @_('NAME LPAREN arguments RPAREN')
52     def stmt(self, p):
53         return ast_module.StatementFunctionCall(p.NAME, p.arguments)
54

```

```

5  class MyParser(Parser):
54
55      @_('expr EQEQ expr')
56      def expr(self, p):
57          return ast_module.ExpressionCompare('==', p.expr0, p.expr1)
58
59      @_('expr NOTEQ expr')
60      def expr(self, p):
61          return ast_module.ExpressionCompare('!=', p.expr0, p.expr1)
62
63      @_('expr LT expr')
64      def expr(self, p):
65          return ast_module.ExpressionBinary(ast_module.Operations.LT, p.expr0, p.expr1)
66
67      @_('expr LE expr')
68      def expr(self, p):
69          return ast_module.ExpressionBinary(ast_module.Operations.LE, p.expr0, p.expr1)
70
71      @_('expr GT expr')
72      def expr(self, p):
73          return ast_module.ExpressionBinary(ast_module.Operations.GT, p.expr0, p.expr1)
74
75      @_('expr GE expr')
76      def expr(self, p):
77          return ast_module.ExpressionBinary(ast_module.Operations.GE, p.expr0, p.expr1)
78
79      @_('expr PLUS expr')
80      def expr(self, p):
81          return ast_module.ExpressionBinary(ast_module.Operations.PLUS, p.expr0, p.expr1)
82
83      @_('expr MINUS expr')
84      def expr(self, p):
85          return ast_module.ExpressionBinary(ast_module.Operations.MINUS, p.expr0, p.expr1)
86
87      @_('expr TIMES expr')
88      def expr(self, p):
89          return ast_module.ExpressionBinary(ast_module.Operations.TIMES, p.expr0, p.expr1)
90
91      @_('expr DIVIDE expr')
92      def expr(self, p):
93          return ast_module.ExpressionBinary(ast_module.Operations.DIVIDE, p.expr0, p.expr1)
94
95      @_('LPAREN expr RPAREN')
96      def expr(self, p):
97          return p.expr
98
99      @_('NUMBER')
100      def expr(self, p):
101          return ast_module.ExpressionNumber(p.NUMBER)
102
103      @_('STRING')
104      def expr(self, p):
105          return ast_module.ExpressionString(p.STRING)
106

```

```

5  class MyParser(Parser):
110
111     @_('FALSE')
112     def expr(self, p):
113         return ast_module.ExpressionBoolean(p.FALSE)
114
115     @_('NAME')
116     def expr(self, p):
117         return ast_module.ExpressionVariable(p.NAME)
118
119     @_('NAME')
120     def parameters(self, p):
121         return [p.NAME]
122
123     @_('NAME COMMA parameters')
124     def parameters(self, p):
125         return [p.NAME] + p.parameters
126
127     @_('')
128     def parameters(self, p):
129         return []
130
131     @_('expr')
132     def arguments(self, p):
133         return [p.expr]
134
135     @_('expr COMMA arguments')
136     def arguments(self, p):
137         return [p.expr] + p.arguments
138
139     @_('expr MOD expr')
140     def expr(self, p):
141         return ast_module.ExpressionBinary(ast_module.Operations.MOD, p.expr0, p.expr1)
142
143     @_('')
144     def arguments(self, p):
145         return []
146

```

3. Language Features in Detail

The PLC programming language incorporates a set of features designed to provide a foundation for writing simple yet functional programs. These features include support for basic data types, dynamic typing, various types of expressions, control flow mechanisms, and functions.

3.1. Types: int, float, boolean, and string

The PLC language supports four fundamental data types: integer (int), floating-point number (float), boolean (boolean), and string (string). Integer and floating-point numbers are recognized by the lexer based on their numerical format [lexica.py]. Boolean values are represented by the keywords true and false, which are also recognized as tokens and assigned their respective boolean values during lexical analysis [lexica.py]. String literals are defined as sequences of characters enclosed within double quotes [lexica.py]. The ExpressionNumber, ExpressionString, and ExpressionBoolean classes in the ast/statement.py file are used to represent these different types of literals in the Abstract Syntax Tree. For instance, the code snippet `x = 10; y = 3.14; flag = true; message = "Hello"` demonstrates the declaration and assignment of values of each supported type to variables. The ability to handle these basic data types is essential for any programming language to manipulate and represent different kinds of information.

3.2. Dynamic Typing: Runtime type checking and error handling

The PLC language employs dynamic typing, which means that the type of variable is not explicitly declared by the programmer and is only determined at runtime based on the value assigned to it. This allows for greater flexibility in writing code, as the same variable can hold values of different types at different points in the program's execution. For example, a variable `a` could first hold an integer value and later be assigned a string value, as illustrated by the inferred snippet `a = 5; a = "hello"`. However, this flexibility necessitates runtime type checking to ensure that operations performed on variables are valid for their current type. The ExpressionBinary.run() method in ast/statement.py demonstrates this runtime type checking by using a try-except block to catch potential TypeError exceptions that might occur during binary operations. Furthermore, the behavior of the addition operator (+) is dynamically determined based on the types of its operands: if either operand is a string, the operation performs string concatenation; otherwise, it performs arithmetic addition. This dynamic behavior highlights a key characteristic of dynamic typing. The implementation of runtime type checking and error handling is

crucial for preventing unexpected behavior and providing informative feedback to the user when type mismatches occur during program execution.

3.3. Arithmetic Expressions: +, -, , / with correct precedence

The PLC language supports the standard arithmetic operations of addition (+), subtraction (-), multiplication (*), and division (/). The parser, as defined in `parsers.py`, enforces the correct order of operations (precedence) according to standard mathematical conventions. Multiplication and division have higher precedence than addition and subtraction, ensuring that in an expression like `5 + 3 * 2`, the multiplication is performed before the addition, resulting in 11. This precedence is defined in the `precedence` attribute of the `MyParser` class [`parsers.py`]. The `ExpressionBinary` class in `ast/statement.py`, along with the `Operations` enum (`PLUS`, `MINUS`, `TIMES`, `DIVIDE`), handles the execution of these arithmetic operations. The inferred code snippet `result = 5 + 3 * 2` illustrates how the precedence rules are applied. The correct implementation of arithmetic expressions with proper precedence is fundamental for allowing users to perform mathematical calculations in a predictable and accurate manner.

3.4. Boolean Expressions: ==, != between arithmetic expressions

The PLC language allows for the comparison of arithmetic expressions using the equality operator (`==`) and the inequality operator (`!=`). These comparisons result in boolean values (`true` or `false`). The grammar rules in `parsers.py` for `expr EQEQ expr` and `expr NOTEQ expr` define these boolean expressions [`parsers.py`]. The `ExpressionCompare` class in `ast/statement.py` handles the evaluation of these comparison operations. For example, the inferred snippet `is_equal = (5 + 2) == 7; is_different = 10 != 5` demonstrates the use of these operators. The ability to form boolean expressions is essential for implementing conditional logic in programs, allowing them to make decisions based on the evaluation of these expressions.

3.5. String Operations: Concatenation using + operator

The PLC language provides a basic string operation: concatenation. Strings can be joined together using the addition operator (+). As observed in the `ExpressionBinary.run()` method in `ast/statement.py`, when the `PLUS` operator is used with string operands (or if one operand is a string), the operation performs concatenation. For instance, the inferred snippet `greeting = "Hello, " + "world!"` would result in the variable `greeting` holding the string value `"Hello, world!"`. String concatenation is a common and

necessary operation in many programming tasks, enabling the manipulation and combination of textual data.

3.6. Control Flow

The PLC language includes two fundamental control flow statements: if-then-else for conditional execution and while-do for iterative execution.

3.6.1. if... then... else... end

The if-then-else statement in the PLC language allows for the conditional execution of blocks of code based on the truth value of a boolean expression. The syntax follows the structure `if <expression> then <statements> [else <statements>] end`, where the else part is optional. The grammar rules in `parsers.py` for `IF expr THEN statements END` and `IF expr THEN statements ELSE statements END` define the structure of this statement [`parsers.py`]. The `StatementIf` class in `ast/statement.py` implements the execution logic. The `run()` method of `StatementIf` first evaluates the condition. It checks if the result is a boolean value and raises a `TypeError` if it is not. If the condition evaluates to true, the statements in the then block are executed. If the condition is false and an else block is present, the statements within the else block are executed. The inferred code snippet `x = 10; if x > 5 then print("x is greater than 5") else print("x is not greater than 5") end` illustrates the use of this statement. Conditional execution is a cornerstone of programming, allowing programs to behave differently based on specific conditions.

3.6.2. while... do... end

The while-do statement in the PLC language provides a mechanism for repeated execution of a block of code as long as a specified boolean condition remains true. The syntax is `while <expression> do <statements> end`. The grammar rule in `parsers.py` for `WHILE expr DO statements END` defines this loop structure [`parsers.py`]. The `StatementWhile` class in `ast/statement.py` implements the execution. The `run()` method of `StatementWhile` evaluates the condition before each iteration. Similar to the if statement, it checks if the condition is a boolean value. If the condition is true, the statements within the do block are executed. This process continues until the condition becomes false, at which point the loop terminates. The inferred code snippet `counter = 0; while counter < 3 do print(counter); counter = counter + 1 end` demonstrates a simple while loop. Iterative execution is essential for automating repetitive tasks and performing operations on collections of data.

3.7. Functions

The PLC language supports user-defined functions, allowing for the modularization and reuse of code. It also includes a built-in function for output.

3.7.1. Definitional parameter passing (no recursion)

User-defined functions in the PLC language can accept parameters which are specified in the function definition. The language employs definitional parameter passing, which, based on the implementation in `memory.py`, appears to be pass-by-value. When a function is called, the values of the arguments are copied into the function's local scope as parameters. The project description explicitly states that recursion is not supported in this language.

3.7.2. User-defined functions with `def... end`

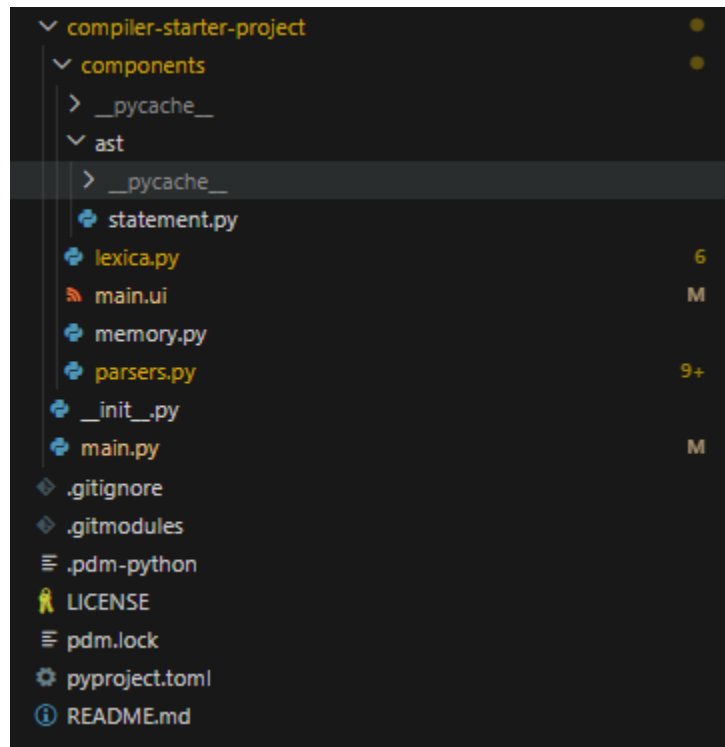
Functions are defined using the `def` keyword, followed by the function name, a list of parameters enclosed in parentheses, the keyword `then`, the body of the function (consisting of a block of statements), and finally the keyword `end`. The grammar rule `DEF NAME LPAREN parameters RPAREN THEN statements END` in `parsers.py` defines this structure [parsers.py]. The `StatementFunctionDef` class in `ast/statement.py` represents a function definition in the AST. The `run()` method of this class stores the function definition in the runtime memory for later use. The inferred code snippet `def greet(name) then print("Hello, " + name) end; greet("Alice")` shows a function definition and a subsequent call to that function. User-defined functions are crucial for organizing code into reusable blocks and improving the overall structure and readability of programs.

3.7.3. Built-in Function: `print(...)`

The PLC language includes a built-in function called `print`, which takes an expression as an argument and displays its value. The grammar rule `PRINT LPAREN expr RPAREN` in `parsers.py` handles the syntax of the `print` function call [parsers.py]. The `StatementPrint` class in `ast/statement.py` implements its functionality. The `run()` method of `StatementPrint` evaluates the given expression and then adds its string representation to the output buffer managed by the `MEMORY` object in `memory.py`. The inferred code snippets `print(10)` and `print("This is output")` demonstrate the use of the `print` function. A built-in `print` function is essential for allowing programs to produce output and communicate results to the user.

4. Compiler Architecture and Implementation

The PLC compiler follows a typical architecture for language processing, consisting of several stages: lexical analysis, syntactic analysis, and semantic analysis/runtime environment.



4.1. Lexical Analysis

The first stage of the compilation process is lexical analysis, where the source code is scanned character by character, and grouped into meaningful units called tokens [lexica.py]. This task is performed by the MyLexer class, which is implemented using the SLY (Sly Lexing and Parsing) library. SLY simplifies the creation of lexers by allowing the developer to define tokens and their corresponding patterns using regular expressions. The lexica.py file defines the MyLexer class, which specifies all the tokens that the PLC language recognizes, along with the regular expressions that match them. This includes keywords, operators, identifiers, literals (numbers, strings, booleans), and punctuation. The lexer also handles comments and whitespace, effectively ignoring them during the tokenization process. The use of SLY provides a robust and efficient way to perform lexical analysis, abstracting away the complexities of manual character-by-character scanning and pattern matching.

4.2. Syntactic Analysis

The next stage is syntactic analysis, or parsing, where the stream of tokens produced by the lexer is analyzed to determine if it conforms to the grammar of the PLC language [parsers.py]. If the token stream is syntactically correct, the parser constructs an Abstract Syntax Tree (AST), which represents the hierarchical structure of the program. The MyParser class, also implemented using the SLY library, performs this task. The parsers.py file defines the grammar rules of the PLC language using methods within the MyParser class, decorated with @_. These rules specify how tokens can be combined to form valid statements and expressions. The parser also defines operator precedence to handle expressions with multiple operators correctly. For example, it ensures that multiplication and division are performed before addition and subtraction. The output of the parsing stage is an AST, which is a tree-like data structure where each node represents a construct in the PLC language, such as an expression, a statement, or a function definition. Using SLY for parsing simplifies the process of defining the language's grammar and automatically generating the code needed to parse the token stream and build the AST. The precedence rules defined in the parser are crucial for ensuring that expressions are interpreted according to the intended order of operations.

4.3. Abstract Syntax Tree (AST)

The Abstract Syntax Tree (AST) is a crucial data structure in the PLC compiler. It provides a structured, hierarchical representation of the source code, abstracting away the concrete syntax and focusing on the essential components of the program [ast/statement.py]. The nodes of the AST represent different language constructs, such as expressions, statements, and declarations. The ast/statement.py file defines the classes that represent these AST nodes. There is an abstract base class ASTNode with an abstract run() method, which is intended to be implemented by its subclasses to define the execution logic for each node type. There are also base classes for Expression and Statement. Specific node types include ExpressionNumber, ExpressionString, ExpressionBoolean, ExpressionVariable, ExpressionBinary, ExpressionCompare, StatementAssign, StatementIf, StatementWhile, StatementFunctionDef, StatementFunctionCall, StatementPrint, and StatementBlock. The Operations enum within this file defines the binary operators used in the language.

Table 1: AST Node Types and Their Purpose

Node Type	Purpose
-----------	---------

ASTNode	Abstract base class for all AST nodes. Defines the run() method.
Expression	Abstract base class for all expression nodes.
ExpressionNumber	Represents a numeric literal (integer or float).
ExpressionString	Represents a string literal.
ExpressionBoolean	Represents a boolean literal (true or false).
ExpressionVariable	Represents a variable reference.
ExpressionBinary	Represents a binary operation (e.g., +, -, *, /, ==, !=, <, >, etc.).
ExpressionCompare	Represents equality (==) or inequality (!=) comparison.
Statement	Abstract base class for all statement nodes.
StatementAssign	Represents a variable assignment.
StatementIf	Represents an if statement.
StatementWhile	Represents a while loop.
StatementFunctionDef	Represents a function definition.
StatementFunctionCall	Represents a function call.
StatementPrint	Represents a print statement.
StatementBlock	Represents a block of statements (e.g., in if, while, or function).

This object-oriented design of the AST, with a clear hierarchy of classes, facilitates the implementation of the interpreter. The run() method in each concrete node class likely contains the logic for executing the corresponding language construct. The use of an enum for operators improves code organization and readability.

4.4. Semantic Analysis and Runtime Environment

Semantic analysis involves checking the meaning and consistency of the program after it has been parsed. In this project, much of the semantic checking, particularly type checking, appears to be performed at runtime due to the dynamic typing nature of the language. The runtime environment, which manages the execution of the program, is implemented in the memory.py file. The Memory class, implemented as a singleton, is responsible for managing variable scopes, storing function definitions, and handling program output. It uses a stack of dictionaries (self.scopes) to represent the different levels of scope that can exist during program execution, such as the global scope and the local scopes created by function calls. The self.functions dictionary stores user-defined functions, mapping their names to their corresponding StatementFunctionDef objects. The self.output list accumulates the output produced by

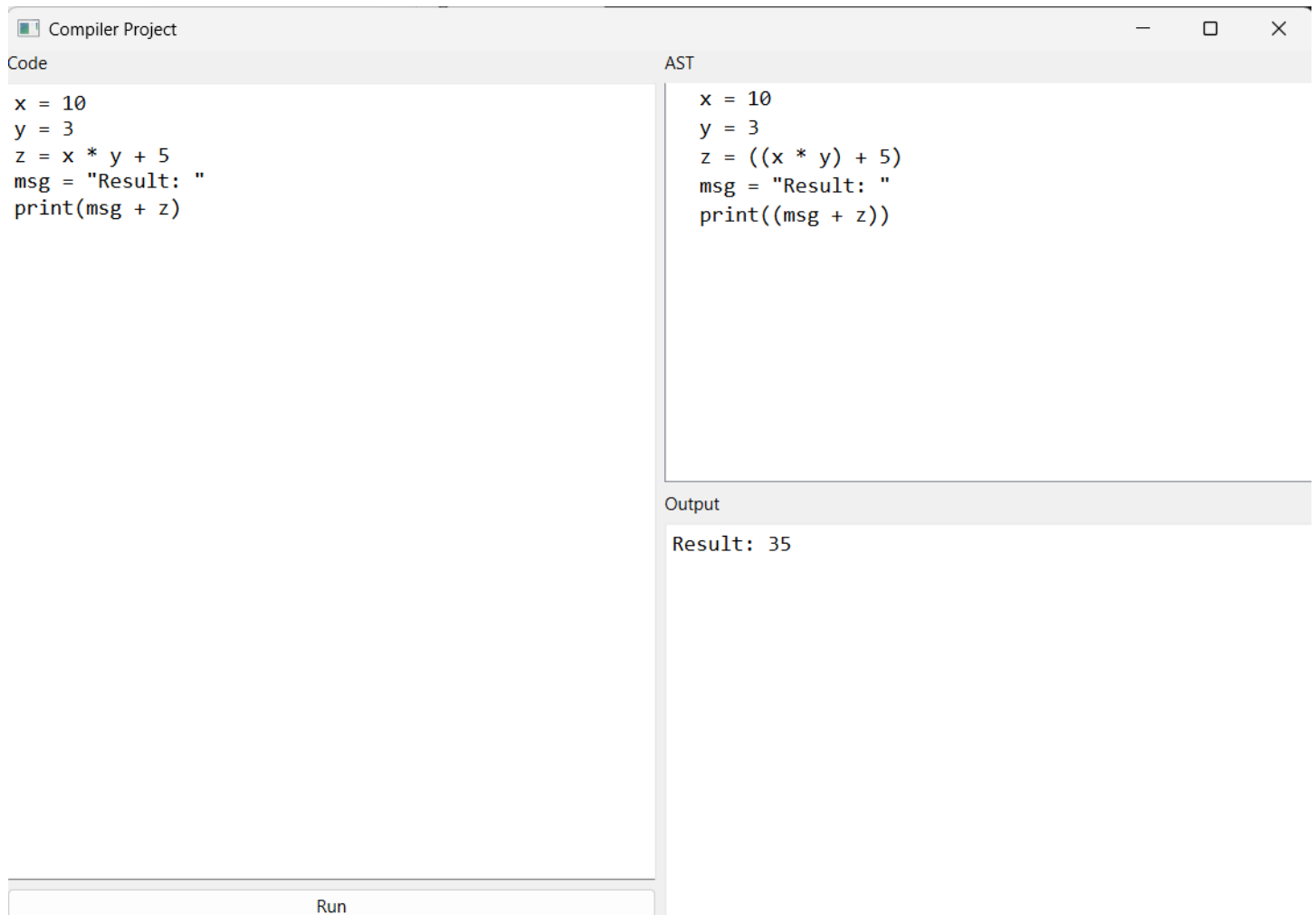
print statements. The Memory class provides methods for getting and setting variable values (`get()`, `set()`), adding to the output (`add_output()`), retrieving the entire output (`get_output()`), clearing the memory (`clear()`), defining functions (`define_function()`), and calling functions (`call_function()`). The use of a stack for scopes correctly handles the creation and destruction of local scopes during function calls. Storing the entire `StatementFunctionDef` object allows the interpreter to access the function's parameters and body when it is called. The `call_function()` method manages the creation of a new scope for the function call, binds the argument values to the parameters, and then executes the function's body. The global `MEMORY` instance provides a single point of access to the runtime environment for other components of the compiler.

5. Graphical User Interface (GUI)

The PLC project includes a graphical user interface (GUI) built using the PyQt6 library, providing an integrated development environment (IDE) for interacting with the language [main.py]. The main entry point of the application is the main.py file, which defines the MainWindow class, inheriting from QMainWindow. The GUI layout is defined in the components/main.ui file (the contents of which are not provided) and is loaded using `uic.loadUi()`. The MainWindow includes a `codeTextEdit` for the user to input PLC code, a `runButton` to trigger the compilation and execution process, an `outputTextEdit` to display the output of the program, and an `astTree` to visualize the generated Abstract Syntax Tree. When the `runButton` is clicked, the `run_code` method is executed. This method retrieves the code from the input text area, clears the runtime memory, and creates instances of the `MyLexer` and `MyParser`. It then attempts to parse the input code. The execution of the parsed AST follows a two-pass approach. In the first pass, it iterates through the top-level statements and executes only the function definitions. This likely allows functions to be called before their definition appears in the code. In the second pass, it executes the remaining statements. Error handling is implemented using a try-except block to catch any exceptions that occur during parsing or execution, displaying the error message in the output text area. If the execution is successful, the output accumulated in the `MEMORY` object is displayed in the `outputTextEdit`. Finally, the `astTree` is populated with a visual representation of the generated AST. The use of PyQt6 enables the creation of a user-friendly interface for the PLC language, making it easier to write, run, and debug code. The separation of the GUI layout from the application logic promotes better code organization and maintainability. The two-pass execution strategy for handling function definitions provides greater flexibility in how users can structure their code. The inclusion of an AST viewer is a valuable feature for understanding the internal workings of the compiler and for debugging the language implementation itself.

6. Test Cases and Execution Results

6.1 Arithmetic and String Operations



The screenshot shows a window titled "Compiler Project" with three main sections: "Code", "AST", and "Output".

Code:

```
x = 10
y = 3
z = x * y + 5
msg = "Result: "
print(msg + z)
```

AST:

```
x = 10
y = 3
z = ((x * y) + 5)
msg = "Result: "
print((msg + z))
```

Output:

```
Result: 35
```

At the bottom of the window, there is a "Run" button.

6.2 If-Else Control Flow with Booleans

Compiler Project

Code

```
score = 85
if score >= 90 then
  print("Excellent")
else
  print("Try harder")
end
```

Run

AST

```
score = 85
if (score >= 90) then
  print("Excellent")
else
  print("Try harder")
end
```

Output

```
Try harder
```

6.3 While Loop with Accumulation

Compiler Project

Code

```
i = 1
total = 0
while i <= 5 do
    total = total + i
    i = i + 1
end
print("Sum is: " + total)
```

AST

```
i = 1
total = 0
while (i <= 5) do
    total = (total + i)
    i = (i + 1)
end
print(("Sum is: " + total))
```

Output

```
Sum is: 15
```

Run

6.4 Function Definition and Call



The screenshot shows a window titled "Compiler Project" with three main sections: Code, AST, and Output. The Code section contains a function definition and a call. The AST section shows the corresponding abstract syntax tree. The Output section shows the result of running the code.

Code

```
def greet(name,ID) then
  print("Hello, " + name + ID)
end

greet("Huy", "st124724")
```

AST

```
def greet(name, ID) then
  print(("Hello, " + name) + ID))
end
greet("Huy", "st124724")
```

Output

```
Hello, Huyst124724
```

Run

6.5 Type Operations and Concatenation

Compiler Project

Code

```
name = "Huy"
age = 23
info = "Name: " + name + ", Age: " + age
print(info)
```

AST

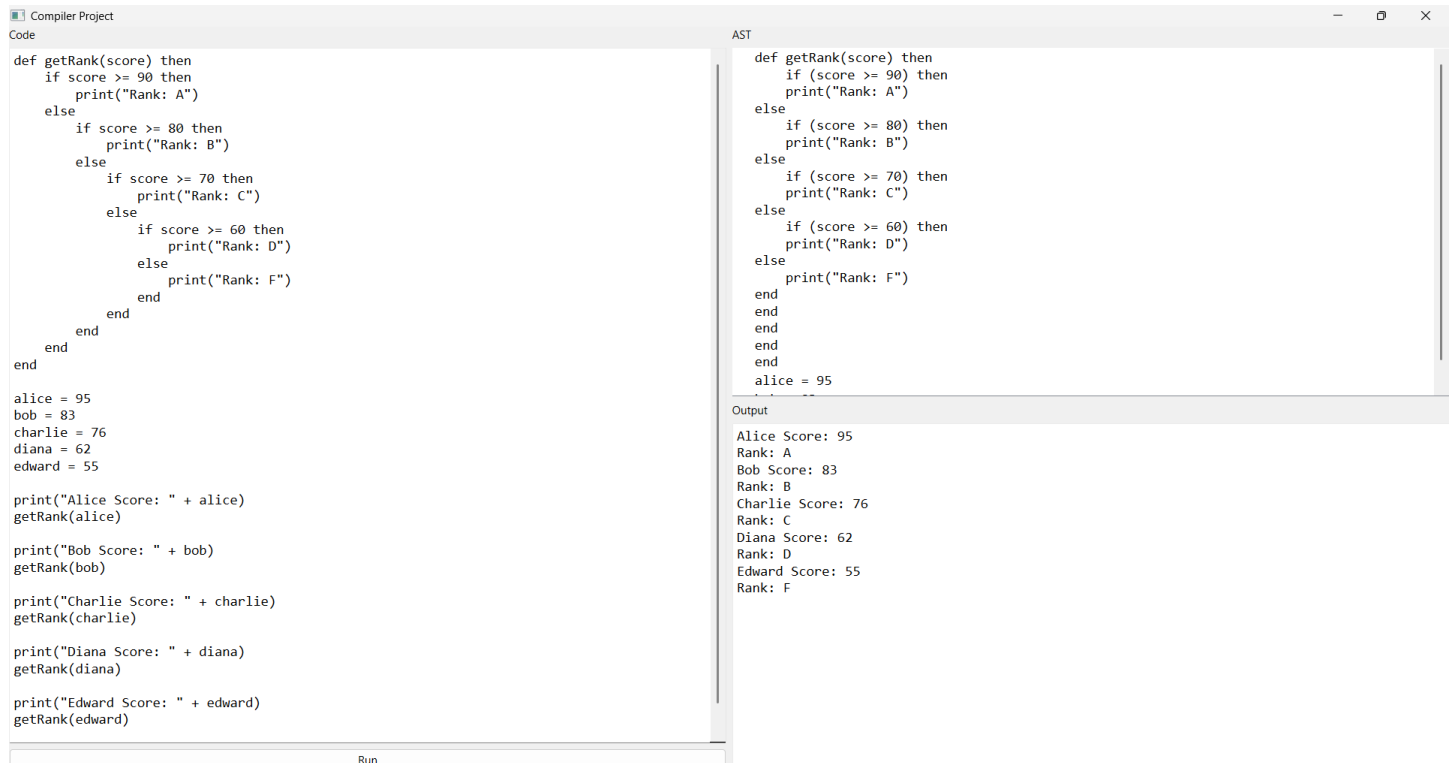
```
name = "Huy"
age = 23
info = (((("Name: " + name) + ", Age: ") + age)
print(info)
```

Output

```
Name: Huy, Age: 23
```

Run

6.6 Boolean Logic and Nested Conditionals



The screenshot shows a compiler project window with two panes: 'Code' and 'AST'. The 'Code' pane contains a function `getRank(score)` that uses nested `if-else` statements to assign ranks from A to F based on score ranges. Below the function, scores for Alice (95), Bob (83), Charlie (76), Diana (62), and Edward (55) are assigned, and each is passed to `getRank`. The 'AST' pane shows the abstract syntax tree for the same code. The 'Output' pane displays the results of the program execution.

```
def getRank(score) then
  if score >= 90 then
    print("Rank: A")
  else
    if score >= 80 then
      print("Rank: B")
    else
      if score >= 70 then
        print("Rank: C")
      else
        if score >= 60 then
          print("Rank: D")
        else
          print("Rank: F")
        end
      end
    end
  end
end

alice = 95
bob = 83
charlie = 76
diana = 62
edward = 55

print("Alice Score: " + alice)
getRank(alice)

print("Bob Score: " + bob)
getRank(bob)

print("Charlie Score: " + charlie)
getRank(charlie)

print("Diana Score: " + diana)
getRank(diana)

print("Edward Score: " + edward)
getRank(edward)
```

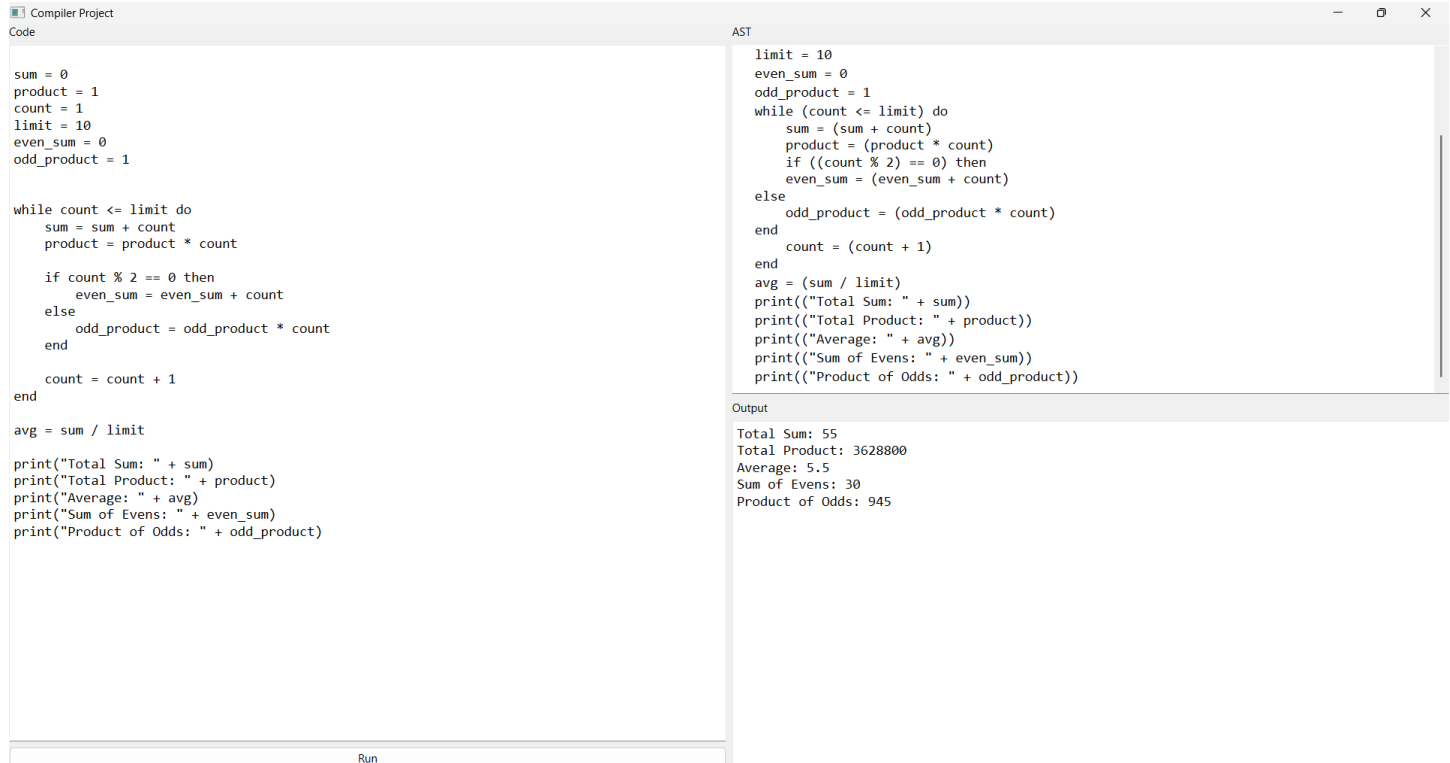
```
def getRank(score) then
  if (score >= 90) then
    print("Rank: A")
  else
    if (score >= 80) then
      print("Rank: B")
    else
      if (score >= 70) then
        print("Rank: C")
      else
        if (score >= 60) then
          print("Rank: D")
        else
          print("Rank: F")
        end
      end
    end
  end
end

alice = 95
```

Output

```
Alice Score: 95
Rank: A
Bob Score: 83
Rank: B
Charlie Score: 76
Rank: C
Diana Score: 62
Rank: D
Edward Score: 55
Rank: F
```

6.7 While Loop and Arithmetic



The screenshot shows a compiler project window with two panes: 'Code' and 'AST'. The 'Code' pane contains a `while` loop that calculates the sum, product, average, and sum/product of even and odd numbers from 1 to 10. The 'AST' pane shows the abstract syntax tree for the same code. The 'Output' pane displays the results of the program execution.

```
sum = 0
product = 1
count = 1
limit = 10
even_sum = 0
odd_product = 1

while count <= limit do
  sum = sum + count
  product = product * count

  if count % 2 == 0 then
    even_sum = even_sum + count
  else
    odd_product = odd_product * count
  end

  count = count + 1
end

avg = sum / limit

print("Total Sum: " + sum)
print("Total Product: " + product)
print("Average: " + avg)
print("Sum of Evens: " + even_sum)
print("Product of Odds: " + odd_product)
```

```
limit = 10
even_sum = 0
odd_product = 1
while (count <= limit) do
  sum = (sum + count)
  product = (product * count)
  if ((count % 2) == 0) then
    even_sum = (even_sum + count)
  else
    odd_product = (odd_product * count)
  end
  count = (count + 1)
end
avg = (sum / limit)
print(("Total Sum: " + sum))
print(("Total Product: " + product))
print(("Average: " + avg))
print(("Sum of Evens: " + even_sum))
print(("Product of Odds: " + odd_product))
```

Output

```
Total Sum: 55
Total Product: 3628800
Average: 5.5
Sum of Evens: 30
Product of Odds: 945
```

6.8 Function with Parameters

Compiler Project

Code

AST

Output

```
def calculateStats(a, b, c) then
  total = a + b + c
  average = total / 3
  maxScore = a
  if b > maxScore then
    maxScore = b
  end
  if c > maxScore then
    maxScore = c
  end
  print("Total: " + total)
  print("Average: " + average)
  print("Max Score: " + maxScore)
end

x = 12
y = 47
z = 24

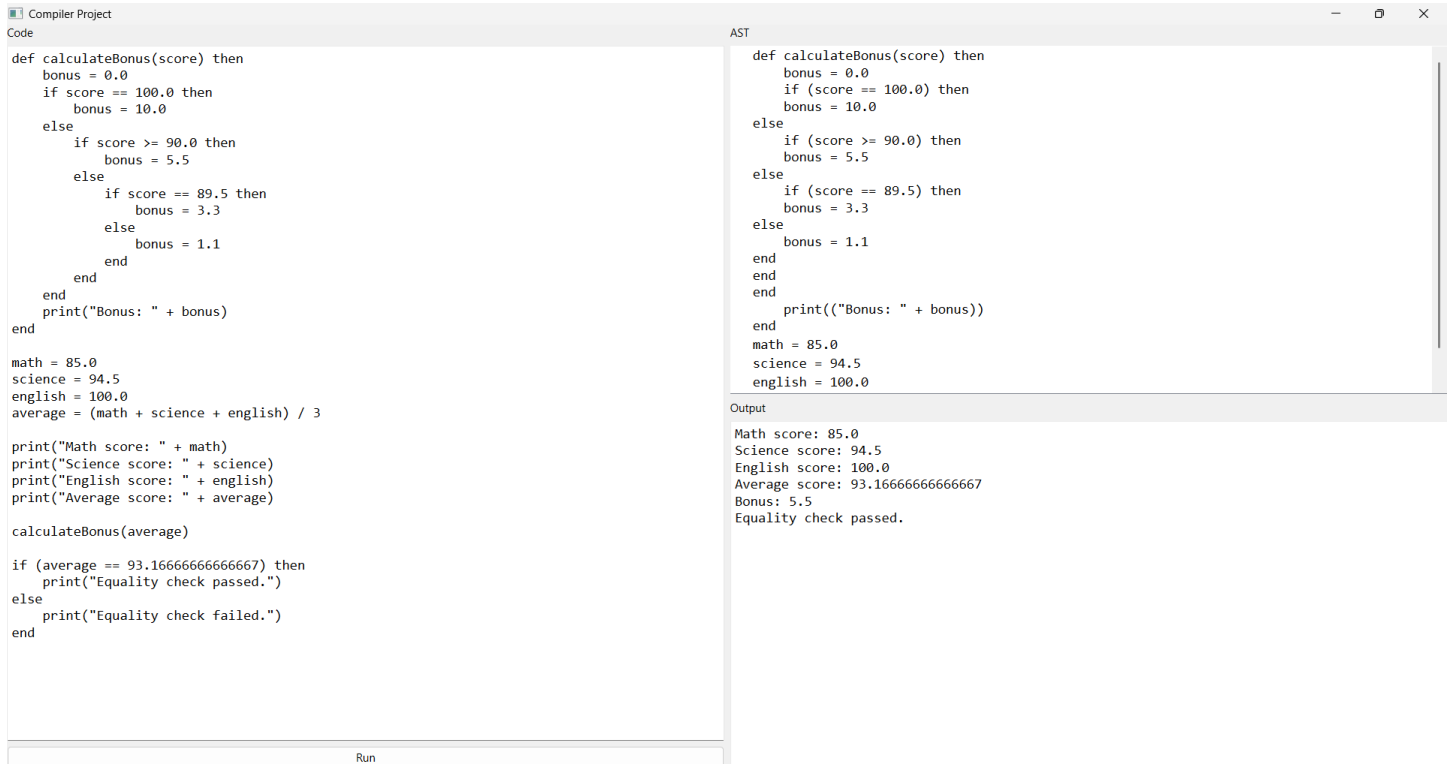
calculateStats(x, y, z)
```

```
def calculateStats(a, b, c) then
  total = ((a + b) + c)
  average = (total / 3)
  maxScore = a
  if (b > maxScore) then
    maxScore = b
  end
  if (c > maxScore) then
    maxScore = c
  end
  print(("Total: " + total))
  print(("Average: " + average))
  print(("Max Score: " + maxScore))
end
x = 12
```

```
Total: 83
Average: 27.666666666666668
Max Score: 47
```

Run

6.9 Float Calculation and Equality Check



The screenshot shows a Python IDE with two panes: 'Code' and 'AST'. The 'Code' pane contains the following Python code:

```
def calculateBonus(score) then
    bonus = 0.0
    if score == 100.0 then
        bonus = 10.0
    else
        if score >= 90.0 then
            bonus = 5.5
        else
            if score == 89.5 then
                bonus = 3.3
            else
                bonus = 1.1
            end
        end
    end
    end
    print("Bonus: " + bonus)
end

math = 85.0
science = 94.5
english = 100.0
average = (math + science + english) / 3

print("Math score: " + math)
print("Science score: " + science)
print("English score: " + english)
print("Average score: " + average)

calculateBonus(average)

if (average == 93.16666666666667) then
    print("Equality check passed.")
else
    print("Equality check failed.")
end
```

The 'AST' pane shows the abstract syntax tree representation of the code. The 'Output' pane shows the following output:

```
Math score: 85.0
Science score: 94.5
English score: 100.0
Average score: 93.16666666666667
Bonus: 5.5
Equality check passed.
```

7. Project Setup and Execution Guide

To run the PLC interpreter, the following prerequisites must be met: Python 3.9.x must be installed on the system. The project utilizes PDM (PEP 582 pyproject.toml based Python package manager) for managing dependencies, although it is also possible to run the project without it. The PyQt6 library is required for the GUI, and the SLY (Sly Lexing and Parsing) library is needed for the lexical and syntactic analysis components.

If using PDM, the dependencies can be installed by navigating to the project directory in the terminal and running the command:

```
pdm install
```

To run the application with PDM, use the command:

```
pdm run app
```

If not using PDM, the required libraries can be installed using pip, the standard Python package installer. Open a terminal and run the following command:

```
pip install PyQt6 sly
```

After installing the dependencies, the application can be run directly using the Python interpreter:

```
python main.py
```

These clear and concise instructions make it straightforward for anyone with the necessary prerequisites to set up and run the PLC interpreter, facilitating experimentation with the language and examination of the compiler's functionality.

8. Team Members and Responsibilities

The PLC final project was developed by the following team members, with the responsibilities outlined below:

Table 2: Team Member Contributions

Name	Student ID	Responsibilities
Aman Bhardwaj	125713	Overall project lead, Lexer implementation, Parser implementation, AST design
Huy Nguyen	124724	Implementation of control flow statements (if, while), boolean expressions
Aung Htet Lwin	125773	Implementation of arithmetic expressions, string operations, function definitions

This division of labor demonstrates a collaborative approach to the project, with each team member contributing specific expertise to different aspects of language and compiler implementation. This specialization has likely allowed for more efficient development and a more comprehensive final product.

9. Conclusion and Future Work

The PLC final project represents a successful endeavor in designing and implementing a custom programming language and its compiler. The language incorporates essential features such as various data types, dynamic typing, arithmetic and boolean expressions, string concatenation, control flow statements (if-then-else and while-do), user-defined functions with parameter passing, and a built-in print function. The compiler architecture leverages the SLY library for lexical and syntactic analysis, resulting in the generation of an Abstract Syntax Tree. The runtime environment, built in Python, manages memory, scopes, and function calls. A user-friendly GUI, developed using PyQt6, provides an integrated environment for writing and executing PLC code, as well as visualizing the generated AST.

While the current implementation provides a solid foundation, there are several potential avenues for future enhancements and extensions. One area for improvement could be the addition of support for more complex data types, such as lists or dictionaries, which would significantly increase the language's expressiveness. Another potential extension is the implementation of support for recursion in function calls, which is currently explicitly disallowed. Adding more built-in functions for common tasks could also enhance the language's usability. Improving error handling to provide more specific and informative error messages would greatly benefit the user experience. Furthermore, if the project were to evolve towards a compiled language, exploring optimization techniques for the generated code would be a valuable direction. On the GUI side, enhancements such as syntax highlighting in the code editor and debugging capabilities would further improve the development experience. Overall, the PLC project demonstrates a strong understanding of programming language design and compiler implementation principles, with numerous possibilities for future development and refinement.