CPS2000 - COMPILER THEORY & PRACTICE REPORT

Andrea Filiberto Lucas (0279704L)

Table of Contents

Preamble

It is necessary to run the Main.py script from the terminal in order to use this implementation. This script requires the path to a text file containing the input source code as a command-line argument. To run the script, use the following command:

```
python3 ./path/to/Main.py ./path/to/source.txt
```

In this case, replace `*./path/to/Main.py*` with the actual path to the Main.py script and `*./path/to/source.txt*` with the path to your source code file.
Please ensure that the current working directory (pwd) **is set to the "2. Code" folder when executing this command.**

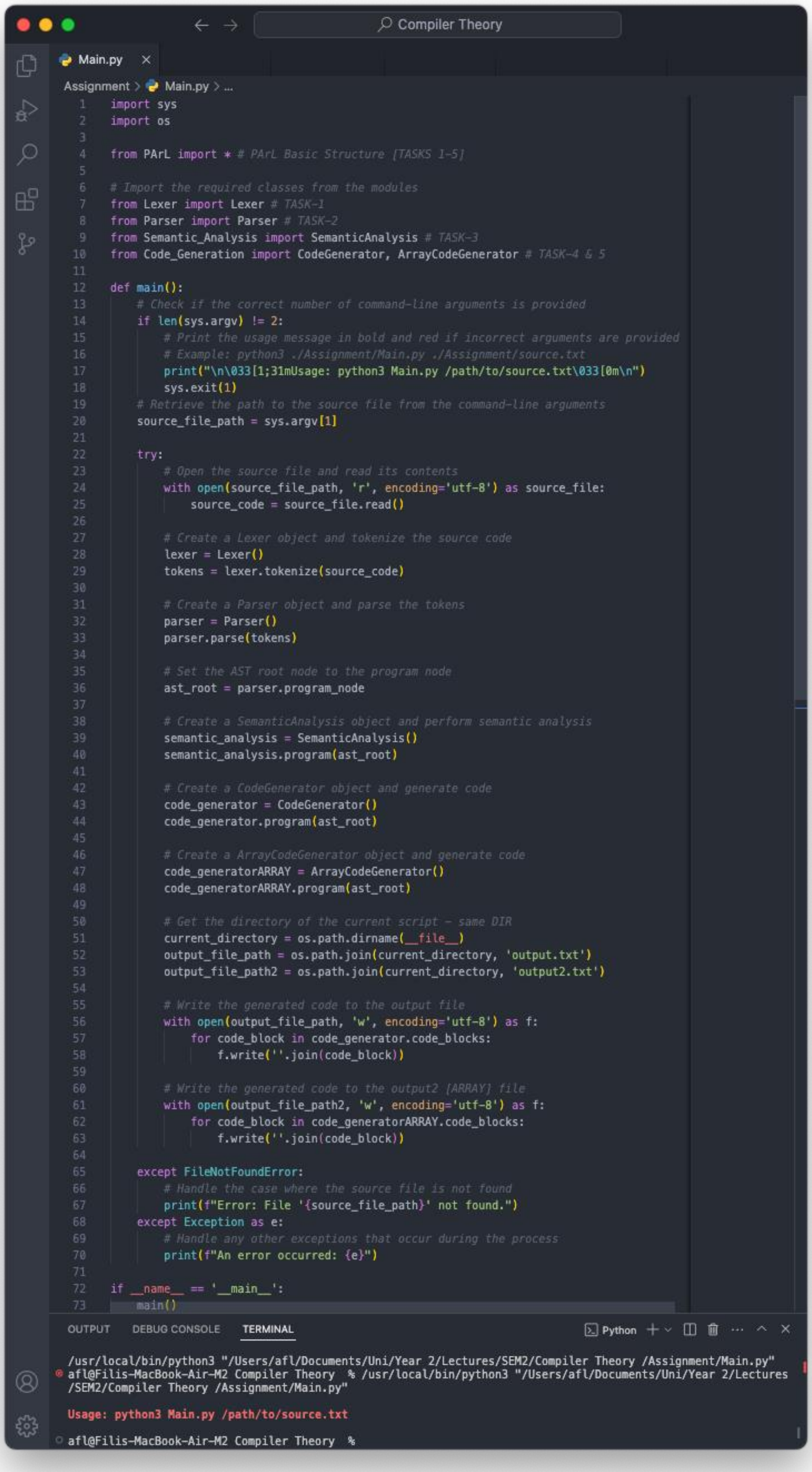
Code Report: Main.py

The Main.py script manages the process of compiling a source code file by combining several components. The script begins by importing the necessary modules. It imports `sys` and `os` for system-level operations, followed by classes (***Lexer, Parser, SemanticAnalysis, CodeGenerator, and ArrayCodeGenerator***) from the other project modules.

The main function contains the script's core functionality. This function first determines whether the correct number of command-line arguments are provided. If not, it prints a usage message in bold red text before exiting. Assuming the correct arguments are entered, the script attempts to open and read the source file specified in the command-line arguments.
Once the source file is read, the script performs lexical analysis by creating a Lexer object and tokenizing the source code. A Parser object then parses the tokens, resulting in an Abstract Syntax Tree (AST). A SemanticAnalysis object is used to perform semantic analysis on the AST, ensuring that the code is correct.

After semantic analysis, the script proceeds to code generation. It creates two code generator objects, CodeGenerator and ArrayCodeGenerator, that generate different sets of code from the AST. The generated code is then written to two separate output files, output.txt and output2.txt, which are saved in the same folder as the script.

The script includes robust error handling to handle a variety of exceptions that may arise during the process. If the source file cannot be found, it prints an error message. It also handles any other exceptions that may occur, providing appropriate error messages to help with debugging.



```python
import sys
import os

from PArL import * # PArL Basic Structure [TASKS 1-5]

# Import the required classes from the modules
from Lexer import Lexer # TASK-1
from Parser import Parser # TASK-2
from Semantic_Analysis import SemanticAnalysis # TASK-3
from Code_Generation import CodeGenerator, ArrayCodeGenerator # TASK-4 & 5

def main():
    # Check if the correct number of command-line arguments is provided
    if len(sys.argv) != 2:
        # Print the usage message in bold and red if incorrect arguments are provided
        # Example: python3 ./Assignment/Main.py ./Assignment/source.txt
        print("\n\033[1;31mUsage: python3 Main.py /path/to/source.txt\033[0m\n")
        sys.exit(1)
    # Retrieve the path to the source file from the command-line arguments
    source_file_path = sys.argv[1]

    try:
        # Open the source file and read its contents
        with open(source_file_path, 'r', encoding='utf-8') as source_file:
            source_code = source_file.read()

        # Create a Lexer object and tokenize the source code
        lexer = Lexer()
        tokens = lexer.tokenize(source_code)

        # Create a Parser object and parse the tokens
        parser = Parser()
        parser.parse(tokens)

        # Set the AST root node to the program node
        ast_root = parser.program_node

        # Create a SemanticAnalysis object and perform semantic analysis
        semantic_analysis = SemanticAnalysis()
        semantic_analysis.program(ast_root)

        # Create a CodeGenerator object and generate code
        code_generator = CodeGenerator()
        code_generator.program(ast_root)

        # Create a ArrayCodeGenerator object and generate code
        code_generatorARRAY = ArrayCodeGenerator()
        code_generatorARRAY.program(ast_root)

        # Get the directory of the current script - same DIR
        current_directory = os.path.dirname(__file__)
        output_file_path = os.path.join(current_directory, 'output.txt')
        output_file_path2 = os.path.join(current_directory, 'output2.txt')

        # Write the generated code to the output file
        with open(output_file_path, 'w', encoding='utf-8') as f:
            for code_block in code_generator.code_blocks:
                f.write(''.join(code_block))

        # Write the generated code to the output2 [ARRAY] file
        with open(output_file_path2, 'w', encoding='utf-8') as f:
            for code_block in code_generatorARRAY.code_blocks:
                f.write(''.join(code_block))

    except FileNotFoundError:
        # Handle the case where the source file is not found
        print(f"Error: File '{source_file_path}' not found.")
    except Exception as e:
        # Handle any other exceptions that occur during the process
        print(f"An error occurred: {e}")

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

```
/usr/local/bin/python3 "/Users/afl/Documents/Uni/Year 2/Lectures/SEM2/Compiler Theory /Assignment/Main.py"
afl@Filis-MacBook-Air-M2 Compiler Theory  % /usr/local/bin/python3 "/Users/afl/Documents/Uni/Year 2/Lectures
/SEM2/Compiler Theory /Assignment/Main.py"

Usage: python3 Main.py /path/to/source.txt

afl@Filis-MacBook-Air-M2 Compiler Theory  %
```

Figure 1: Screenshot showing the terminal if one runs Main.py incorrectly

Code Report: PArL.py

The PArL.py script is a key component of the compiler project, providing essential structures and constants used across multiple tasks. This script is used throughout different stages of compiler development. The script is divided into several sections, with each playing an important role in the compilation process.

The different sections of PArL.py

The script begins by defining several constants that represent various states and tokens and are required for the lexing and parsing stages of compilation. These constants assist in determining the current state of the lexical analyzer or parser, as well as the type of token being processed. For example, states such as INTEGER_LITERAL, FLOAT_LITERAL, and IDENTIFIER assist the lexer and parser in recognising various constructs in the source code.

STATES are used in compilers to keep track of the lexical analyzer or parser's current state or mode. These states help identify various constructs in source code, such as whether the current input is part of an integer literal, a floating-point literal, or a function call. The states are critical for ensuring that the lexer and parser properly transition between different types of input.

TOKENS are the results of the lexical analysis phase; they represent the smallest units of meaning in source code, such as keywords (*ex: if, for*), operators (*ex: +, -, /, **), and literals (*ex: numbers, strings*). Each token has a unique type and value that are used in later stages of compilation to create the syntax tree and generate code.

The **STATE_CONVERTOR** is a mapping between states and their corresponding tokens. This is critical for converting the internal state representation during parsing to the token representation used in subsequent stages of compilation. (*For example, the state INTEGER_LITERAL is mapped to the token <integer_literal>, making it easier to parse the tokenized input*).

TERMINAL_SYMBOLS include identifiers, literals, and various operators, which serve as the foundation for the language's syntax. They are the exact characters or strings found in the source code. Terminal symbols serve as the foundation for the language's grammar, defining the tokens that may be used in source code.

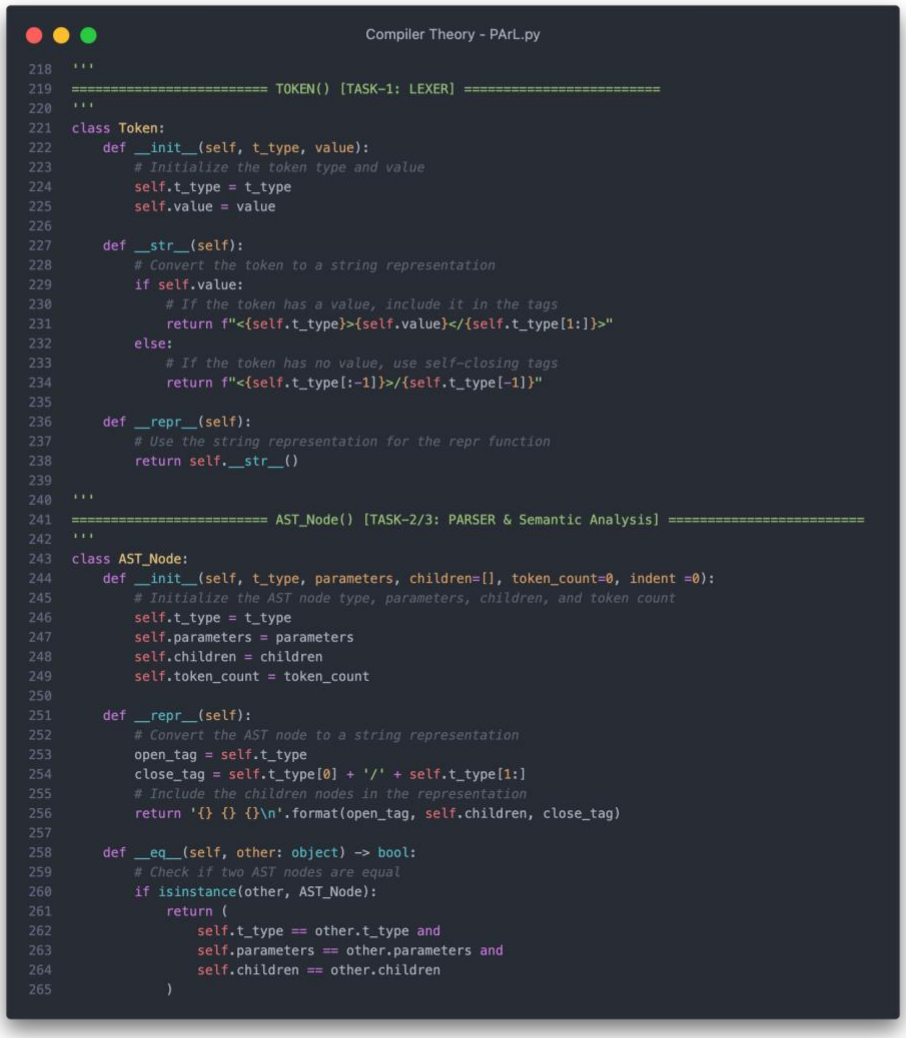
BASIC_PArL lists and categorises the basic elements of the language being compiled, such as the alphabet (*letters and digits*), types (*int, float, bool & color*), and various operators and keywords. This section defines the fundamental building blocks of the language's syntax and semantics. The lexer uses these elements to recognise different parts of the source code and generate the corresponding tokens.

The **INSTRUCTION_SET** specifies the low-level operations that the compiler will produce as the final result of the code generation phase. These instructions represent the operations that the virtual machine can perform, such as arithmetic, memory management, and control flow. The instruction set ensures that the compiled code can be executed properly on the target machine.

Explanation of Token & AST_Node classes

The **Token class** is intended to contain the tokens found during lexical analysis. It has two primary attributes: t_type, which specifies the type of the token (*ex: integer literal, identifier*), and value, which stores the token's actual value (*ex: the specific integer/string*). The '__str__' method returns a string representing the token, which is useful for debugging and logging. This representation includes the type and value of the token, making it clear what each token represents. Furthermore,the '__repr__' method ensures that the token's representation matches its string form, which helps with token list inspection.

The **AST_Node class** represents a node in the Abstract Syntax Tree (AST), a hierarchical structure that depicts the syntactic structure of source code. Each node in the AST represents a construct from the source code, such as an expression, statement, or declaration. The AST_Node class has several attributes: t_type, which indicates the type of construct; parameters, which contain any additional information required for the construct; children, a list of child nodes representing nested constructs; and token_count, which counts the number of tokens used in the construct. The '__repr__' method returns a string representation of the node, including its type and children, which makes it easier to visualise and debug the AST. This hierarchical representation is essential for semantic analysis and code generation because it enables the compiler to understand the structure and relationships between various parts of the source code.



```python
'''
========================= TOKEN() [TASK-1: LEXER] =========================
'''
class Token:
    def __init__(self, t_type, value):
        # Initialize the token type and value
        self.t_type = t_type
        self.value = value

    def __str__(self):
        # Convert the token to a string representation
        if self.value:
            # If the token has a value, include it in the tags
            return f"<{self.t_type}>{self.value}</{self.t_type[1:]}>"
        else:
            # If the token has no value, use self-closing tags
            return f"<{self.t_type[:-1]}>/{self.t_type[-1]}"

    def __repr__(self):
        # Use the string representation for the repr function
        return self.__str__()

'''
========================= AST_Node() [TASK-2/3: PARSER & Semantic Analysis] =========================
'''
class AST_Node:
    def __init__(self, t_type, parameters, children=[], token_count=0, indent =0):
        # Initialize the AST node type, parameters, children, and token count
        self.t_type = t_type
        self.parameters = parameters
        self.children = children
        self.token_count = token_count

    def __repr__(self):
        # Convert the AST node to a string representation
        open_tag = self.t_type
        close_tag = self.t_type[0] + '/' + self.t_type[1:]
        # Include the children nodes in the representation
        return '{} {} {}\n'.format(open_tag, self.children, close_tag)

    def __eq__(self, other: object) -> bool:
        # Check if two AST nodes are equal
        if isinstance(other, AST_Node):
            return (
                self.t_type == other.t_type and
                self.parameters == other.parameters and
                self.children == other.children
            )
```

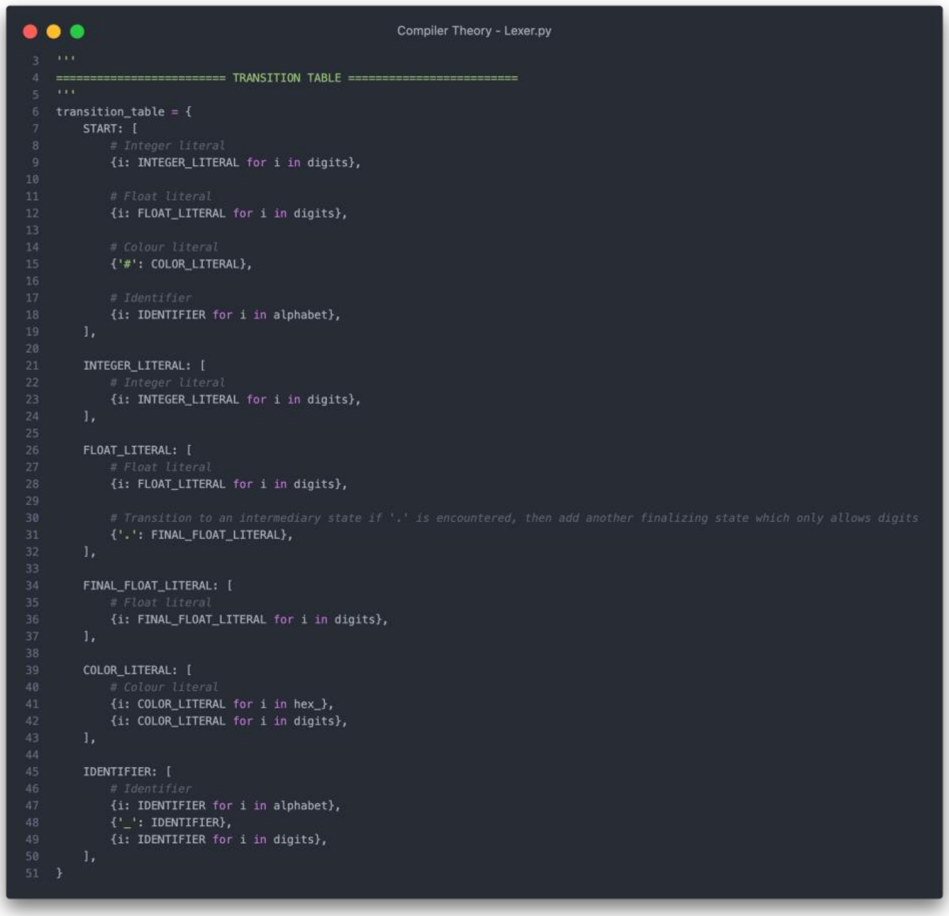
Figure 2: Classes: Token & AST_Node

Task 1 - Table-driven lexer - No Arrays

A lexer, also known as a lexical analyzer, is the first stage of a compiler. Its primary function is to convert the source code from a sequence of characters to a sequence of tokens. Tokens are the smallest units of meaning in source code, and include keywords, operators, identifiers, and literals. The lexer removes any whitespace and comments that are not relevant to syntax analysis. The lexer produces a stream of tokens, which the parser uses to build the syntax tree.

Moreover, a **Transition Table** is a data structure that defines state transitions in a finite state machine, which is a lexer's underlying mechanism. It translates the current state and input character into the next state. The transition table assists the lexer in determining what state to move to next based on the current character being processed. It ensures that the lexer correctly recognises patterns in the source code, by cycling through a set of predefined states.

In **Lexer.py,** the transition table is defined as a dictionary with each key representing a state and the value a list of dictionaries. Each dictionary in the list maps a character (*or set of characters*) to a new state, allowing the lexer to efficiently manage multiple character classes and state transitions.



Compiler Theory - Lexer.py

```python
'''
======================== TRANSITION TABLE ========================
'''
transition_table = {
    START: [
        # Integer literal
        {i: INTEGER_LITERAL for i in digits},

        # Float literal
        {i: FLOAT_LITERAL for i in digits},

        # Colour literal
        {'#': COLOR_LITERAL},

        # Identifier
        {i: IDENTIFIER for i in alphabet},
    ],

    INTEGER_LITERAL: [
        # Integer literal
        {i: INTEGER_LITERAL for i in digits},
    ],

    FLOAT_LITERAL: [
        # Float literal
        {i: FLOAT_LITERAL for i in digits},

        # Transition to an intermediary state if '.' is encountered, then add another finalizing state which only allows digits
        {'.': FINAL_FLOAT_LITERAL},
    ],

    FINAL_FLOAT_LITERAL: [
        # Float literal
        {i: FINAL_FLOAT_LITERAL for i in digits},
    ],

    COLOR_LITERAL: [
        # Colour literal
        {i: COLOR_LITERAL for i in hex_},
        {i: COLOR_LITERAL for i in digits},
    ],

    IDENTIFIER: [
        # Identifier
        {i: IDENTIFIER for i in alphabet},
        {'_': IDENTIFIER},
        {i: IDENTIFIER for i in digits},
    ],
}
```

Figure 3: transition_table

Explanation of the Lexer Class

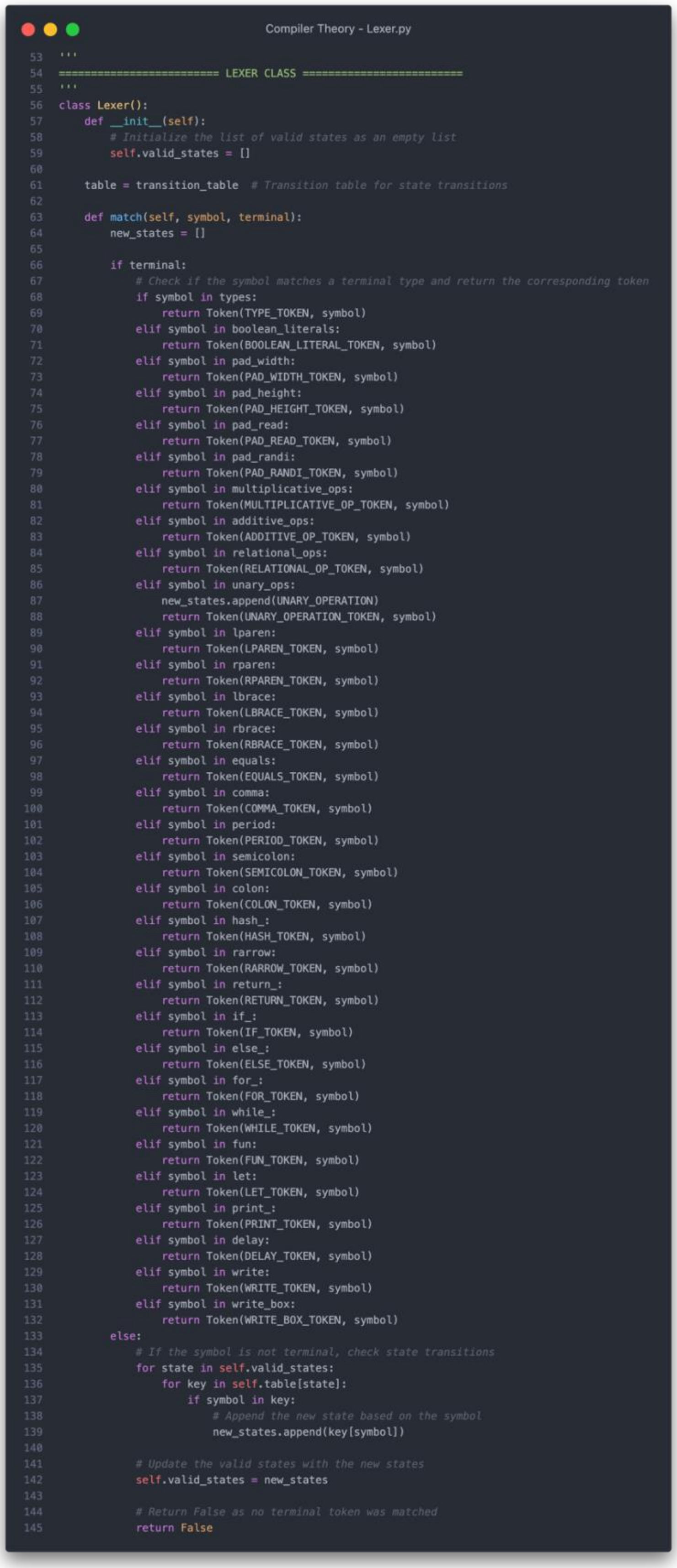
The Lexer class in Lexer.py uses a finite state machine approach to tokenize the input source code.

Initialization

The '__**init**__' method sets the valid_states attribute to an empty list. This list will contain the current valid states of the tokenization process. The **'transition_table**' is assigned to the table attribute, and it defines state transitions.

Match Method

The match method is in charge of assigning symbols to their corresponding tokens or switching between states based on the input symbol.
Terminal Matching: When terminal is True, the method determines whether the symbol matches any terminal symbols (keywords, operators, etc.) and returns the corresponding Token. This involves comparing the symbol to various categories such as types, boolean_literals, multiplicative_ops, and others. If a match is found, a new Token object is generated and returned.
State Transitions: When terminal is False, the method iterates through the current valid_states and updates them using the symbol and transition table. If a symbol corresponds to a key in the transition table for a specific state, the new state is added to new_states. The valid_states are then updated to include new_states.



Compiler Theory - Lexer.py

```python
'''
========================= LEXER CLASS =========================
'''
class Lexer():
    def __init__(self):
        # Initialize the list of valid states as an empty list
        self.valid_states = []

    table = transition_table  # Transition table for state transitions

    def match(self, symbol, terminal):
        new_states = []

        if terminal:
            # Check if the symbol matches a terminal type and return the corresponding token
            if symbol in types:
                return Token(TYPE_TOKEN, symbol)
            elif symbol in boolean_literals:
                return Token(BOOLEAN_LITERAL_TOKEN, symbol)
            elif symbol in pad_width:
                return Token(PAD_WIDTH_TOKEN, symbol)
            elif symbol in pad_height:
                return Token(PAD_HEIGHT_TOKEN, symbol)
            elif symbol in pad_read:
                return Token(PAD_READ_TOKEN, symbol)
            elif symbol in pad_randi:
                return Token(PAD_RANDI_TOKEN, symbol)
            elif symbol in multiplicative_ops:
                return Token(MULTIPLICATIVE_OP_TOKEN, symbol)
            elif symbol in additive_ops:
                return Token(ADDITIVE_OP_TOKEN, symbol)
            elif symbol in relational_ops:
                return Token(RELATIONAL_OP_TOKEN, symbol)
            elif symbol in unary_ops:
                new_states.append(UNARY_OPERATION)
                return Token(UNARY_OPERATION_TOKEN, symbol)
            elif symbol in lparen:
                return Token(LPAREN_TOKEN, symbol)
            elif symbol in rparen:
                return Token(RPAREN_TOKEN, symbol)
            elif symbol in lbrace:
                return Token(LBRACE_TOKEN, symbol)
            elif symbol in rbrace:
                return Token(RBRACE_TOKEN, symbol)
            elif symbol in equals:
                return Token(EQUALS_TOKEN, symbol)
            elif symbol in comma:
                return Token(COMMA_TOKEN, symbol)
            elif symbol in period:
                return Token(PERIOD_TOKEN, symbol)
            elif symbol in semicolon:
                return Token(SEMICOLON_TOKEN, symbol)
            elif symbol in colon:
                return Token(COLON_TOKEN, symbol)
            elif symbol in hash_:
                return Token(HASH_TOKEN, symbol)
            elif symbol in rarrow:
                return Token(RARROW_TOKEN, symbol)
            elif symbol in return_:
                return Token(RETURN_TOKEN, symbol)
            elif symbol in if_:
                return Token(IF_TOKEN, symbol)
            elif symbol in else_:
                return Token(ELSE_TOKEN, symbol)
            elif symbol in for_:
                return Token(FOR_TOKEN, symbol)
            elif symbol in while_:
                return Token(WHILE_TOKEN, symbol)
            elif symbol in fun:
                return Token(FUN_TOKEN, symbol)
            elif symbol in let:
                return Token(LET_TOKEN, symbol)
            elif symbol in print_:
                return Token(PRINT_TOKEN, symbol)
            elif symbol in delay:
                return Token(DELAY_TOKEN, symbol)
            elif symbol in write:
                return Token(WRITE_TOKEN, symbol)
            elif symbol in write_box:
                return Token(WRITE_BOX_TOKEN, symbol)
        else:
            # If the symbol is not terminal, check state transitions
            for state in self.valid_states:
                for key in self.table[state]:
                    if symbol in key:
                        # Append the new state based on the symbol
                        new_states.append(key[symbol])

            # Update the valid states with the new states
            self.valid_states = new_states

            # Return False as no terminal token was matched
            return False
```

Figure 4: Lexer Class

Explanation of the Tokenize Function

The **tokenize function** is the main method that processes the input source code and produces a list of tokens.

String Preprocessing

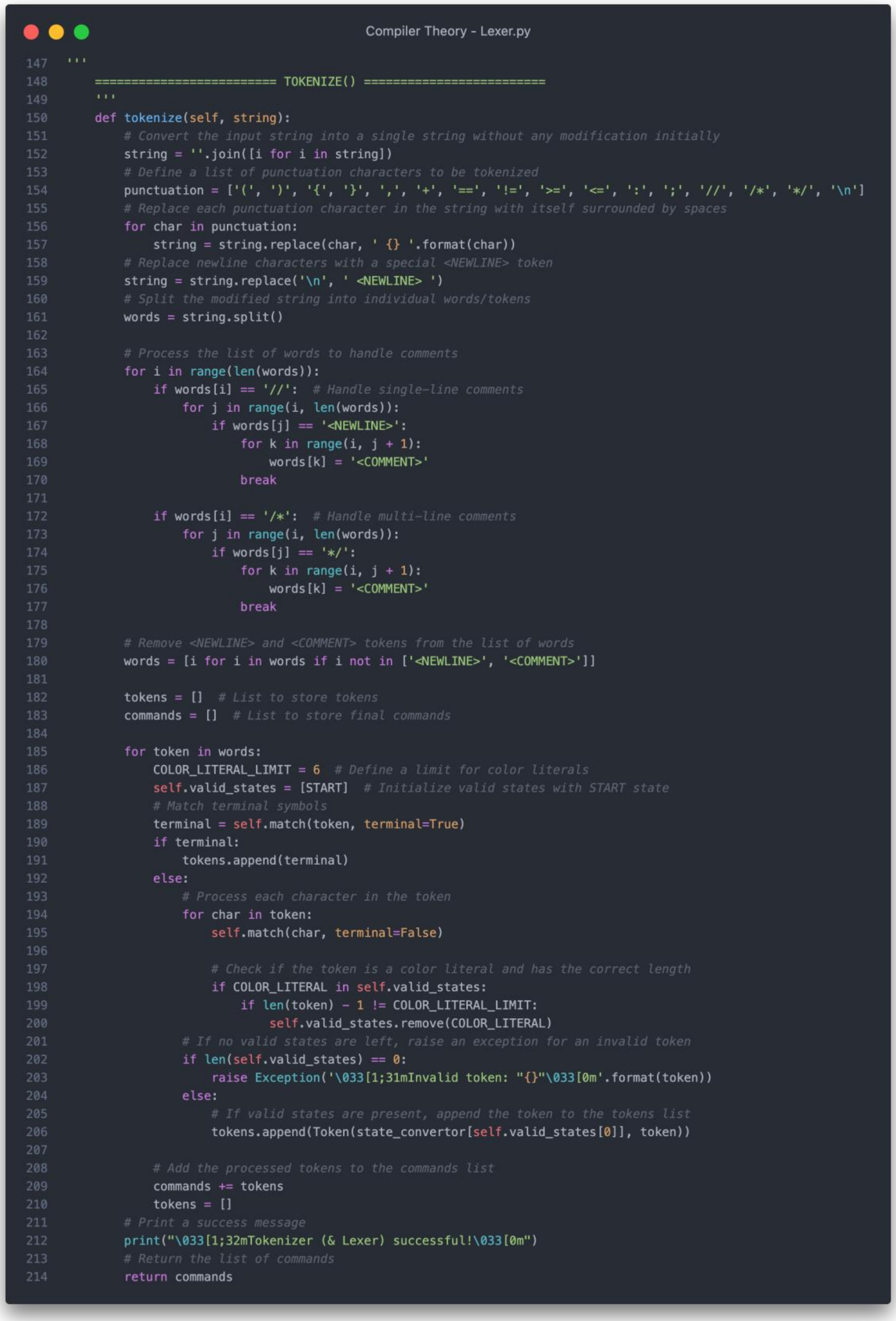
The function begins by converting the input string into a single continuous string and specifying a list of punctuation characters that must be tokenized separately. Each punctuation character in the string is surrounded by spaces to ensure that they are treated as distinct tokens. To replace newline characters, use the **<NEWLINE>** token.

Comment Handling

The function uses the list of words to handle comments. Single-line comments (***beginning with //***) are identified and replaced with **<COMMENT>** tokens until a newline occurs. Such tokens are also used to replace multi-line comments, which are enclosed by **/* and */.**

Tokenizaton

The core of the tokenize function processes each word/token in the input string. For each token, the function initialises the `` `valid_states` `` with the `` `START` `` state and first looks for terminal symbols. If a terminal match is discovered, the associated token is added to the tokens list. If no terminal match is found, the function processes each character in the token and updates the `` `valid_states` `` using the transition table. If a token is identified as a colour literal, the function ensures that it is of the correct length (*6 characters*). If there are no valid states after processing a token, an invalid token exception is raised. Otherwise, a valid token is added to the `` `tokens` `` list. Finally, the processed tokens are added to the `` `command` `` list and a success message is displayed. The function returns a list of commands, which represent the tokenized source code.



```python
'''
    ========================= TOKENIZE() =========================
    '''
    def tokenize(self, string):
        # Convert the input string into a single string without any modification initially
        string = ''.join([i for i in string])
        # Define a list of punctuation characters to be tokenized
        punctuation = ['(', ')', '{', '}', ',', '+', '==', '!=', '>=', '<=', ':', ';', '//', '/*', '*/', '\n']
        # Replace each punctuation character in the string with itself surrounded by spaces
        for char in punctuation:
            string = string.replace(char, ' {} '.format(char))
        # Replace newline characters with a special <NEWLINE> token
        string = string.replace('\n', ' <NEWLINE> ')
        # Split the modified string into individual words/tokens
        words = string.split()

        # Process the list of words to handle comments
        for i in range(len(words)):
            if words[i] == '//':  # Handle single-line comments
                for j in range(i, len(words)):
                    if words[j] == '<NEWLINE>':
                        for k in range(i, j + 1):
                            words[k] = '<COMMENT>'
                        break

            if words[i] == '/*':  # Handle multi-line comments
                for j in range(i, len(words)):
                    if words[j] == '*/':
                        for k in range(i, j + 1):
                            words[k] = '<COMMENT>'
                        break

        # Remove <NEWLINE> and <COMMENT> tokens from the list of words
        words = [i for i in words if i not in ['<NEWLINE>', '<COMMENT>']]

        tokens = []  # List to store tokens
        commands = []  # List to store final commands

        for token in words:
            COLOR_LITERAL_LIMIT = 6  # Define a limit for color literals
            self.valid_states = [START]  # Initialize valid states with START state
            # Match terminal symbols
            terminal = self.match(token, terminal=True)
            if terminal:
                tokens.append(terminal)
            else:
                # Process each character in the token
                for char in token:
                    self.match(char, terminal=False)

                    # Check if the token is a color literal and has the correct length
                    if COLOR_LITERAL in self.valid_states:
                        if len(token) - 1 != COLOR_LITERAL_LIMIT:
                            self.valid_states.remove(COLOR_LITERAL)
                # If no valid states are left, raise an exception for an invalid token
                if len(self.valid_states) == 0:
                    raise Exception('\033[1;31mInvalid token: "{}"\033[0m'.format(token))
                else:
                    # If valid states are present, append the token to the tokens list
                    tokens.append(Token(state_convertor[self.valid_states[0]], token))

            # Add the processed tokens to the commands list
            commands += tokens
            tokens = []
        # Print a success message
        print("\033[1;32mTokenizer (& Lexer) successful!\033[0m")
        # Return the list of commands
        return commands
```

Figure 5: Tokenize() Function

Task 2 - Hand-crafted LL (K) parser - No Arrays

A parser is an essential part of a compiler that organises the lexer's token sequence into a structured representation known as an Abstract Syntax Tree (AST). The parser checks that the sequence of tokens follows the grammatical rules of the programming language. If the tokens form a valid sequence, the parser generates an AST, which represents the source code's hierarchical structure. The AST is then used in subsequent stages of compilation, including semantic analysis and code generation. The parser's primary goal is to ensure that the source code is syntactically correct and to generate an intermediate representation that accurately depicts the program's structure.

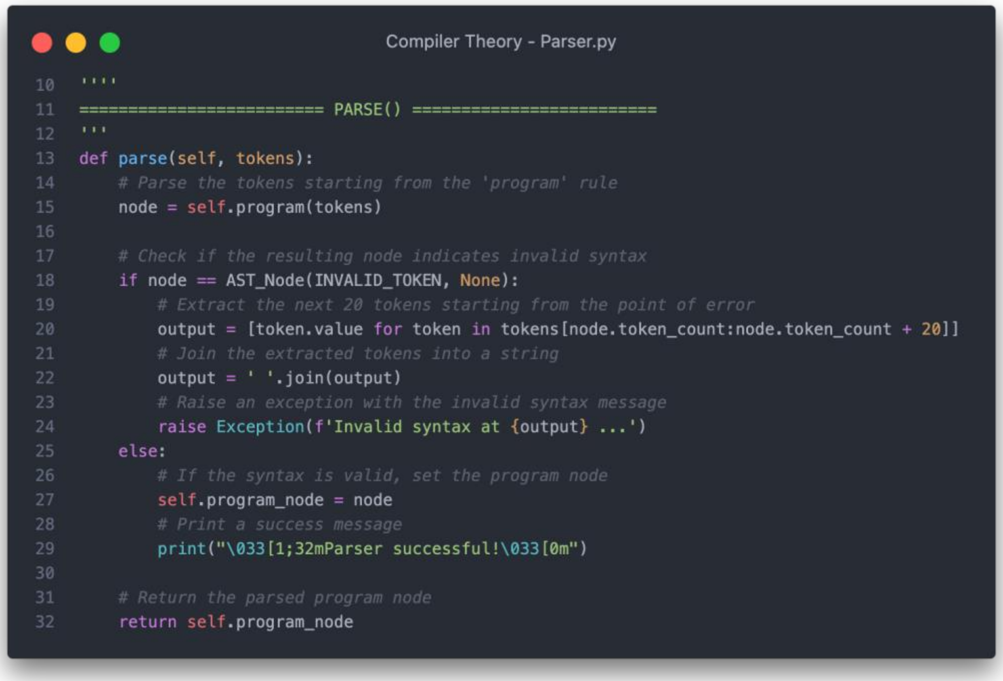
The **Parser.py script** defines **a Parser class** that converts tokens into an AST. It includes a variety of methods for dealing with different aspects of the language's syntax, such as statements, expressions, and function declarations. The primary method is parse, which orchestrates the parsing process beginning with the highest-level construct.

Parser Class

The Parser class is initialised with the program_node attribute set to None. If the parsing succeeds, this attribute will eventually contain the AST's root node.

Parse Method

The parse method is the parser's entry point. It starts by calling the programme method to begin parsing the tokens from the top-level construct, which is usually the entire programme. If the programme method returns an invalid node, indicating a syntax error, the parse method extracts and prints the incorrect portion of the source code while throwing an exception. If the parsing is successful, program_node stores the AST's root node and prints a success message.

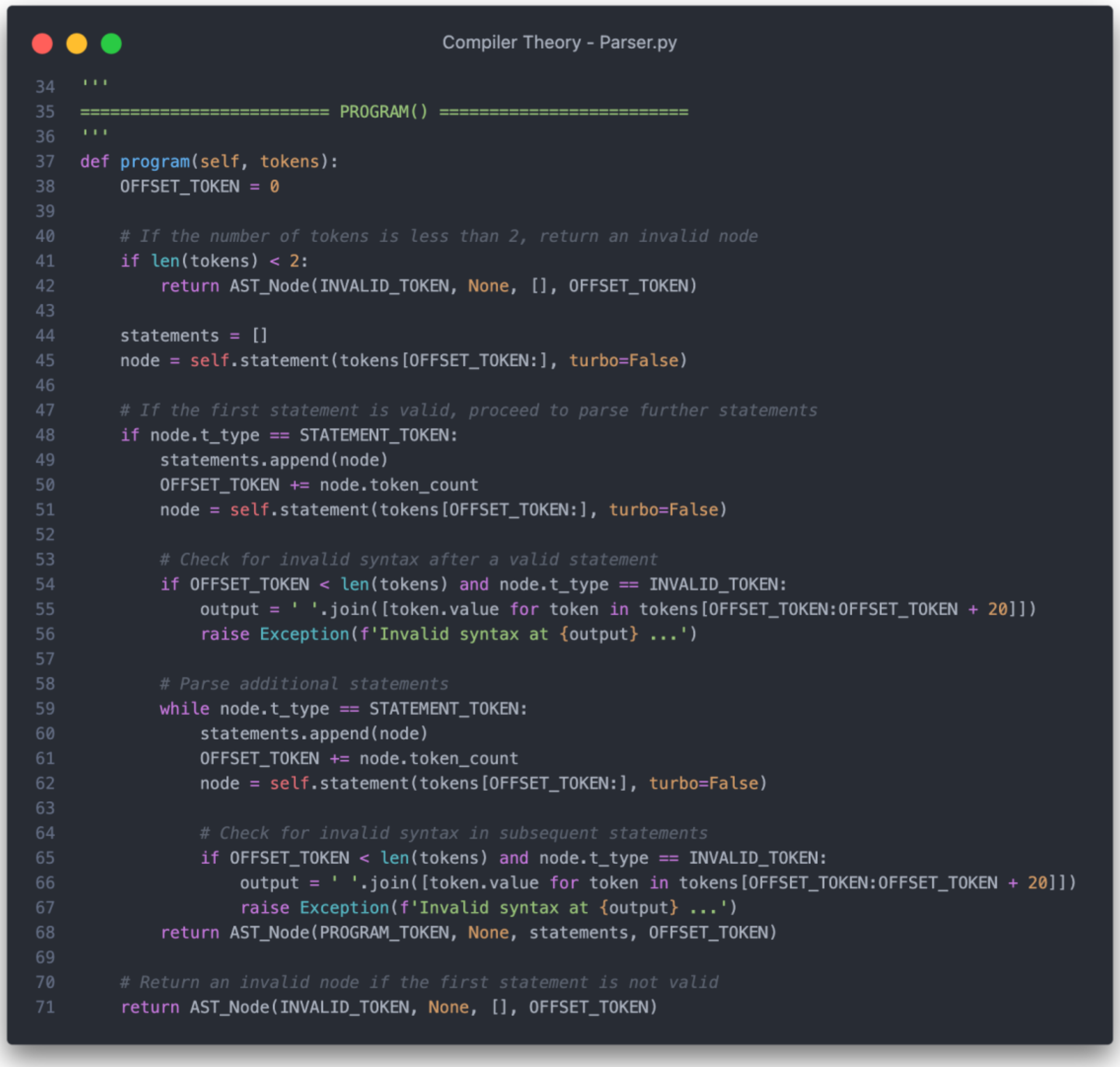


```python
''''
======================== PARSE() ========================
'''
def parse(self, tokens):
    # Parse the tokens starting from the 'program' rule
    node = self.program(tokens)

    # Check if the resulting node indicates invalid syntax
    if node == AST_Node(INVALID_TOKEN, None):
        # Extract the next 20 tokens starting from the point of error
        output = [token.value for token in tokens[node.token_count:node.token_count + 20]]
        # Join the extracted tokens into a string
        output = ' '.join(output)
        # Raise an exception with the invalid syntax message
        raise Exception(f'Invalid syntax at {output} ...')
    else:
        # If the syntax is valid, set the program node
        self.program_node = node
        # Print a success message
        print("\033[1;32mParser successful!\033[0m")

    # Return the parsed program node
    return self.program_node
```

Figure 6: Parse Method

Program Method

The program method parses the entire programme. It initialises an offset and determines whether there are enough tokens to create a valid programme. It begins by parsing the first statement via the statement method. If the first statement is valid, it continues to parse additional statements, adding them to the statements list. If any statement is invalid, it throws an exception containing the incorrect portion of the source code. Finally, it **returns an AST_Node** that represents the entire programme.

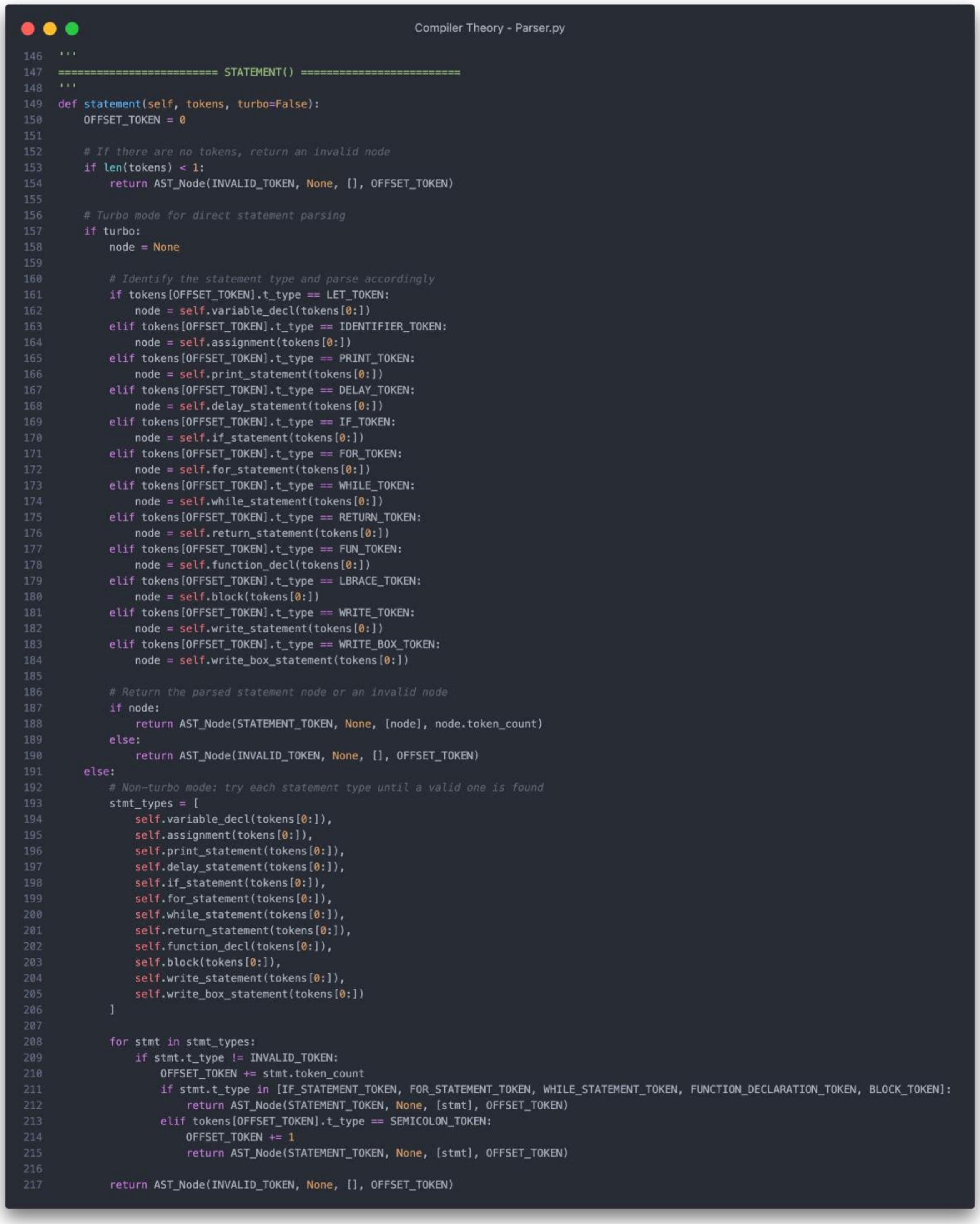


```python
'''
======================== PROGRAM() ========================
'''
def program(self, tokens):
    OFFSET_TOKEN = 0

    # If the number of tokens is less than 2, return an invalid node
    if len(tokens) < 2:
        return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)

    statements = []
    node = self.statement(tokens[OFFSET_TOKEN:], turbo=False)

    # If the first statement is valid, proceed to parse further statements
    if node.t_type == STATEMENT_TOKEN:
        statements.append(node)
        OFFSET_TOKEN += node.token_count
        node = self.statement(tokens[OFFSET_TOKEN:], turbo=False)

        # Check for invalid syntax after a valid statement
        if OFFSET_TOKEN < len(tokens) and node.t_type == INVALID_TOKEN:
            output = ' '.join([token.value for token in tokens[OFFSET_TOKEN:OFFSET_TOKEN + 20]])
            raise Exception(f'Invalid syntax at {output} ...')

        # Parse additional statements
        while node.t_type == STATEMENT_TOKEN:
            statements.append(node)
            OFFSET_TOKEN += node.token_count
            node = self.statement(tokens[OFFSET_TOKEN:], turbo=False)

            # Check for invalid syntax in subsequent statements
            if OFFSET_TOKEN < len(tokens) and node.t_type == INVALID_TOKEN:
                output = ' '.join([token.value for token in tokens[OFFSET_TOKEN:OFFSET_TOKEN + 20]])
                raise Exception(f'Invalid syntax at {output} ...')
        return AST_Node(PROGRAM_TOKEN, None, statements, OFFSET_TOKEN)

    # Return an invalid node if the first statement is not valid
    return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)
```

Figure 7: Program Method

Statement Method

The statement method parses various types of statements. It operates in two modes: turbo and non-turbo. In turbo mode, the method identifies and parses the statement type using the first token. In non-turbo mode, it tests each statement type sequentially until a valid one is found. If no valid statement is found, the method returns an invalid node rather than an AST_Node representing the parsed statement.

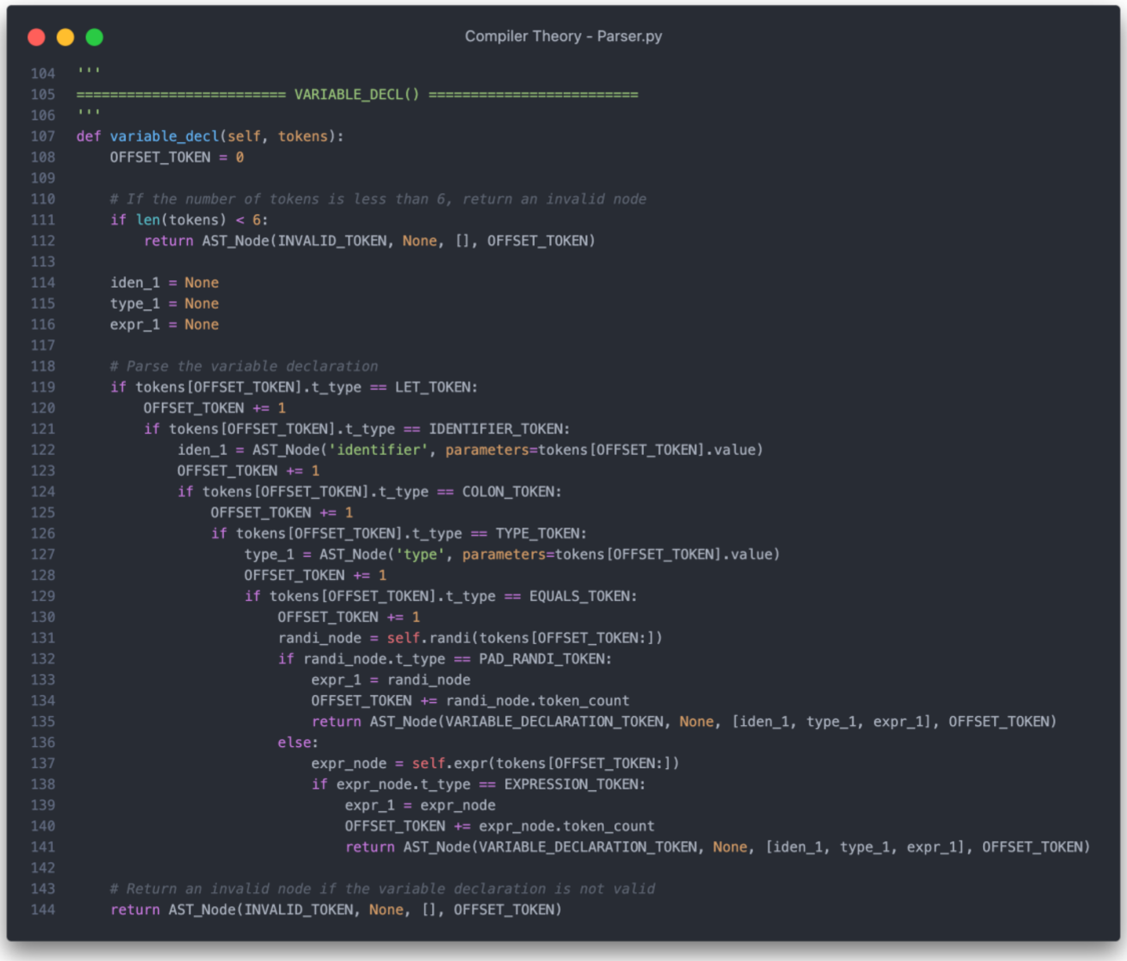


```python
'''
======================== STATEMENT() ========================
'''
def statement(self, tokens, turbo=False):
    OFFSET_TOKEN = 0

    # If there are no tokens, return an invalid node
    if len(tokens) < 1:
        return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)

    # Turbo mode for direct statement parsing
    if turbo:
        node = None

        # Identify the statement type and parse accordingly
        if tokens[OFFSET_TOKEN].t_type == LET_TOKEN:
            node = self.variable_decl(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == IDENTIFIER_TOKEN:
            node = self.assignment(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == PRINT_TOKEN:
            node = self.print_statement(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == DELAY_TOKEN:
            node = self.delay_statement(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == IF_TOKEN:
            node = self.if_statement(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == FOR_TOKEN:
            node = self.for_statement(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == WHILE_TOKEN:
            node = self.while_statement(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == RETURN_TOKEN:
            node = self.return_statement(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == FUN_TOKEN:
            node = self.function_decl(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == LBRACE_TOKEN:
            node = self.block(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == WRITE_TOKEN:
            node = self.write_statement(tokens[0:])
        elif tokens[OFFSET_TOKEN].t_type == WRITE_BOX_TOKEN:
            node = self.write_box_statement(tokens[0:])

        # Return the parsed statement node or an invalid node
        if node:
            return AST_Node(STATEMENT_TOKEN, None, [node], node.token_count)
        else:
            return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)
    else:
        # Non-turbo mode: try each statement type until a valid one is found
        stmt_types = [
            self.variable_decl(tokens[0:]),
            self.assignment(tokens[0:]),
            self.print_statement(tokens[0:]),
            self.delay_statement(tokens[0:]),
            self.if_statement(tokens[0:]),
            self.for_statement(tokens[0:]),
            self.while_statement(tokens[0:]),
            self.return_statement(tokens[0:]),
            self.function_decl(tokens[0:]),
            self.block(tokens[0:]),
            self.write_statement(tokens[0:]),
            self.write_box_statement(tokens[0:])
        ]

        for stmt in stmt_types:
            if stmt.t_type != INVALID_TOKEN:
                OFFSET_TOKEN += stmt.token_count
                if stmt.t_type in [IF_STATEMENT_TOKEN, FOR_STATEMENT_TOKEN, WHILE_STATEMENT_TOKEN, FUNCTION_DECLARATION_TOKEN, BLOCK_TOKEN]:
                    return AST_Node(STATEMENT_TOKEN, None, [stmt], OFFSET_TOKEN)
                elif tokens[OFFSET_TOKEN].t_type == SEMICOLON_TOKEN:
                    OFFSET_TOKEN += 1
                    return AST_Node(STATEMENT_TOKEN, None, [stmt], OFFSET_TOKEN)

        return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)
```

Figure 8: Statement Method

Variable Declaration Method

The variable_decl method parses variable declarations. It starts by determining whether there are sufficient tokens to form a valid variable declaration. If the declaration is valid, it parses the variable name, type, and initialization expression. If any part of the declaration is invalid, it will return an invalid node.

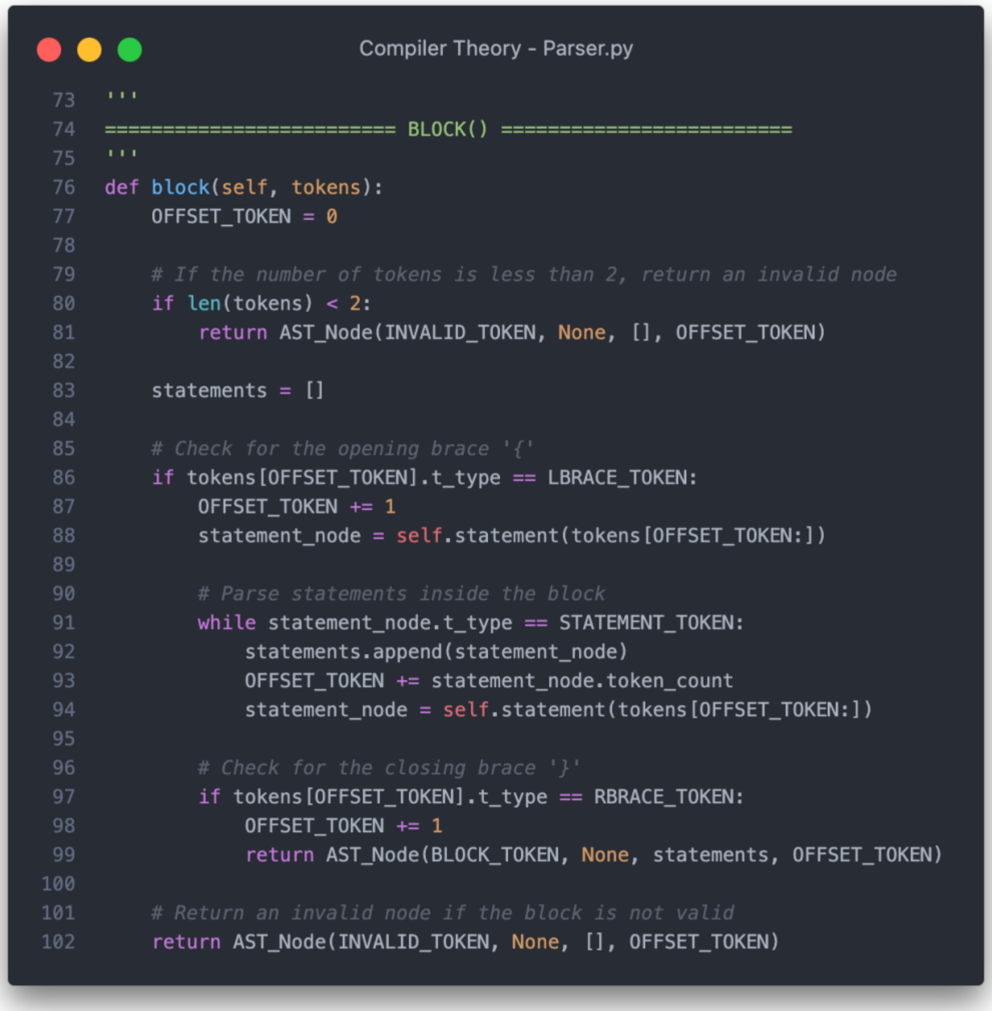


```python
'''
======================== VARIABLE_DECL() ========================
'''
def variable_decl(self, tokens):
    OFFSET_TOKEN = 0

    # If the number of tokens is less than 6, return an invalid node
    if len(tokens) < 6:
        return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)

    iden_1 = None
    type_1 = None
    expr_1 = None

    # Parse the variable declaration
    if tokens[OFFSET_TOKEN].t_type == LET_TOKEN:
        OFFSET_TOKEN += 1
        if tokens[OFFSET_TOKEN].t_type == IDENTIFIER_TOKEN:
            iden_1 = AST_Node('identifier', parameters=tokens[OFFSET_TOKEN].value)
            OFFSET_TOKEN += 1
            if tokens[OFFSET_TOKEN].t_type == COLON_TOKEN:
                OFFSET_TOKEN += 1
                if tokens[OFFSET_TOKEN].t_type == TYPE_TOKEN:
                    type_1 = AST_Node('type', parameters=tokens[OFFSET_TOKEN].value)
                    OFFSET_TOKEN += 1
                    if tokens[OFFSET_TOKEN].t_type == EQUALS_TOKEN:
                        OFFSET_TOKEN += 1
                        randi_node = self.randi(tokens[OFFSET_TOKEN:])
                        if randi_node.t_type == PAD_RANDI_TOKEN:
                            expr_1 = randi_node
                            OFFSET_TOKEN += randi_node.token_count
                            return AST_Node(VARIABLE_DECLARATION_TOKEN, None, [iden_1, type_1, expr_1], OFFSET_TOKEN)
                        else:
                            expr_node = self.expr(tokens[OFFSET_TOKEN:])
                            if expr_node.t_type == EXPRESSION_TOKEN:
                                expr_1 = expr_node
                                OFFSET_TOKEN += expr_node.token_count
                                return AST_Node(VARIABLE_DECLARATION_TOKEN, None, [iden_1, type_1, expr_1], OFFSET_TOKEN)

    # Return an invalid node if the variable declaration is not valid
    return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)
```

Figure 9: Variable Decleration Method

Block Method

The block method parses a block of statements enclosed in braces ({}). It begins by looking for an opening brace and then parses the statements within the block until a closing brace is found. If the block is valid, it returns an AST_Node corresponding to the block; otherwise, it returns an invalid node.



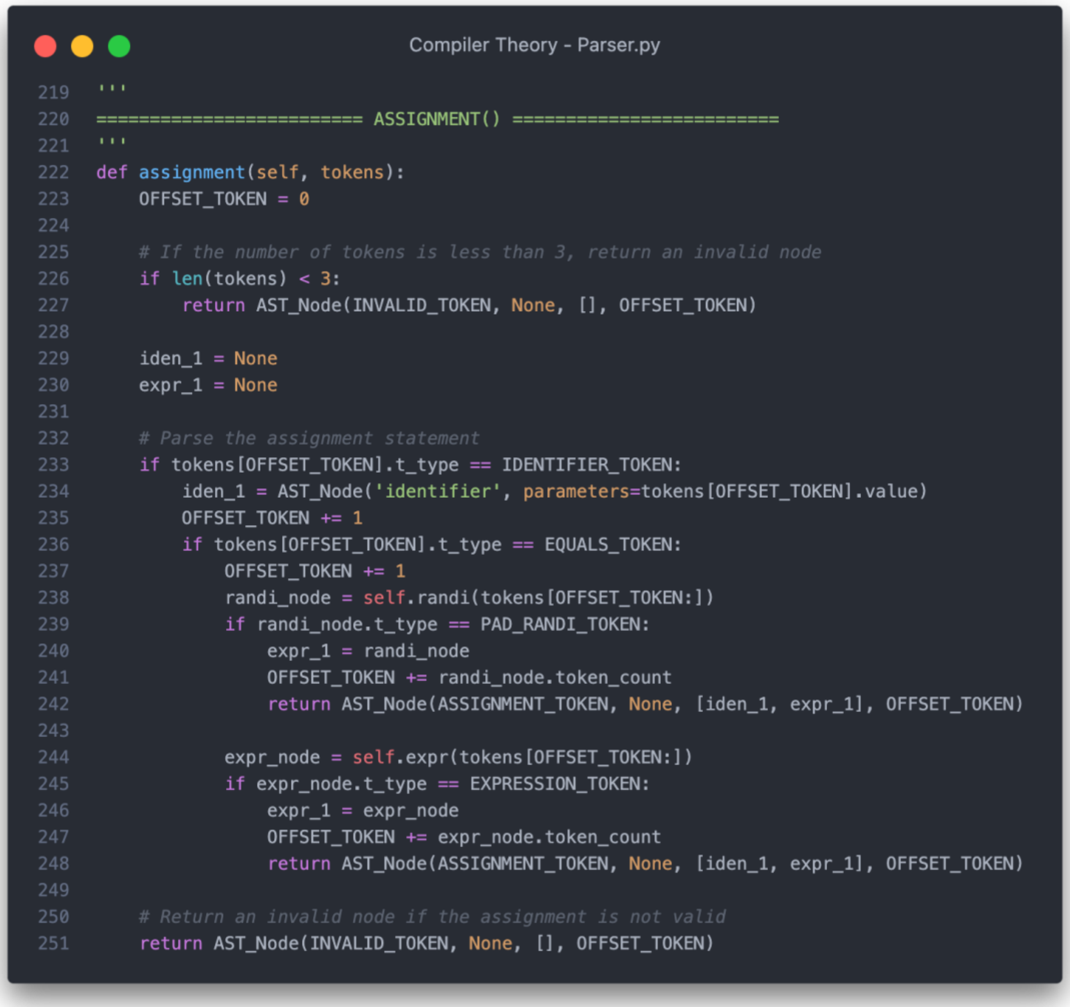
Compiler Theory - Parser.py

```python
'''
======================== BLOCK() ========================
'''
def block(self, tokens):
    OFFSET_TOKEN = 0

    # If the number of tokens is less than 2, return an invalid node
    if len(tokens) < 2:
        return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)

    statements = []

    # Check for the opening brace '{'
    if tokens[OFFSET_TOKEN].t_type == LBRACE_TOKEN:
        OFFSET_TOKEN += 1
        statement_node = self.statement(tokens[OFFSET_TOKEN:])

        # Parse statements inside the block
        while statement_node.t_type == STATEMENT_TOKEN:
            statements.append(statement_node)
            OFFSET_TOKEN += statement_node.token_count
            statement_node = self.statement(tokens[OFFSET_TOKEN:])

        # Check for the closing brace '}'
        if tokens[OFFSET_TOKEN].t_type == RBRACE_TOKEN:
            OFFSET_TOKEN += 1
            return AST_Node(BLOCK_TOKEN, None, statements, OFFSET_TOKEN)

    # Return an invalid node if the block is not valid
    return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)
```

Figure 10: Block Method

Assignment Method

The assignment method interprets assignment statements. It determines whether there are enough tokens and whether they follow the proper pattern for an assignment (*identifier, equals sign, and expression*). If the assignment is valid, it returns an AST_Node, otherwise an invalid node.



Compiler Theory - Parser.py

```python
'''
========================= ASSIGNMENT() =========================
'''
def assignment(self, tokens):
    OFFSET_TOKEN = 0

    # If the number of tokens is less than 3, return an invalid node
    if len(tokens) < 3:
        return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)

    iden_1 = None
    expr_1 = None

    # Parse the assignment statement
    if tokens[OFFSET_TOKEN].t_type == IDENTIFIER_TOKEN:
        iden_1 = AST_Node('identifier', parameters=tokens[OFFSET_TOKEN].value)
        OFFSET_TOKEN += 1
        if tokens[OFFSET_TOKEN].t_type == EQUALS_TOKEN:
            OFFSET_TOKEN += 1
            randi_node = self.randi(tokens[OFFSET_TOKEN:])
            if randi_node.t_type == PAD_RANDI_TOKEN:
                expr_1 = randi_node
                OFFSET_TOKEN += randi_node.token_count
                return AST_Node(ASSIGNMENT_TOKEN, None, [iden_1, expr_1], OFFSET_TOKEN)

            expr_node = self.expr(tokens[OFFSET_TOKEN:])
            if expr_node.t_type == EXPRESSION_TOKEN:
                expr_1 = expr_node
                OFFSET_TOKEN += expr_node.token_count
                return AST_Node(ASSIGNMENT_TOKEN, None, [iden_1, expr_1], OFFSET_TOKEN)

    # Return an invalid node if the assignment is not valid
    return AST_Node(INVALID_TOKEN, None, [], OFFSET_TOKEN)
```

Figure 11: Assignment Method

Formal Parameter Methods

The formal_params method parses a set of formal parameters, whereas the formal_param method parses a single formal parameter. Both methods ensure that the parameters have the proper syntax and return either valid AST_Node representations or invalid nodes.

Overview of Other Methods and Functions

Print Statement Method

The print_statement method parses printed statements. It determines whether the tokens match the pattern for a print statement (*print keyword followed by expression*). If valid, it returns an AST_Node containing the print statement; otherwise, it returns an invalid node.

Delay Statement Method

The delay_statement() method parses delay statements. It determines whether the tokens match the pattern for a delay statement (*delay keyword followed by an expression*). If valid, it returns an AST_Node containing the delay statement; otherwise, it returns an invalid node.

If Statement Method

The if_statement() method parses if statements. It looks for the if keyword, followed by a parenthesized condition and a series of statements. It may also check for an else block. If it is valid, it returns an AST_Node containing the if statement; otherwise, it returns an invalid node.

For Statement Method

The for_statement() method parses for loops. It looks for the for keyword, followed by initialization, condition, and post-expression, all enclosed in brackets, and then a block of statements. If valid, it returns an AST_Node containing the for loop; otherwise, it returns an invalid node.

While Statement Method

Similarly, the while_statement method handles while loops. It looks for the while keyword, followed by a parenthesized condition and a series of statements. If valid, it returns an AST_Node containing the while loop; otherwise, it returns an invalid node.

Return Statement Method

The return_statement method processes return statements. It looks for the return keyword, followed by an expression. If valid, it returns an AST_Node containing the return statement; otherwise, it returns an invalid node.

Unary Method

The unary method interprets unary operations. It looks for a unary operator, followed by an expression. If valid, it returns an AST_Node that represents the unary operation; otherwise, it returns an invalid node.

Factor Method

The factor method parses factors, which can include literals, identifiers, function calls, subexpressions, and unary operations. It evaluates each possibility in turn and returns a valid AST_Node representing the factor, or an invalid node if no valid factor is discovered.

Term Method

The term method parses terms composed of factors that may be separated by multiplicative operators. It parses the factors and operators, stacking them if necessary, and returns an AST_Node containing the term or an invalid node if no valid term is discovered.

Simple Expression Method

The simple_expr method parses simple expressions composed of terms that may be separated by additive operators. It parses the terms and operators, stacking operations as needed, and returns an AST_Node containing the simple expression or an invalid node if no valid simple expression is found.

Expression Method

The expr method parses expressions, which can contain relational operators that separate simple expressions. It parses simple expressions and relational operators, stacking operations as needed, and returns either an AST_Node representing the expression or an invalid node if no valid expression is found.

Literal Method

The literal method accepts boolean, integer, float, and colour literals, as well as PAD operations. It returns an AST_Node that represents the literal, or an invalid node if no valid literal is found.

Function Call Method

The function_call method processes function calls. It looks for an identifier, followed by a parenthesized list of actual parameters. If valid, it returns an AST_Node that represents the function call; otherwise, it returns an invalid node.

The Actual Parameters Method

The actual_params method parses a list of actual parameters, ensuring that the syntax is correct, and returns an AST_Node representing the parameters, or an invalid node if no valid parameters are found.

Sub Expression Method

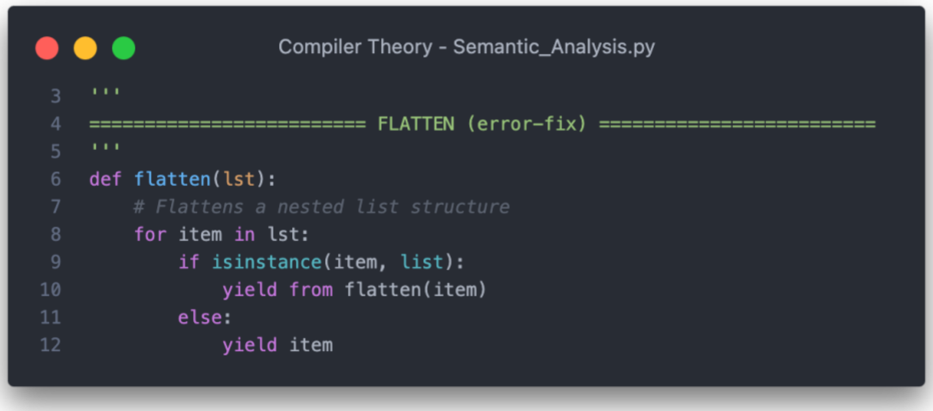
The sub_expr method parses subexpressions that are enclosed in brackets. It returns an AST_Node if the sub-expression is valid, or an invalid node otherwise.

Task 3 - Semantic Analysis Pass - No Arrays

After syntax analysis (parsing), the compiler design process moves on to **semantic analysis**. Syntax analysis ensures that the source code is structurally correct and follows the grammar rules of the language, whereas semantic analysis ensures that it is meaningful and logically consistent. This phase detects semantic errors like type mismatches, undeclared variables, and scope violations. Semantic analysis is critical for ensuring that the program's operations make sense and follow the rules of the language, thereby avoiding runtime errors and ensuring correct programme behaviour.

The Need for Flattening

In semantic analysis, flattening is the process of transforming a nested list structure into a single, flat list. This is necessary because the Abstract Syntax Tree (AST) generated by the parser may contain deeply nested nodes. Flattening makes it easier to traverse and analyse the AST by providing a linear sequence of nodes. This makes it easier to apply semantic rules and check each node consistently.



```python
'''
========================= FLATTEN (error-fix) =========================
'''
def flatten(lst):
    # Flattens a nested list structure
    for item in lst:
        if isinstance(item, list):
            yield from flatten(item)
        else:
            yield item
```

Figure 12: Flatten Method

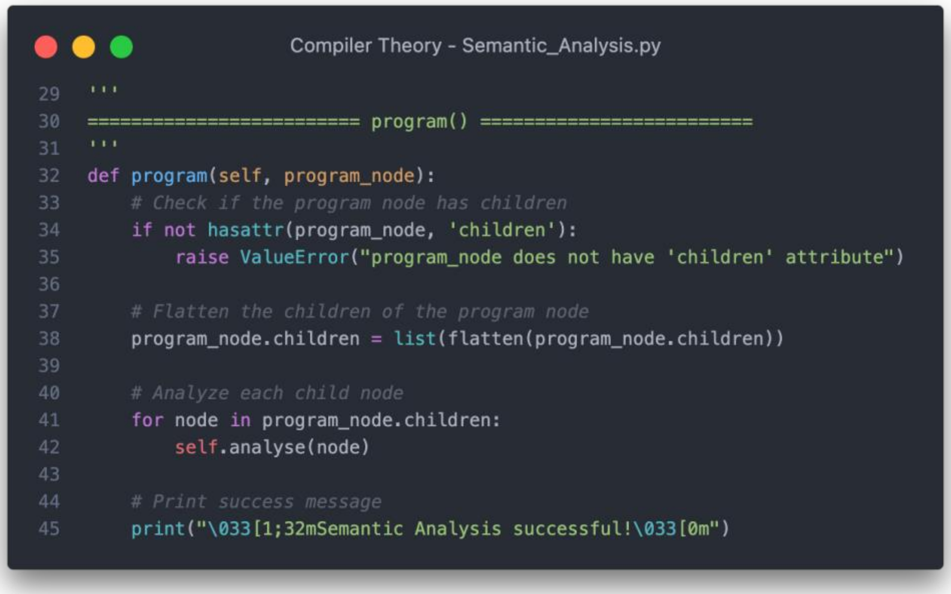
The script's flatten function recursively iterates through the nested list structure and returns each item, ensuring that the resulting structure is a flat list. This function is used in the programme and analysis methods to ensure that the AST is easy to navigate and analyse.

Semantic Analysis Class

The SemanticAnalysis class performs the program's semantic analysis. It starts with a stack that handles variable scopes and default values for various data types. The stack keeps track of variable declarations and ensures that variables are used within their proper scope.

Program Method

The program method is the starting point for semantic analysis. It first determines whether the programme node has children, then flattens and analyses each one. If the program node does not have any children, it returns a ValueError. If the analysis is successful, it outputs a success message.

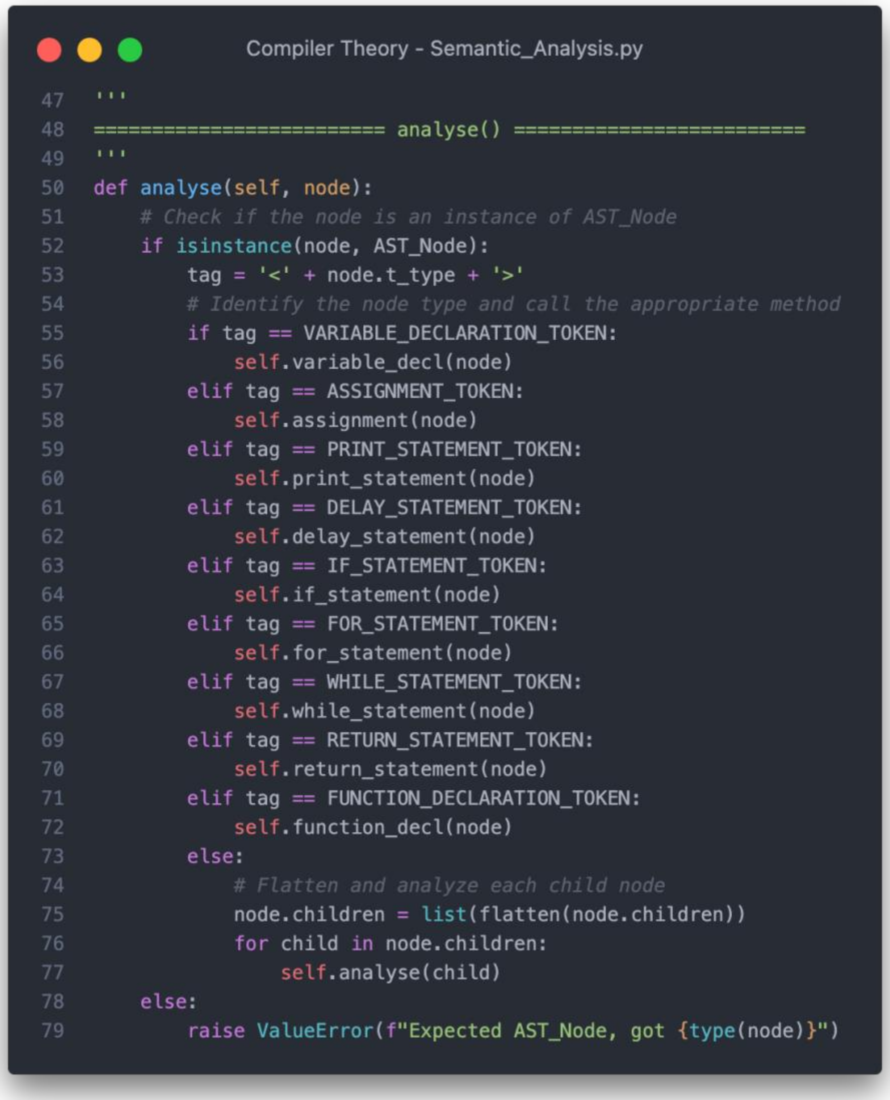


```python
'''
========================= program() =========================
'''
def program(self, program_node):
    # Check if the program node has children
    if not hasattr(program_node, 'children'):
        raise ValueError("program_node does not have 'children' attribute")

    # Flatten the children of the program node
    program_node.children = list(flatten(program_node.children))

    # Analyze each child node
    for node in program_node.children:
        self.analyse(node)

    # Print success message
    print("\033[1;32mSemantic Analysis successful!\033[0m")
```

Figure 13: Program Method

Analyse Method

Every node in the AST is subjected to the analyse method. It determines the type of node and invokes the appropriate method to perform specific semantic checks. If the node is not an instance of AST_Node, it returns a ValueError. It flattens the children in nodes with children and recursively calls analyse on each child.


Compiler Theory - Semantic_Analysis.py

```python
'''
========================= analyse() =========================
'''
def analyse(self, node):
    # Check if the node is an instance of AST_Node
    if isinstance(node, AST_Node):
        tag = '<' + node.t_type + '>'
        # Identify the node type and call the appropriate method
        if tag == VARIABLE_DECLARATION_TOKEN:
            self.variable_decl(node)
        elif tag == ASSIGNMENT_TOKEN:
            self.assignment(node)
        elif tag == PRINT_STATEMENT_TOKEN:
            self.print_statement(node)
        elif tag == DELAY_STATEMENT_TOKEN:
            self.delay_statement(node)
        elif tag == IF_STATEMENT_TOKEN:
            self.if_statement(node)
        elif tag == FOR_STATEMENT_TOKEN:
            self.for_statement(node)
        elif tag == WHILE_STATEMENT_TOKEN:
            self.while_statement(node)
        elif tag == RETURN_STATEMENT_TOKEN:
            self.return_statement(node)
        elif tag == FUNCTION_DECLARATION_TOKEN:
            self.function_decl(node)
        else:
            # Flatten and analyze each child node
            node.children = list(flatten(node.children))
            for child in node.children:
                self.analyse(child)
    else:
        raise ValueError(f"Expected AST_Node, got {type(node)}")
```

Figure 14: Analyse Method

Overview of Other Methods and Functions

Stack Get and Set Methods

The stack_get method looks for a variable in either the current or all scopes, depending on whether the current_scope_only flag is set. If the variable is found, it returns the value; otherwise, it returns False. The stack_set method initialises a variable in the current scope with the specified value, ensuring that the variable is properly stored in the stack.

Variable Declaration Method

The variable_decl method handles variable declarations. It checks for type mismatches and ensures that the variable is not re-declared within the current scope. If the declaration is correct, it adds the variable to the current scope along with its type and value.

Assignment Method

The assignment method handles variable assignments. Before assigning a value, it checks to see if the variable exists and that the type of the variable matches the assigned value. If the assignment is valid, it updates the variable's value on the stack.

Print Statement Method

The print_statement method evaluates the expression to be printed and returns the result. This method ensures that the expression is evaluated correctly and yields the expected result.

Delay Statement Method

The delay_statement method evaluates the expression for the delay duration. This method ensures that the expression is correctly evaluated, but it causes no actual delay in the semantic analysis phase.

If Statement Method

The if_statement method determines the condition of an if statement. Based on the value of the condition, it examines the appropriate block of statements (either the if block or the else block, if present).

For Statement Method

The for_statement method handles for loops. It creates a new scope for the loop, handles the initialization, evaluates the condition, and analyses the loop body. It continues to loop while the condition is true, updating the loop variable with the assignment expression.

While Statement Method

The while_statement method handles while loops. It creates a new scope for the loop, evaluates the loop condition, and examines the loop body while the condition holds true. After the loop, it leaves the scope.

Return Statement Method

The return_statement method evaluates the return expression, exits the current scope, and saves the return value. It ensures that the function returns the appropriate type of value.

Function Declaration Method

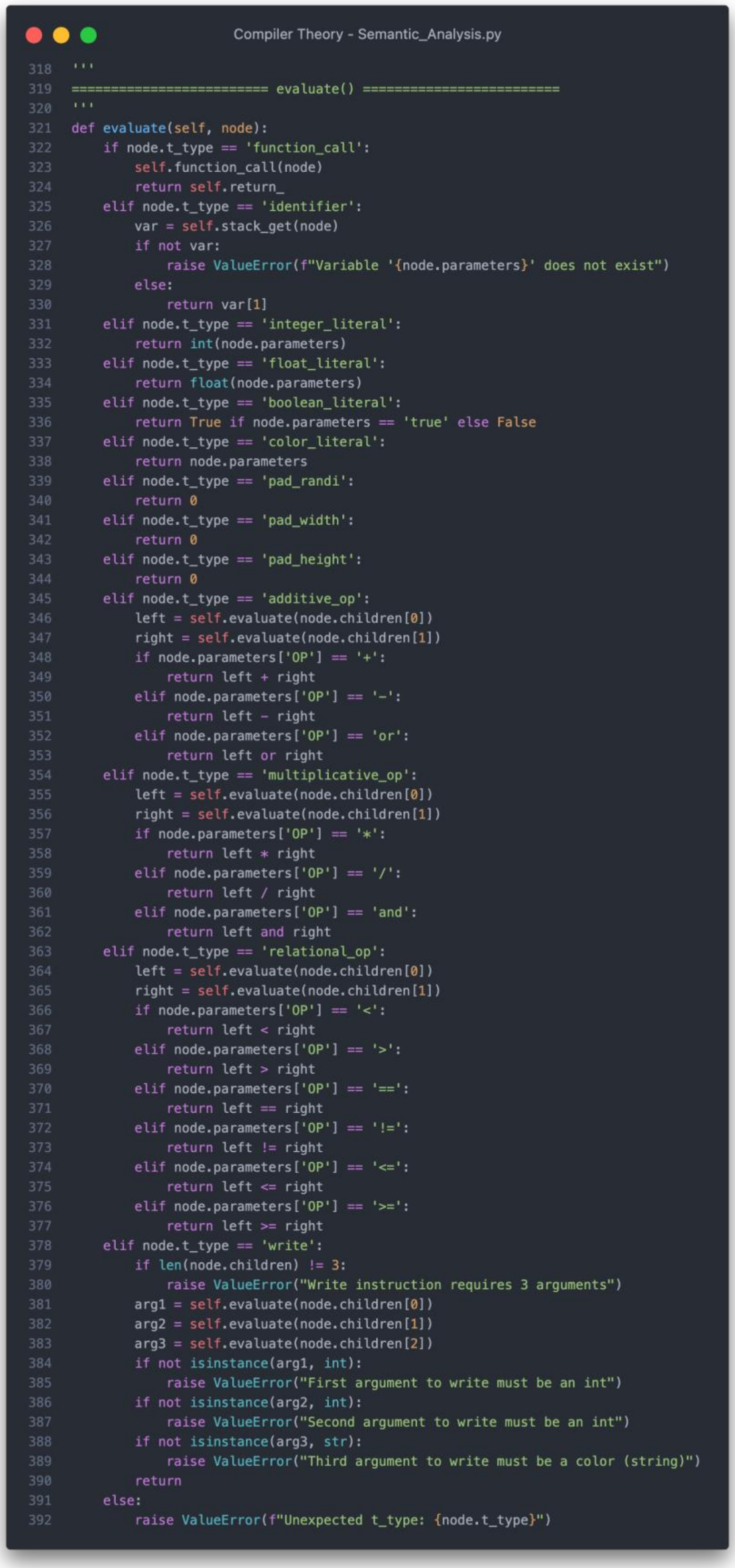
The function_decl method handles function declarations. It checks for function re-declarations and creates a new scope for the function parameters and body. It ensures that parameters are properly added to the function scope and that the function has a valid return statement with the appropriate return type.

Function Call Method

The function_call method handles function calls. It determines whether the function exists and ensures that the arguments match the function's parameters in both type and number. It defines a new scope for the function call, evaluates the arguments, and examines the function body.

Evaluate Method

Finally, the evaluate method is a helper function for evaluating expressions. It handles a variety of nodes, including function calls, identifiers, literals, and operations. It evaluates expressions recursively, ensuring that the results are correct and consistent with the language's semantic rules.



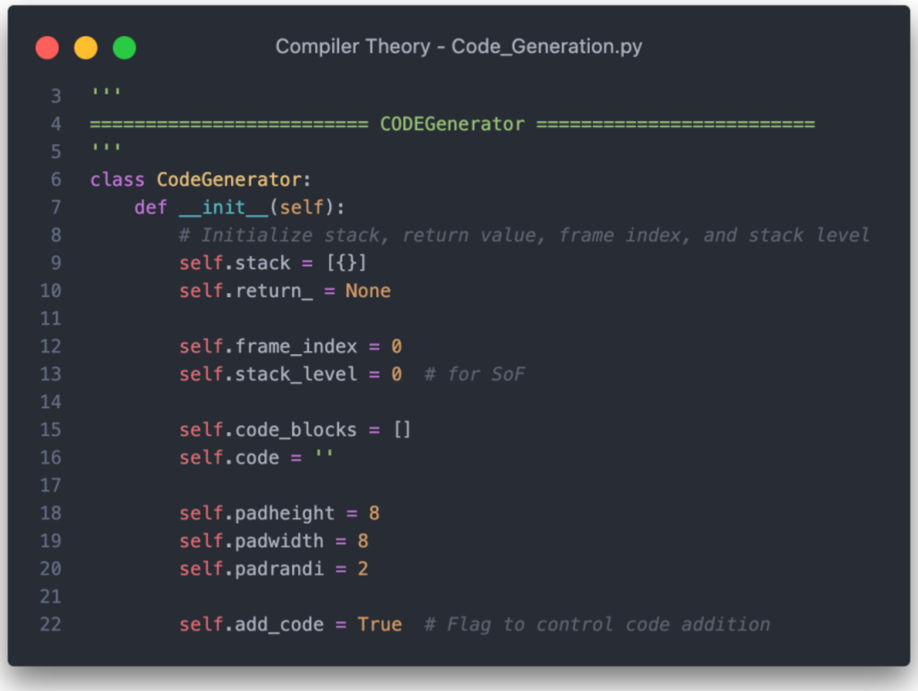
```python
'''
======================== evaluate() ========================
'''
def evaluate(self, node):
    if node.t_type == 'function_call':
        self.function_call(node)
        return self.return_
    elif node.t_type == 'identifier':
        var = self.stack_get(node)
        if not var:
            raise ValueError(f"Variable '{node.parameters}' does not exist")
        else:
            return var[1]
    elif node.t_type == 'integer_literal':
        return int(node.parameters)
    elif node.t_type == 'float_literal':
        return float(node.parameters)
    elif node.t_type == 'boolean_literal':
        return True if node.parameters == 'true' else False
    elif node.t_type == 'color_literal':
        return node.parameters
    elif node.t_type == 'pad_randi':
        return 0
    elif node.t_type == 'pad_width':
        return 0
    elif node.t_type == 'pad_height':
        return 0
    elif node.t_type == 'additive_op':
        left = self.evaluate(node.children[0])
        right = self.evaluate(node.children[1])
        if node.parameters['OP'] == '+':
            return left + right
        elif node.parameters['OP'] == '-':
            return left - right
        elif node.parameters['OP'] == 'or':
            return left or right
    elif node.t_type == 'multiplicative_op':
        left = self.evaluate(node.children[0])
        right = self.evaluate(node.children[1])
        if node.parameters['OP'] == '*':
            return left * right
        elif node.parameters['OP'] == '/':
            return left / right
        elif node.parameters['OP'] == 'and':
            return left and right
    elif node.t_type == 'relational_op':
        left = self.evaluate(node.children[0])
        right = self.evaluate(node.children[1])
        if node.parameters['OP'] == '<':
            return left < right
        elif node.parameters['OP'] == '>':
            return left > right
        elif node.parameters['OP'] == '==':
            return left == right
        elif node.parameters['OP'] == '!=':
            return left != right
        elif node.parameters['OP'] == '<=':
            return left <= right
        elif node.parameters['OP'] == '>=':
            return left >= right
    elif node.t_type == 'write':
        if len(node.children) != 3:
            raise ValueError("Write instruction requires 3 arguments")
        arg1 = self.evaluate(node.children[0])
        arg2 = self.evaluate(node.children[1])
        arg3 = self.evaluate(node.children[2])
        if not isinstance(arg1, int):
            raise ValueError("First argument to write must be an int")
        if not isinstance(arg2, int):
            raise ValueError("Second argument to write must be an int")
        if not isinstance(arg3, str):
            raise ValueError("Third argument to write must be a color (string)")
        return
    else:
        raise ValueError(f"Unexpected t_type: {node.t_type}")
```

Figure 15: Evaluate Method

Task 4 - PArIR Code Generation Pass - No Arrays

Code Generation is a critical step in the compiler design process that converts the intermediate representation of source code, known as an AST, into executable code. This phase converts high-level constructs to lower-level code that a virtual machine can understand and execute. The generated code may take the form of assembly language, machine code, or in this case a virtual machine intermediate code. Code generation ensures that high-level programme semantics are preserved and correctly implemented in executable code, allowing for the program's final execution.

The **CodeGenerator class** starts with a stack that manages variable scopes, frame indices, and stack levels. It also includes code blocks and a flag that controls whether or not code is added.

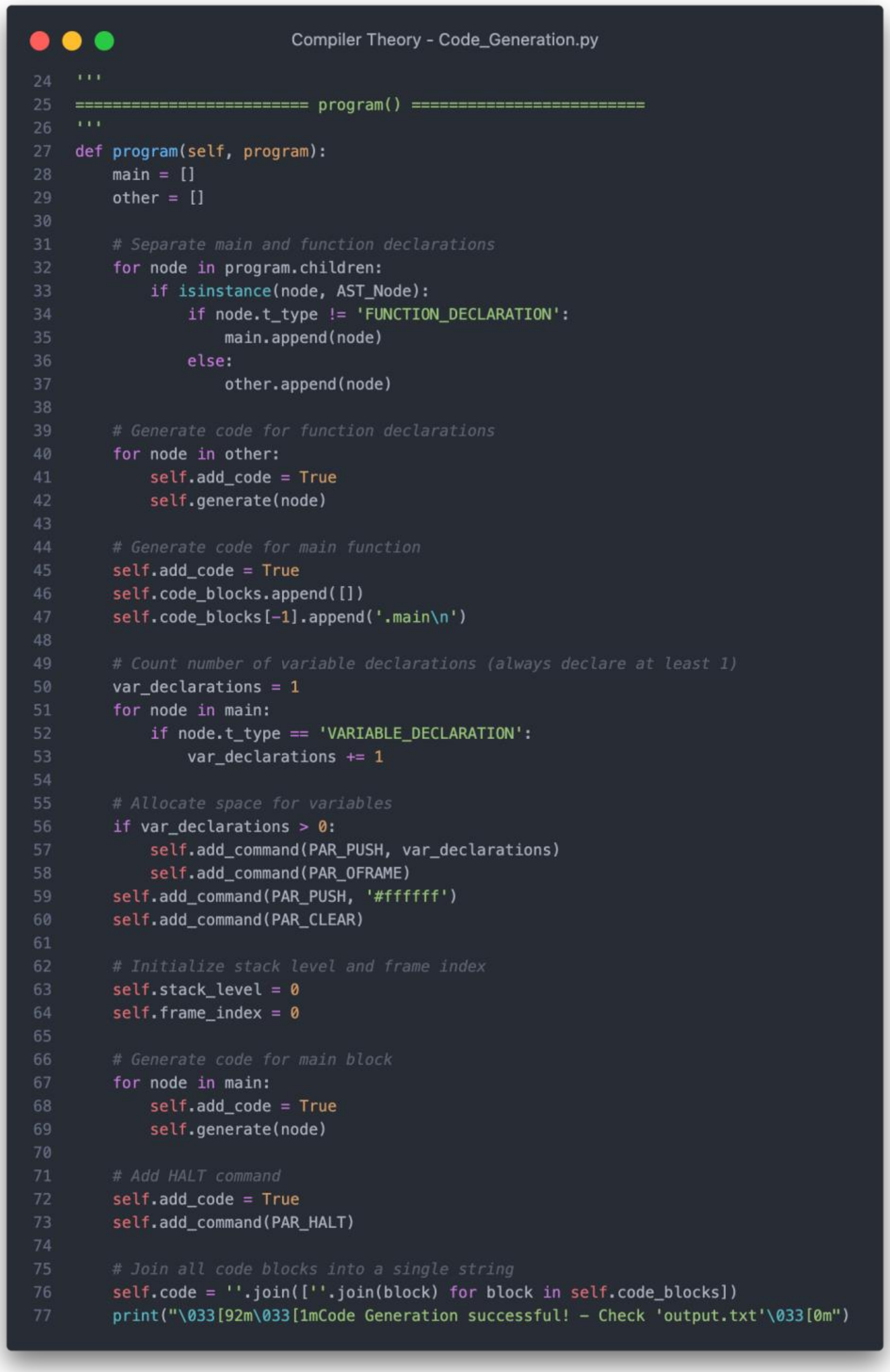


```python
'''
======================== CODEGenerator ========================
'''
class CodeGenerator:
    def __init__(self):
        # Initialize stack, return value, frame index, and stack level
        self.stack = [{}]
        self.return_ = None

        self.frame_index = 0
        self.stack_level = 0  # for SoF

        self.code_blocks = []
        self.code = ''

        self.padheight = 8
        self.padwidth = 8
        self.padrandi = 2

        self.add_code = True  # Flag to control code addition
```

Figure 16: Code Generator Class

Program Method

This method is, once again, the starting point for code generation. It separates the main function from the other function declarations, generates code for each function, and finally generates code for the main function. It also handles variable declarations and adds the commands required for programme initialization.

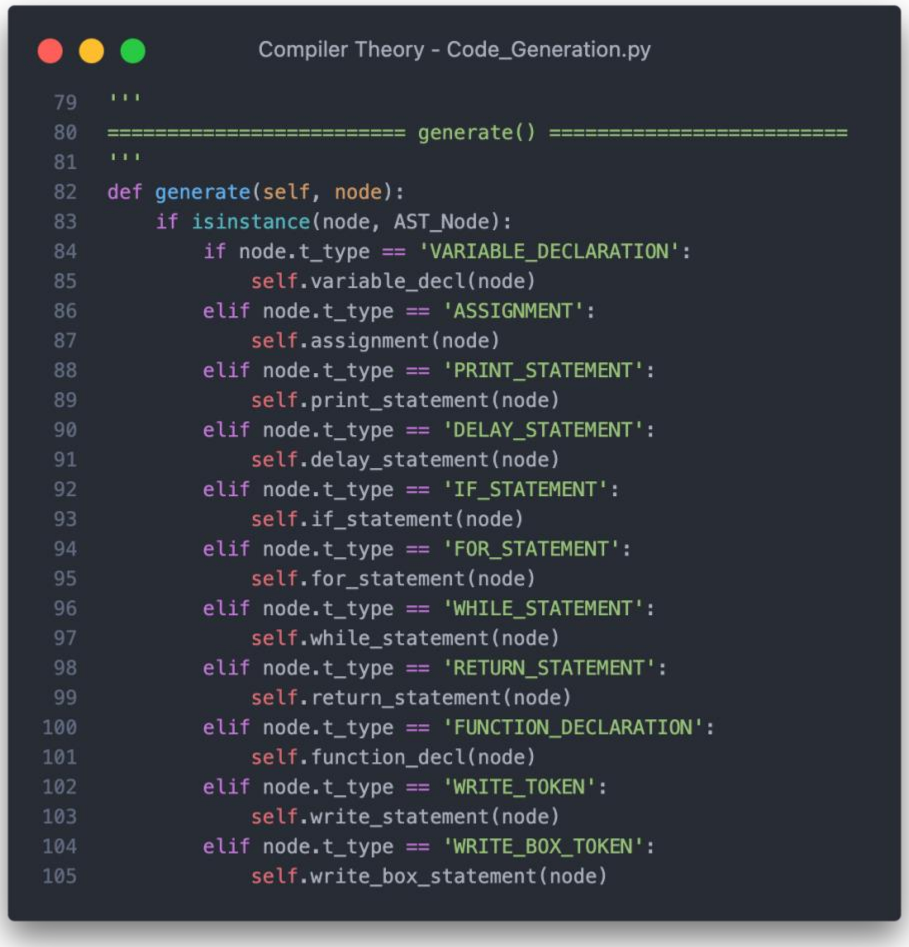

Compiler Theory - Code_Generation.py

```python
'''
========================= program() =========================
'''
def program(self, program):
    main = []
    other = []

    # Separate main and function declarations
    for node in program.children:
        if isinstance(node, AST_Node):
            if node.t_type != 'FUNCTION_DECLARATION':
                main.append(node)
            else:
                other.append(node)

    # Generate code for function declarations
    for node in other:
        self.add_code = True
        self.generate(node)

    # Generate code for main function
    self.add_code = True
    self.code_blocks.append([])
    self.code_blocks[-1].append('.main\n')

    # Count number of variable declarations (always declare at least 1)
    var_declarations = 1
    for node in main:
        if node.t_type == 'VARIABLE_DECLARATION':
            var_declarations += 1

    # Allocate space for variables
    if var_declarations > 0:
        self.add_command(PAR_PUSH, var_declarations)
        self.add_command(PAR_OFRAME)
    self.add_command(PAR_PUSH, '#ffffff')
    self.add_command(PAR_CLEAR)

    # Initialize stack level and frame index
    self.stack_level = 0
    self.frame_index = 0

    # Generate code for main block
    for node in main:
        self.add_code = True
        self.generate(node)

    # Add HALT command
    self.add_code = True
    self.add_command(PAR_HALT)

    # Join all code blocks into a single string
    self.code = ''.join([''.join(block) for block in self.code_blocks])
    print("\033[92m\033[1mCode Generation successful! - Check 'output.txt'\033[0m")
```

Figure 17: Program Method

Generate Method

The generate method generates code for different types of nodes in the AST. It recognises the node type and invokes the appropriate method to generate specific code. This method serves as a dispatcher, ensuring that all node types are handled correctly.


Compiler Theory - Code_Generation.py

```python
'''
======================== generate() ========================
'''
def generate(self, node):
    if isinstance(node, AST_Node):
        if node.t_type == 'VARIABLE_DECLARATION':
            self.variable_decl(node)
        elif node.t_type == 'ASSIGNMENT':
            self.assignment(node)
        elif node.t_type == 'PRINT_STATEMENT':
            self.print_statement(node)
        elif node.t_type == 'DELAY_STATEMENT':
            self.delay_statement(node)
        elif node.t_type == 'IF_STATEMENT':
            self.if_statement(node)
        elif node.t_type == 'FOR_STATEMENT':
            self.for_statement(node)
        elif node.t_type == 'WHILE_STATEMENT':
            self.while_statement(node)
        elif node.t_type == 'RETURN_STATEMENT':
            self.return_statement(node)
        elif node.t_type == 'FUNCTION_DECLARATION':
            self.function_decl(node)
        elif node.t_type == 'WRITE_TOKEN':
            self.write_statement(node)
        elif node.t_type == 'WRITE_BOX_TOKEN':
            self.write_box_statement(node)
```

Figure 18: Generate Method

Overview of Other Methods and Functions

Stack Get and Set Methods

The stack_get method searches the stack frames for a variable and returns it. The stack_set method initialises a variable in the stack, updates its value, and manages its scope within stack frames.

Add Command Method

The add_command method inserts a command into the current code block. It handles a variety of operators and operands, ensuring that each command uses the correct syntax. This method ensures that the generated code is properly formatted and contains the necessary instructions.

Variable Declaration Method

The variable_decl method handles variable declarations. It evaluates the expression for the variable's initial value, assigns the variable to the stack, and adds commands to push and store the value.

Assignment Method

The assignment method handles variable assignments. It evaluates the expression for the new value, gets the variable's memory location from the stack, and adds commands to push the value and update the variable in memory.

Print Statement Method

The print_statement method evaluates the expression to be printed and includes commands for pushing the value to the stack and printing it. This method ensures that print statements in the source code are converted to executable print instructions.

Delay Statement Method

The delay_statement method evaluates the delay duration expression and includes commands for pushing the value to the stack and executing the delay. This method ensures that any delay statements in the source code are properly implemented in the generated code.

If Statement Method

The if_statement method creates code for if statements. It evaluates the condition, adds conditional jump commands, and generates code for true and false blocks (if they exist). It manages stack frames and inserts jump commands as needed to ensure that execution proceeds correctly.

Add and Remove Stack Frame Methods

The add_stack_frame method creates a new stack frame to handle variable scopes. On the contrary, the remove_stack_frame method removes the current stack frame, updates variable scopes, and adds the commands required to close the frame.

For Statement Method

The for_statement method generates code for for-loops. It handles initialization, condition evaluation, loop body, and increment expressions, as well as the commands required to manage the loop's control flow and variable scopes.

While Statement Method

Similarly to the above, the while_statement method creates code for while loops. It evaluates the condition, generates code for the loop body, and adds commands to manage the loop's control flow, ensuring that the loop runs correctly while the condition is met.

Return Statement Method

The return_statement method evaluates the return expression and adds commands for pushing the return value to the stack and executing the return instruction. This method ensures that functions return the exact values specified in the source code.

Function Declaration Method

The function_decl method creates code for function declarations. It manages stack frames, parameters, and function bodies, including commands for allocating variable space and ensuring proper function execution.

Function Call Method

The function_call method creates code for function calls. It places arguments on the stack, adds commands to call the function, and manages the control flow for function calls, ensuring that functions are executed with the correct parameters and return values.

Write & Write_Box Statement Method

The write_statement method manages the __write statement. It evaluates the write expressions and adds commands to push the values to the stack and execute the write instruction. Moreover, the write_box_statement function evaluates the expressions for the write box operation and adds commands to push the values to the stack and perform the write box operation.

Code Evaluate Method

The code_evaluate method evaluates expressions from the AST. It handles a variety of nodes, including identifiers, literals, and operations, as well as the necessary commands for evaluating and pushing values onto the stack. This method ensures that expressions are accurately translated into executable instructions.

Evaluate Method

Like the previous method, the evaluate method is a helper function for evaluating expressions. It manages function calls, literals, and identifiers, ensuring that values are correctly retrieved and used in generated code.

Task 5 - Arrays in ParL

In compiler theory & practice, allowing arrays in code generation adds a significant layer of functionality. Arrays are fundamental data structures that enable the storage and manipulation of multiple values under a single identifier. This capability is required for many programming tasks, including data collection, algorithm implementation, and efficient resource management. Integrating array support during the code generation phase entails converting high-level array operations into low-level instructions that a virtual machine or processor can execute.

The **ArrayCodeGenerator class extends the CodeGenerator class** by providing additional array-handling functionality. This extension expands on the foundational code generation tasks (Task 4) and adds new commands for array manipulation.

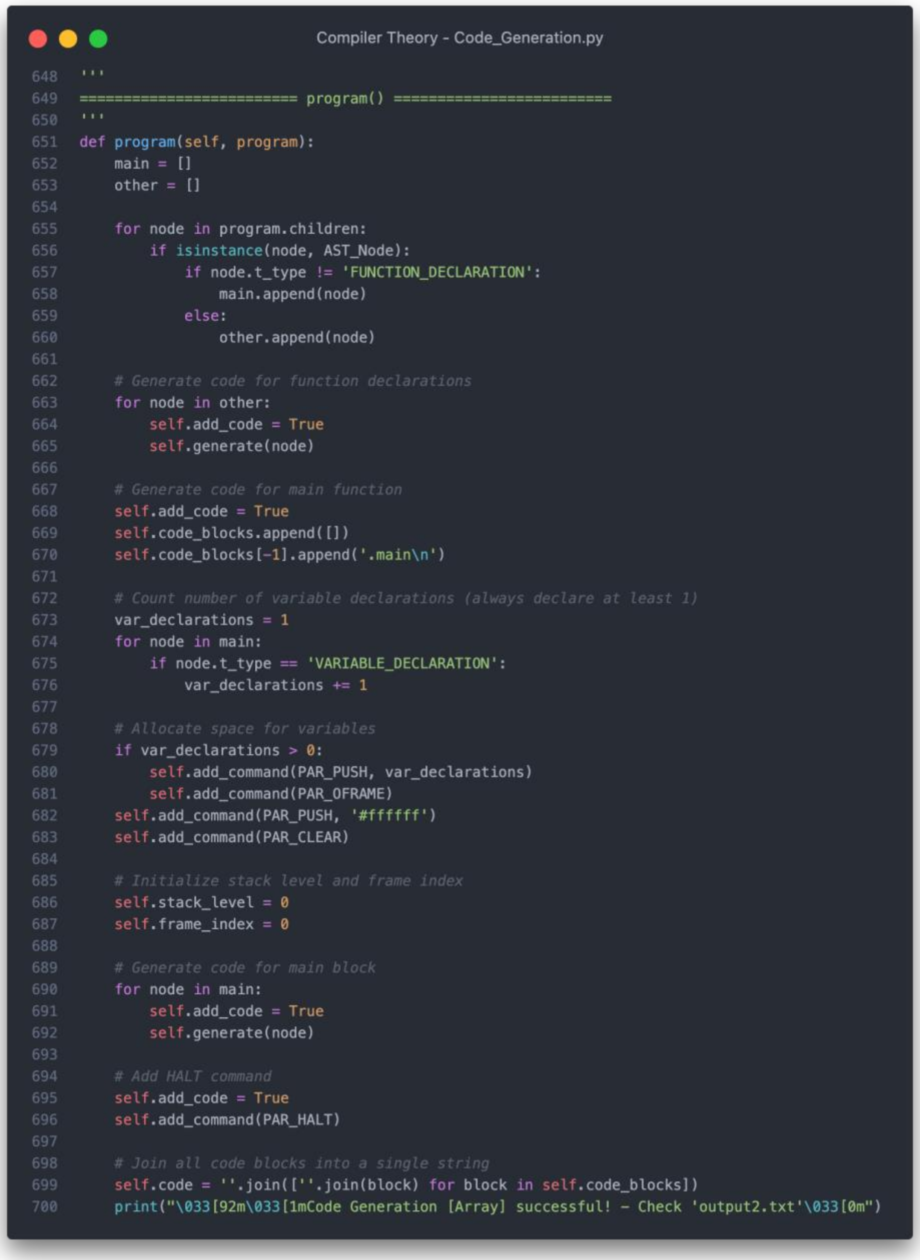
Overview of Methods

Array Code Generator Class

The ArrayCodeGenerator class extends the CodeGenerator class, inheriting its methods but adding array-specific functionality.

Programme Method

The programme method creates code for the entire programme, including function declarations and the main function. It decouples the main code from function declarations, generates code for each component, and includes initialization and halt commands. This method ensures that array-related commands are correctly included in the generated code.



```python
'''
========================= program() =========================
'''
def program(self, program):
    main = []
    other = []

    for node in program.children:
        if isinstance(node, AST_Node):
            if node.t_type != 'FUNCTION_DECLARATION':
                main.append(node)
            else:
                other.append(node)

    # Generate code for function declarations
    for node in other:
        self.add_code = True
        self.generate(node)

    # Generate code for main function
    self.add_code = True
    self.code_blocks.append([])
    self.code_blocks[-1].append('.main\n')

    # Count number of variable declarations (always declare at least 1)
    var_declarations = 1
    for node in main:
        if node.t_type == 'VARIABLE_DECLARATION':
            var_declarations += 1

    # Allocate space for variables
    if var_declarations > 0:
        self.add_command(PAR_PUSH, var_declarations)
        self.add_command(PAR_OFRAME)
    self.add_command(PAR_PUSH, '#ffffff')
    self.add_command(PAR_CLEAR)

    # Initialize stack level and frame index
    self.stack_level = 0
    self.frame_index = 0

    # Generate code for main block
    for node in main:
        self.add_code = True
        self.generate(node)

    # Add HALT command
    self.add_code = True
    self.add_command(PAR_HALT)

    # Join all code blocks into a single string
    self.code = ''.join([''.join(block) for block in self.code_blocks])
    print("\033[92m\033[1mCode Generation [Array] successful! - Check 'output2.txt'\033[0m")
```

Figure 19: Program Method

Generate Method

The generate method sends nodes to the appropriate handler methods based on their type. It calls array-specific node methods such as array_decl, array_assignment, and array_access. For other nodes, it invokes the superclass's generate method.

Array Declaration Method

The array_decl method handles array declarations. It determines the size of the array, allocates space for it, and initialises its elements. This method checks that arrays are properly declared and initialised in memory.

Array Assignment Method

The array_assignment method is responsible for assigning values to array elements. It evaluates the index and value expressions, determines the memory location, and stores the result in the array. This method ensures that array elements are properly assigned and stored in memory.

Array Access Method

The array_access method is used to access array elements. It evaluates the index expression, determines the memory location, and pushes the result to the stack. This method ensures that array elements are correctly accessed and returned.

Add Command Method

The add_command method adds commands to the current code block, including general and array-specific commands. It accurately formats the commands and incorporates them into the code being generated.

Code Evaluate Method

The code_evaluate method evaluates expressions, including those that contain arrays. It supports various node types, adds the commands required for evaluating and pushing values onto the stack, and ensures that array operations are correctly translated into executable instructions.

Testings & Results

Task 1 - Table-driven lexer - No Arrays

source.txt

```
// This func takes two
integers and returns true
if x > y else return
false

fun XGY (x:int, y:int) ->
bool{
    return (y > x);
   }
```

Terminal Output

```
Tokenizer (& Lexer) successful!
<fun> fun </fun>
<identifier> XGY </identifier>
<lparen> ( </lparen>
<identifier> x </identifier>
<colon> : </colon>
<type> int </type>
<comma> , </comma>
<identifier> y </identifier>
<colon> : </colon>
<type> int </type>
<rparen> ) </rparen
<rarrow> -> </rarrow>
<type> bool </type>
<lbrace> { </lbrace>
<return> return </return>
<lparen> ( </lparen>
<identifier> y </identifier>
<relational_op> > </relational_op>
<identifier> x </identifier>
<rparen> ) </rparen>
<semicolon> ; </semicolon>
<rbrace> } </rbrace>
```

The above test shows a source code file and the terminal output of the lexer. The source code defines the function XGY, which accepts two integers and returns a boolean indicating whether the first integer is greater than the second. The terminal output confirms successful tokenization by displaying tokens and various symbols that represent different parts of the source code. This output shows that the lexer can correctly identify and categorise the source code components, whilst ignoring the in line (single) comment.

```
let a:color = #000000000;
```

```
An error occurred: Invalid token:
"#000000000"
```

```
/* This
* is
* a Multi-line
* Comment */
let _x:int = 26;
```

```
An error occurred: Invalid token:
"_x"
```

```
let x:float = 26.26.26
```

```
An error occurred: Invalid token:
"26.26.26"
```

As seen from the above examples, the lexer handles errors correctly, stopping the process when invalid tokens are encountered.

Task 2 - Hand-crafted LL(k) parser - No Arrays

source.txt

```
fun XGY (x:int, y:int) -
> bool{
    return (y > x);
   }
```

Terminal Output

```
Tokenizer (& Lexer) successful!
Parser successful!

<program>
    <statement>
        <function_declaration>
<identifier>identifier</identifier>
            <formal_parameters>
                <formal_parameter>
<identifier>identifier</identifier>
<type>type</type>
                </formal_parameter>
                <formal_parameter>
<identifier>identifier</identifier>
<type>type</type>
                </formal_parameter>
            </formal_parameters>
            <type>type</type>
            <block>
                <statement>
<return_statement>
<expression>
<simple_expression>
                    <term>
<factor>
              <sub_expression>

<expression>
<relational_op>
<simple_expression>
<term>
<factor>
<identifier>identifier</identifier>
</factor>
                     </term>

</simple_expression>
<simple_expression>
<term>
<factor>
<identifier>identifier</identifier>
</factor>
                     </term>
</simple_expression>
</relational_op>
</expression>
</sub_expression>
```

```
                        </factor>
</term>
</simple_expression>
            </expression>
</return_statement>
                </statement>
            </block>
        </function_declaration>
    </statement>
</program>
```

The above code snippet shows the output of the hand-crafted LL(k=1) parser that parsed the same function XGY from the source code. The parser deconstructs the function into grammatical components, identifying and nesting elements like the function_declaration, formal_parameters, block, statement, return_statement, and various expressions. This detailed breakdown demonstrates the parser's ability to correctly interpret the code structure and generate a hierarchical representation that follows the programming language's syntax rules.

```
let number int = 5;
```

```
Tokenizer (& Lexer) successful!
An error occurred: Invalid syntax
at let number int = 5 ; ...
```

```
un AddNumbers (a:int,
b:int) -> int {
    let sum:int = a + b;
    return sum;
}
```

```
Tokenizer (& Lexer) successful!
An error occurred: Invalid syntax
at un AddNumbers ( a : int , b :
int ) -> int { let sum : int = a
...
```

```
if ( y <  ) {
    let t:bool = true;
}
```

```
Tokenizer (& Lexer) successful!
An error occurred: Invalid syntax
at if ( y < ) { let t : boolean =
true ; } ...
```

```
while ( > 2) {
    __delay 2;
}
```

```
Tokenizer (& Lexer) successful!
An error occurred: Invalid syntax
at while ( > 2 ) { __delay 2 ; }
...
```

As seen from the above examples, the hand-crafted parser handles errors correctly, stopping the process when invalid syntax is detected.

Task 3 - Semantic Analysis Pass - No Arrays

source.txt	Terminal Output
`fun MoreThan50 ( x : int ) -> bool { let x : int = 23; // syntax ok, but this should not be allowed !! if ( x <= 50 ) { return false; } return true; }`	`Tokenizer (& Lexer) successful!` `Parser successful!` `Variable x already exists`
`let x : int = 45; // this is fine while ( x < 50 ) { __print MoreThan50 ( x ); // "false" x6 since bool operator is < x = x + 1; }`	`Tokenizer (& Lexer) successful!` `Parser successful!` `False` `False` `False` `False` `False` `False`
`// This func takes two integers and returns true if x > y else return false fun XGY (x:int, y:int) -> bool { return (y > x); }`	`Tokenizer (& Lexer) successful!` `Parser successful!` `Semantic Analysis successful!`

The above depict the semantic analysis pass for three different code snippets. In the first snippet, the analyzer detects a semantic error, indicating that the variable x is redeclared within the MoreThan50 function, which is not permitted. The second snippet demonstrates a valid declaration and use of x in a while loop and a print statement, but the function call produces multiple False outputs due to the condition being checked. The third snippet depicts the function XGY, which is successfully analysed without errors, demonstrating that it follows both syntactical and semantic rules. This demonstrates the semantic analyzer's ability to enforce variable scope rules and ensure logical correctness in code.

Task 4 - PArIR Code Generation Pass - No Arrays

Example 1:

source.txt

```
for ( let i : int = 0; i <
10; i = i + 1 ) {
    __print i;
    __delay 100;

}
```

output.txt

```
.main
push 1
oframe
push #ffffff
clear
push 1
oframe
push 0
push 0
push 0
st
push 10
push [0:0]
lt
push 0
eq
push #PC+14
cjmp2
push [0:0]
print
push 100
delay
push 1
push [0:0]
add
push 0
push 0
st
push #PC-17
jmp
cframe
halt
```

The above shows the code generation pass for a simple for loop in the PArIR programming language, which converts high-level source code into a lower-level intermediate representation (IR). The source code defines a loop that starts with i = 0, increments it by 1 in each iteration, and runs while i < 10. Within the loop, it prints i and waits for 100 units.

The output.txt file contains the corresponding PArIR instructions. It starts by defining the main function and initialising the stack frame. The loop initialization pushes 0 onto the stack as the initial value of i, followed by the loop condition check (i 10). If the condition is met, it prints i, waits 100 units, increments i by one, and checks the loop condition again. If the condition is false, it exits the loop and stops the execution. This output shows how high-level loop constructs are translated into low-level instructions that perform the same operations.

Example 2:

source.txt

```
let c : colour = 0 as colour;
for ( let i : int = 0; i <
64; i = i + 1 ) {
    c = ( random int 1677216
) as colour;
    __clear c;
    __delay 16;
}
```

output.txt

```
..main
push 4
jmp
halt
push 1
oframe
push 0
push 0
push 0
st
push 1
oframe
push 0
push 0
push 0
st
push 64
push [0:0]
lt
push #PC+4
cjmp
push #PC+22
jmp
push 0
oframe
push 1677216
irnd
push 0
push 2
st
push [0:2]
clear
push 16
delay
cframe
push 1
push [0:0]
add
push 0
push 0
st
push #PC-25
jmp
cframe
cframe
halt
```

Similarly, this code snippet shows the code generation pass for a more complex for loop that initialises a colour variable c, runs it 64 times, assigns a random integer to c with each iteration, clears the colour, and delays for 16 units. The corresponding PArIR code shows the setup for the main function, loop initialization, condition checking, random number assignment, colour clearing, and delay instructions, which demonstrates the translation of high-level loop logic into low-level instructions.

Task 5 - Arrays in PArL

source.txt

```
// x is an array of 16 +ve
integers
fun MaxInArray ( x : int [ 8
] ) -> int {
    let m : int = 0;
    for ( let i : int = 0; i
< 8; i = i + 1 ) {
        if ( x [ i ] > m ) {
m = x [ i ]; }
    }
    return m;
}

let list_of_integers : int [
] = [ 23, 54, 3, 65, 99, 120,
34, 21 ];
let max : int = MaxInArray (
list_of_integers );
__print max;
```

output2.txt

```
.main
push 4
jmp
halt
push 10
oframe
push #PC+52
jmp
.MaxInArray
push 9
alloc
push 0
push 8
push 0
st
push 1
oframe
push 0
push 0
push 0
st
push 8
push [0:0]
lt
push #PC+4
cjmp
push #PC+29
jmp
push 0
oframe
push [8:2]
push [0:1]
push +[0:2]
gt
push #PC+4
cjmp
push #PC+10
jmp
push 0
oframe
push [0:2]
push +[0:3]
push 8
push 3
st
cframe
cframe
```

```
push 1
push [0:0]
add
push 0
push 0
st
push #PC-32
jmp
cframe
push [8:0]
ret
push 21
push 34
push 120
push 99
push 65
push 3
push 54
push 23
push 8
push 1
push 0
sta
push 8
pusha [1:0]
push 8
push .MaxInArray
call
push 9
push 0
st
push [9:0]
print
cframe
halt
```

This example shows the code generation pass for **handling arrays** in PArIR. The source code includes a function called MaxInArray that finds the highest value in an array of eight integers. The corresponding PArIR instructions demonstrate the translation of high-level constructs to low-level operations. The code begins by defining the main function, initialising the required variables, and allocating memory for the array. It then enters a loop in which it initialises the index and compares each array element to determine the maximum value. The loop contains instructions for incrementing the index, conditional jumping based on comparisons, and updating the maximum value. After processing all of the elements, the function returns the maximum value, which is then displayed in the main program. This demonstrates how complex operations such as loops, conditionals, and array handling are systematically converted into sequential low-level instructions that manage memory, perform comparisons, and regulate program execution flow.