CPS2000 - Compiler Theory and Practice

Assignment Report

Task 1: Table-driven Lexer Implementation

Introduction

The goal of Task 1 is to develop a table-driven lexer for the PArL language. This lexer is responsible for tokenizing the source code into meaningful tokens that the parser can process. The lexer is designed using a Deterministic Finite Automaton (DFA) approach, where the transition table simulates the DFA's transition function. This report details the implementation process of the lexer, which extends the one provided on the VLE.

Lexer Implementation

The lexer classifies the input source code into various token types using a state transition table. The primary components of the lexer are:

- **Token Types**: An enumeration defining the various types of tokens the lexer can recognize.
- **Lexer Class**: Contains methods for initializing the transition table, categorizing characters, and generating tokens.

The lexer implementation is defined in the `lexer.py` file.

Token Types

We define an enumeration `TokenType` to represent different kinds of tokens in the PArL language:

```
from enum import Enum

class TokenType(Enum):
    IDENTIFIER = "IDENTIFIER"
    LITERAL = "LITERAL"
    WHITESPACE = "WHITESPACE"
    OPERATOR = "OPERATOR"
    DELIMITER = "DELIMITER"
    VOID = "VOID"
    END = "END"
    SPECIAL_FUNCTION = "SPECIAL_FUNCTION"
    KEYWORD = "KEYWORD"
    ARROW = "ARROW"
    LEXICAL_ERROR = "LEXICAL_ERROR"
```

Lexer Class

The `Lexer` class is responsible for reading the source code, processing it character by character, and generating tokens based on the DFA's state transitions.

Initialization

The lexer initializes with a list of lexemes and states. It also sets up a 2D transition table `Tx` to simulate the DFA:

```
class Lexer:
    def __init__(self):
        self.lexeme_list = ["_", "letter", "digit", "ws", "eq", "sc",
"other", "op", "delim", "dot", "hash", "gt", "minus"]
        self.states_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
        self.states_accp = [1, 2, 3, 4, 5, 6, 7, 8, 10, 12]

        self.rows = len(self.states_list)
        self.cols = len(self.lexeme_list)

        # Let's take integer -1 to represent the error state for this DFA
        self.Tx = [[-1 for j in range(self.cols)] for i in
range(self.rows)]
        self.InitialiseTxTable()
```

Transition Table Initialization

The `InitialiseTxTable` method sets up the DFA transition table based on the lexical rules of the PArL language. This method defines how the lexer transitions between states when different types of characters are encountered:

```
    def InitialiseTxTable(self):
        # Update Tx to represent the state transition function of the DFA

        # Variables and Identifiers
        # Starting with a letter or underscore leads to state 1
(identifier)
        self.Tx[0][self.lexeme_list.index("letter")] = 1
        self.Tx[0][self.lexeme_list.index("_")] = 1
        # Continuing with letters, digits, or underscores keeps in state 1
        self.Tx[1][self.lexeme_list.index("letter")] = 1
        self.Tx[1][self.lexeme_list.index("digit")] = 1
        self.Tx[1][self.lexeme_list.index("_")] = 1

        # White Space
        # White space transitions to state 2
        self.Tx[0][self.lexeme_list.index("ws")] = 2
        self.Tx[2][self.lexeme_list.index("ws")] = 2

        # Equal sign (=)
        # Equals sign transitions to state 3 (operator)
```

```
        self.Tx[0][self.lexeme_list.index("eq")] = 3

        # Integers and Literals
        # Starting with a digit transitions to state 4 (integer literal)
        self.Tx[0][self.lexeme_list.index("digit")] = 4
        self.Tx[4][self.lexeme_list.index("digit")] = 4
        # A dot after a digit transitions to state 8 (floating-point
literal)
        self.Tx[4][self.lexeme_list.index("dot")] = 8
        self.Tx[8][self.lexeme_list.index("digit")] = 8

        # Semicolon sign (;)
        # Semicolon transitions to state 5 (delimiter)
        self.Tx[0][self.lexeme_list.index("sc")] = 5

        # Operators
        # Operators transition to state 6
        self.Tx[0][self.lexeme_list.index("op")] = 6
        # Special case for operators like >=, <=, ==, !=
        self.Tx[6][self.lexeme_list.index("eq")] = 6
        # Greater than symbol transitions to state 6
        self.Tx[0][self.lexeme_list.index("gt")] = 6

        # Delimiters
        # Delimiters transition to state 7
        self.Tx[0][self.lexeme_list.index("delim")] = 7

        # Hexadecimal Literals (Colors)
        # Hash transitions to state 9
        self.Tx[0][self.lexeme_list.index("hash")] = 9
        # Hex digits or letters transition to state 10
        self.Tx[9][self.lexeme_list.index("digit")] = 10
        self.Tx[9][self.lexeme_list.index("letter")] = 10
        self.Tx[10][self.lexeme_list.index("digit")] = 10
        self.Tx[10][self.lexeme_list.index("letter")] = 10

        # Arrow (->)
        # Minus transitions to state 11
        self.Tx[0][self.lexeme_list.index("minus")] = 11
        # Greater than symbol after minus transitions to state 12 (arrow)
        self.Tx[11][self.lexeme_list.index("gt")] = 12
```

This initialization method defines the transitions for various token types, enabling the lexer to handle identifiers, literals, operators, delimiters, and other token categories based on the defined DFA.

Character Categorization

The `CatChar` method categorizes characters based on their type. This categorization is crucial for determining the transitions in the DFA:

```
    def CatChar(self, character):
        cat = "other"
        if character.isalpha(): cat = "letter"
        if character.isdigit(): cat = "digit"
        if character == "_": cat = "_"
        if character.isspace(): cat = "ws"
        if character == ";": cat = "sc"
        if character == "=": cat = "eq"
        if character in "+*/<>!": cat = "op"
        if character == ">": cat = "gt"
        if character == "-": cat = "minus"
        if character in "{}()[],;:": cat = "delim"
        if character == ".": cat = "dot"
        if character == "#": cat = "hash"
        return cat
```

Lexeme Classification

The lexer uses specific patterns to classify lexemes into different token types. The following descriptions detail how each type of lexeme is recognized:

- **Keywords**: Recognized as reserved words in the language. Examples include `fun`, `let`, `return`, `if`, `else`, `for`, `while`, `as`, `int`, `float`, `bool`, and `colour`.
- **Operators**: Include arithmetic and logical operators such as `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `and`, `or`, `not`, and `=`.
- **Delimiters**: Characters that separate tokens, such as `{`, `}`, `(`, `)`, `[`, `]`, `,`, `;`, and `:`.
- **Arrow**: The `->` symbol used in function return types.
- **Literals**: Include integer and floating-point numbers, boolean literals (`true` and `false`), and hexadecimal color codes (`#000000` to `#ffffff`).
- **Identifiers**: Names used for variables, functions, etc. They start with a letter or underscore and can contain letters, digits, and underscores.
- **Special Functions**: Recognized as built-in functions like `__print`, `__delay`, `__write`, `__write_box`, `__random_int`, `__width`, `__height`, and `__read`.
- **Whitespace**: Spaces, tabs, and newlines that separate tokens but are not meaningful by themselves.

These classifications are based on patterns typically expressed in regular expressions but implemented through the DFA transitions in the lexer.

Token Generation

The `NextToken` method processes the input string and generates tokens based on the DFA states. Here's how the method works:

1. **Initialization**:
    - `state`: Set to 0, representing the initial state.
    - `stack`: Used to keep track of states.
    - `lexeme`: Stores the current lexeme being processed.
    - `start_idx`: Marks the starting index of the lexeme in the source code.

- `stack.append(-2)`: Inserts the error state at the bottom

of the stack.

2. **Processing Loop**:
    - The loop runs while the state is not -1 (error state).
    - If the current state is an accepting state, the stack is cleared.
    - The current state is pushed onto the stack.
    - The next character is read using the `NextChar` method.
    - If the end of the input is reached, the loop breaks.
    - The character is appended to the lexeme, and the source index is incremented.
    - The character category is determined using the `CatChar` method.
    - The state is updated based on the transition table (`Tx`).
3. **Error Handling**:
    - If an error state is encountered, the last character is removed from the lexeme.
    - The stack is processed to handle syntax errors and determine the final state.
4. **Final State Handling**:
    - The final state is checked against the stack to determine the correct token type.
    - The token is returned along with the lexeme.

```
def NextToken(self, src_program_str, src_program_idx):
    state = 0  # initial state is 0 - check Tx
    stack = []
    lexeme = ""
    start_idx = src_program_idx
    stack.append(-2)  # insert the error state at the bottom of the stack.

    while state != -1:
        if self.AcceptingStates(state):
            stack.clear()
        stack.append(state)

        exists, character = self.NextChar(src_program_str,
src_program_idx)
        if not exists:
            break  # Break out of loop if we're at the end of the string

        lexeme += character
        src_program_idx += 1

        cat = self.CatChar(character)
        if cat not in self.lexeme_list:
            state = -1
        else:
            state = self.Tx[state][self.lexeme_list.index(cat)]

    if lexeme:
```

```
            lexeme = lexeme[:-1]  # remove the last character added which sent
the lexer to state -1

        syntax_error = False
        # rollback
        while len(stack) > 0:
            if stack[-1] == -2:  # report a syntax error
                syntax_error = True
                break

            # Pop this state if not an accepting state.
            if not self.AcceptingStates(stack[-1]):
                stack.pop()
                if lexeme:
                    lexeme = lexeme[:-1]
                src_program_idx -= 1

            # This is an accepting state ... return it.
            else:
                state = stack.pop()
                break

        if syntax_error:
            # Continue collecting characters until whitespace or semicolon is
found
            while True:
                exists, character = self.NextChar(src_program_str,
src_program_idx)
                if not exists or character.isspace() or character == ";":
                    break
                lexeme += character
                src_program_idx += 1
            return (TokenType.LEXICAL_ERROR.value, lexeme), lexeme

        if self.AcceptingStates(state):
            return self.GetTokenTypeByFinalState(state, lexeme), lexeme
        else:
            return (TokenType.LEXICAL_ERROR.value, lexeme), lexeme
```

Generating Tokens for the Entire Program

The `GenerateTokens` method is responsible for iterating through the source code and generating a list of tokens. This method continuously calls the `NextToken` function to extract tokens until the end of the source code is reached.

Detailed Explanation of `GenerateTokens` Method:

1. **Initialization**:
    - `tokens_list`: An empty list to store the generated tokens.
    - `src_program_idx`: An index variable initialized to 0, which tracks the current position in the source code.

2. **Token Generation Loop**:
    - The method enters a `while` loop that continues as long as the current index (`src_program_idx`) is less than the length of the source code (`src_program_str`).
    - Inside the loop, the `NextToken` method is called with the source code string and the current index. This method returns a token and the corresponding lexeme.
    - If the generated token is not a whitespace (`TokenType.WHITESPACE`), it is appended to `tokens_list`. Whitespace tokens are ignored as they do not carry semantic meaning for the parser.
    - If a lexical error (`TokenType.LEXICAL_ERROR`) is encountered, the index is incremented by the length of the erroneous lexeme to skip over it. This allows the lexer to continue processing the remaining code.
    - Otherwise, the index is incremented by the length of the valid lexeme to move to the next part of the source code.
3. **End of Source Code Check**:
    - The loop includes a check to break explicitly if the end of the source code is reached (`src_program_idx >= (len(src_program_str) - 1)`), ensuring the lexer terminates correctly.
4. **Return Tokens List**:
    - After exiting the loop, the method returns `tokens_list`, which contains all the tokens generated from the source code.

```
def GenerateTokens(self, src_program_str):
    tokens_list = []
    src_program_idx = 0

    while src_program_idx < len(src_program_str):
        token, lexeme = self.NextToken(src_program_str, src_program_idx)
        if token[0] != TokenType.WHITESPACE.value:
            tokens_list.append(token)
        if token[0] == TokenType.LEXICAL_ERROR.value:
            src_program_idx += len(lexeme)  # Skip the erroneous lexeme
        else:
            src_program_idx += len(lexeme)

        if src_program_idx >= (len(src_program_str) - 1):
            break  # Explicitly break the loop if we've reached the end of
the input string

    return tokens_list
```

Conclusion

This lexer successfully tokenizes PArL programs by simulating a DFA using a table-driven approach. It handles identifiers, literals, operators, delimiters, and various other token types, ensuring the tokens are

correctly categorized and any lexical errors are reported.

Task 2: Hand-crafted LL(k) Parser Implementation

Introduction

The goal of Task 2 is to develop a hand-crafted LL(k) parser for the PArL language. This parser processes tokens generated by the lexer, producing an Abstract Syntax Tree (AST) that represents the syntactic structure of the source code. This report details the implementation process of the parser, including the definition of AST nodes and the parsing logic.

Abstract Syntax Tree (AST) Node Definitions

The AST nodes are defined in the `parser_nodes.py` file. Each class represents a different type of syntactic construct in the PArL language. Here's an overview of the node classes and their attributes:

Node Classes

The Abstract Syntax Tree (AST) is constructed using a set of node classes, each representing different constructs in the PArL language.

ProgramNode

The `ProgramNode` represents the root of the AST, encompassing the entire program. It contains a list of statements, which can be function declarations, variable declarations, or other statements.

```
class ProgramNode:
    def __init__(self, statements):
        self.statements = statements

    def __str__(self):
        return f"ProgramNode(statements={self.statements})"
```

- **Attributes**:
  - `statements`: A list of statements that make up the program.

FunctionDeclNode

The `FunctionDeclNode` represents a function declaration, including its name, parameters, return type, and body.

```
class FunctionDeclNode:
    def __init__(self, identifier, params, return_type, block):
        self.identifier = identifier
        self.params = params
        self.return_type = return_type
        self.block = block
```

```
    def __str__(self):
        return (f"FunctionDeclNode(identifier={self.identifier}, params=
{self.params}, "
                f"return_type={self.return_type}, block={self.block})")
```

- **Attributes**:
  - `identifier`: The name of the function.
  - `params`: A list of `ParamNode` objects representing the function parameters.
  - `return_type`: The return type of the function.
  - `block`: A `BlockNode` representing the body of the function.

ParamNode

The `ParamNode` represents a parameter in a function declaration, including its name and type.

```
class ParamNode:
    def __init__(self, identifier, param_type):
        self.identifier = identifier
        self.param_type = param_type

    def __str__(self):
        return f"ParamNode(identifier={self.identifier}, param_type=
{self.param_type})"
```

- **Attributes**:
  - `identifier`: The name of the parameter.
  - `param_type`: The type of the parameter.

BlockNode

The `BlockNode` represents a block of statements enclosed in curly braces `{}`.

```
class BlockNode:
    def __init__(self, statements):
        self.statements = statements

    def __str__(self):
        return f"BlockNode(statements={self.statements})"
```

- **Attributes**:
  - `statements`: A list of statements contained within the block.

VariableDeclNode

The `VariableDeclNode` represents a variable declaration, including its name, type, and optional initializer expression.

```
class VariableDeclNode:
    def __init__(self, identifier, var_type, expr=None):
        self.identifier = identifier
        self.var_type = var_type
        self.expr = expr

    def __str__(self):
        return (f"VariableDeclNode(identifier={self.identifier}, var_type=
{self.var_type}, "
                f"expr={self.expr})")
```

- **Attributes**:
  - `identifier`: The name of the variable.
  - `var_type`: The type of the variable.
  - `expr`: An optional expression representing the initializer.

AssignmentNode

The `AssignmentNode` represents an assignment statement, where a value is assigned to a variable.

```
class AssignmentNode:
    def __init__(self, identifier, expr):
        self.identifier = identifier
        self.expr = expr

    def __str__(self):
        return f"AssignmentNode(identifier={self.identifier}, expr=
{self.expr})"
```

- **Attributes**:
  - `identifier`: The name of the variable being assigned.
  - `expr`: The expression whose value is assigned to the variable.

ReturnStatementNode

The `ReturnStatementNode` represents a return statement in a function.

```
class ReturnStatementNode:
    def __init__(self, expr):
        self.expr = expr

    def __str__(self):
        return f"ReturnStatementNode(expr={self.expr})"
```

- **Attributes**:
  - `expr`: The expression whose value is returned by the function.

IfStatementNode

The `IfStatementNode` represents an if-else statement, including the condition, if-block, and optional else-block.

```
class IfStatementNode:
    def __init__(self, condition, if_block, else_block=None):
        self.condition = condition
        self.if_block = if_block
        self.else_block = else_block

    def __str__(self):
        return (f"IfStatementNode(condition={self.condition}, if_block=
{self.if_block}, "
                f"else_block={self.else_block})")
```

- **Attributes**:
  - `condition`: The condition expression that determines which block to execute.
  - `if_block`: The block of statements executed if the condition is true.
  - `else_block`: An optional block of statements executed if the condition is false.

ForStatementNode

The `ForStatementNode` represents a for loop, including the initializer, condition, post-expression, and body.

```
class ForStatementNode:
    def __init__(self, init, condition, post, block):
        self.init = init
        self.condition = condition
        self.post = post
        self.block = block

    def __str__(self):
        return (f"ForStatementNode(init={self.init}, condition=
{self.condition}, "
                f"post={self.post}, block={self.block})")
```

- **Attributes**:
  - `init`: The initializer statement executed once before the loop starts.
  - `condition`: The condition expression evaluated before each iteration.
  - `post`: The post-expression executed after each iteration.
  - `block`: The block of statements executed in each iteration.

WhileStatementNode

The `WhileStatementNode` represents a while loop, including the condition and body.

```
class WhileStatementNode:
    def __init__(self, condition, block):
        self.condition = condition
        self.block = block

    def __str__(self):
        return f"WhileStatementNode(condition={self.condition}, block=
{self.block})"
```

- **Attributes**:
  - `condition`: The condition expression evaluated before each iteration.
  - `block`: The block of statements executed in each iteration.

PrintStatementNode

The `PrintStatementNode` represents a print statement that outputs the value of an expression.

```
class PrintStatementNode:
    def __init__(self, expr):
        self.expr = expr

    def __str__(self):
        return f"PrintStatementNode(expr={self.expr})"
```

- **Attributes**:
  - `expr`: The expression whose value is printed.

DelayStatementNode

The `DelayStatementNode` represents a delay statement that pauses execution for a specified duration.

```
class DelayStatementNode:
    def __init__(self, expr):
        self.expr = expr

    def __str__(self):
        return f"DelayStatementNode(expr={self.expr})"
```

- **Attributes**:
  - `expr`: The expression representing the delay duration.

WriteStatementNode

The `WriteStatementNode` represents a write statement that outputs values to a specified location.

```
class WriteStatementNode:
    def __init__(self, args):
        self.args = args

    def __str__(self):
        return f"WriteStatementNode(args={self.args})"
```

- **Attributes**:
  - `args`: A list of expressions representing the arguments of the write statement.

BinaryOpNode

The `BinaryOpNode` represents a binary operation, including the left operand, operator, and right operand.

```
class BinaryOpNode:
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right

    def __str__(self):
        return f"BinaryOpNode(left={self.left}, operator={self.operator},
right={self.right})"
```

- **Attributes**:
  - `left`: The left operand of the binary operation.
  - `operator`: The binary operator (e.g., `+`, `-`, `*`, `/`).
  - `right`: The right operand of the binary operation.

UnaryOpNode

The `UnaryOpNode` represents a unary operation, including the operator and operand.

```
class UnaryOpNode:
    def __init__(self, operator, operand):
        self.operator = operator
        self.operand = operand

    def __str__(self):
        return f"UnaryOpNode(operator={self.operator}, operand=
{self.operand})"
```

- **Attributes**:
  - `operator`: The unary operator (e.g., `-`, `not`).
  - `operand`: The operand of the unary operation.

LiteralNode

The `LiteralNode` represents a literal value, such as a number, boolean, or color.

```
class LiteralNode:
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f"LiteralNode(value={self.value})"
```

- **Attributes**:
  - `value`: The literal value.

IdentifierNode

The `IdentifierNode` represents an identifier, such as a variable or function name.

```
class IdentifierNode:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"IdentifierNode(name={self.name})"
```

- **Attributes**:
  - `name`: The name of the identifier

.

FunctionCallNode

The `FunctionCallNode` represents a function call, including the function name and arguments.

```
class FunctionCallNode:
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __str__(self):
        return f"FunctionCallNode(name={self.name}, args={self.args})"
```

- **Attributes**:
  - `name`: The name of the function being called.
  - `args`: A list of expressions representing the arguments passed to the function.

CastNode

The `CastNode` represents a type cast, where an expression is cast to a different type.

```
class CastNode:
    def __init__(self, expr, target_type):
        self.expr = expr
        self.target_type = target_type

    def __str__(self):
        return f"CastNode(expr={self.expr}, target_type=
{self.target_type})"
```

- **Attributes**:
  - `expr`: The expression being cast.
  - `target_type`: The type to which the expression is being cast.

These classes define the various node types that will be used to build the AST during parsing. Each node class includes an `__init__` method for initializing its attributes and a `__str__` method for generating a string representation of the node. The `__str__` methods are particularly useful for debugging and traversing the AST to verify its structure.

Traversing the AST

A utility function `traverse` iwas created to print the structure of the AST. This function recursively visits each node and prints its type and attributes.

```
def traverse(node, indent=0):
    ind = '  ' * indent
    if isinstance(node, ProgramNode):
        print(f"{ind}ProgramNode:")
        for stmt in node.statements:
            traverse(stmt, indent + 1)
    elif isinstance(node, FunctionDeclNode):
        print(f"{ind}FunctionDeclNode: {node.identifier}")
        print(f"{ind}  Params:")
        for param in node.params:
            traverse(param, indent + 2)
        print(f"{ind}  ReturnType: {node.return_type}")
        traverse(node.block, indent + 1)
    elif isinstance(node, ParamNode):
        print(f"{ind}ParamNode: {node.identifier}: {node.param_type}")
    elif isinstance(node, BlockNode):
        print(f"{ind}BlockNode:")
        for stmt in node.statements:
```

```
            traverse(stmt, indent + 1)
    elif isinstance(node, VariableDeclNode):
        print(f"{ind}VariableDeclNode: {node.identifier}:
{node.var_type}")
        if node.expr:
            traverse(node.expr, indent + 1)
    elif isinstance(node, AssignmentNode):
        print(f"{ind}AssignmentNode: {node.identifier}")
        traverse(node.expr, indent + 1)
    elif isinstance(node, ReturnStatementNode):
        print(f"{ind}ReturnStatementNode:")
        traverse(node.expr, indent + 1)
    elif isinstance(node, IfStatementNode):
        print(f"{ind}IfStatementNode:")
        print(f"{ind}  Condition:")
        traverse(node.condition, indent + 2)
        print(f"{ind}  IfBlock:")
        traverse(node.if_block, indent + 2)
        if node.else_block:
            print(f"{ind}  ElseBlock:")
            traverse(node.else_block, indent + 2)
    elif isinstance(node, ForStatementNode):
        print(f"{ind}ForStatementNode:")
        if node.init:
            print(f"{ind}  Init:")
            traverse(node.init, indent + 2)
        print(f"{ind}  Condition:")
        traverse(node.condition, indent + 2)
        if node.post:
            print(f"{ind}  Post:")
            traverse(node.post, indent + 2)
        print(f"{ind}  Block:")
        traverse(node.block, indent + 2)
    elif isinstance(node, WhileStatementNode):
        print(f"{ind}WhileStatementNode:")
        print(f"{ind}  Condition:")
        traverse(node.condition, indent + 2)
        print(f"{ind}  Block:")
        traverse(node.block, indent + 2)
    elif isinstance(node, PrintStatementNode):
        print(f"{ind}PrintStatementNode:")
        traverse(node.expr, indent + 1)
    elif isinstance(node, DelayStatementNode):
        print(f"{ind}DelayStatementNode:")
        traverse(node.expr, indent + 1)
    elif isinstance(node, WriteStatementNode):
        print(f"{ind}WriteStatementNode:")
        for arg in node.args:
            traverse(arg, indent + 1)
    elif isinstance(node, BinaryOpNode):
        print(f"{ind}BinaryOpNode: {node.operator}")
        traverse(node.left, indent + 1)
        traverse(node.right, indent + 1)
    elif isinstance(node, UnaryOpNode):
```

```
        print(f"{ind}UnaryOpNode: {node.operator}")
        traverse(node.operand, indent + 1)
    elif isinstance(node, LiteralNode):
        print(f"{ind}LiteralNode: {node.value}")


 elif isinstance(node, IdentifierNode):
        print(f"{ind}IdentifierNode: {node.name}")
    elif isinstance(node, FunctionCallNode):
        print(f"{ind}FunctionCallNode: {node.name}")
        for arg in node.args:
            traverse(arg, indent + 1)
    elif isinstance(node, CastNode):
        print(f"{ind}CastNode:")
        print(f"{ind}  Expr:")
        traverse(node.expr, indent + 2)
        print(f"{ind}  TargetType: {node.target_type}")
    else:
        print(f"{ind}Unknown node: {node}")
```

Parser Implementation

The parser processes the tokens generated by the lexer and constructs the AST. The parser is implemented in the `LLK_parser.py` file.

Initialization

The `Parser` class initializes with a list of tokens and manages the current position within this list.

```
from parser_nodes import *
from lexer import Lexer

class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.current_token_index = 0
        self.current_token = self.tokens[self.current_token_index]

    def advance(self):
        self.current_token_index += 1
        if self.current_token_index < len(self.tokens):
            self.current_token = self.tokens[self.current_token_index]
        else:
            self.current_token = ('EOF', '')

    def parse(self):
        return self.parse_program()
```

Program Parsing

The `parse_program` method is the entry point for parsing the entire source code. It iterates over the tokens, parsing each statement until the end of the file is reached.

```
    def parse_program(self):
        statements = []
        while self.current_token[0] != 'EOF':
            statements.append(self.parse_statement())
        return ProgramNode(statements)
```

Statement Parsing

The `parse_statement` method determines the type of statement based on the current token and calls the appropriate parsing method.

```
    def parse_statement(self):
        if self.current_token[0] == 'KEYWORD':
            if self.current_token[1] == 'fun':
                return self.parse_function_decl()
            elif self.current_token[1] == 'let':
                return self.parse_variable_decl()
            elif self.current_token[1] == 'return':
                return self.parse_return_statement()
            elif self.current_token[1] == 'if':
                return self.parse_if_statement()
            elif self.current_token[1] == 'for':
                return self.parse_for_statement()
            elif self.current_token[1] == 'while':
                return self.parse_while_statement()
            else:
                raise SyntaxError(f"Unexpected keyword
{self.current_token[1]}")
        elif self.current_token[0] == 'IDENTIFIER':
            return self.parse_assignment()
        elif self.current_token[0] == 'DELIMITER' and
self.current_token[1] == '{':
            return self.parse_block()
        elif self.current_token[0] == 'SPECIAL_FUNCTION':
            if self.current_token[1] == '__print':
                return self.parse_print_statement()
            elif self.current_token[1] == '__delay':
                return self.parse_delay_statement()
            elif self.current_token[1] == '__write' or
self.current_token[1] == '__write_box':
                return self.parse_write_statement()
        else:
            raise SyntaxError(f"Unexpected token {self.current_token}")
```

Function Declaration Parsing

The `parse_function_decl` method is responsible for parsing function declarations in the PArL language. This includes parsing the function name, parameters, return type, and the body of the function, which is a block of statements.

`parse_function_decl` Method

The method starts by advancing past the `fun` keyword and then processes each part of the function declaration sequentially:

1. **Function Name**:
    - The function name is expected to be an identifier. The current token should be an identifier representing the function's name.
    - The method captures this identifier and then advances to the next token.
2. **Parameter List**:
    - The parameter list starts with an opening parenthesis `(` and ends with a closing parenthesis `)`.
    - The `expect` method is used to ensure the current token is the expected delimiter (`(`). If it is not, a `SyntaxError` is raised.
    - The `parse_formal_params` method is called to parse the parameters within the parentheses.
    - After parsing the parameters, the method again uses `expect` to ensure the closing parenthesis `)` is present.
3. **Return Type**:
    - The return type is indicated by an arrow `->` followed by the type.
    - The `expect` method ensures the current token is the arrow `->`.
    - The next token should be the return type, which is captured and the method advances past it.
4. **Function Body**:
    - The function body is a block of statements enclosed in curly braces `{}`.
    - The `parse_block` method is called to parse the block of statements that form the function body.

The method then returns a `FunctionDeclNode` containing the parsed function name, parameters, return type, and body.

```
def parse_function_decl(self):
    self.advance()  # skip 'fun'
    identifier = self.current_token[1]
    self.advance()  # skip identifier
    self.expect('DELIMITER', '(')
    params = self.parse_formal_params()
    self.expect('DELIMITER', ')')
    self.expect('ARROW', '->')
    return_type = self.current_token[1]
    self.advance()  # skip return type
```

```
        block = self.parse_block()
        return FunctionDeclNode(identifier, params, return_type, block)
```

`parse_formal_params` Method

This method handles the parsing of the function's formal parameters:

1. **Parameter List**:
    - An empty list `params` is initialized to store the parsed parameters.
    - If the current token is not a closing parenthesis `)`, it indicates the presence of parameters.
    - The `parse_param` method is called to parse the first parameter.
    - A `while` loop checks for additional parameters, which are separated by commas. The loop continues to call `parse_param` until no more commas are found.

The method returns the list of `ParamNode` objects.

```
def parse_formal_params(self):
    params = []
    if self.current_token[0] != 'DELIMITER' or self.current_token[1] !=
')':
        params.append(self.parse_param())
        while self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ',':
            self.advance()  # skip ','
            params.append(self.parse_param())
    return params
```

`parse_param` Method

This method parses an individual parameter:

1. **Parameter Name**:
    - The current token should be an identifier representing the parameter's name.
    - This identifier is captured and the method advances to the next token.
2. **Parameter Type**:
    - The `expect` method ensures that the next token is a colon `:` separating the parameter name from its type.
    - The subsequent token is expected to be the type of the parameter, which is captured and the method advances past it.

The method returns a `ParamNode` representing the parameter.

```
def parse_param(self):
    identifier = self.current_token[1]
    self.advance()  # skip identifier
```

```
        self.expect('DELIMITER', ':')
        param_type = self.current_token[1]
        self.advance()  # skip type
        return ParamNode(identifier, param_type)
```

Block Parsing

The `parse_block` method parses a block of statements enclosed in curly braces `{}`. This method ensures that the block starts and ends with the correct delimiters and processes each statement within the block.

1. **Expect Opening Brace**:
    - The method starts by calling `expect` to ensure the current token is a `{`. If it is not, a `SyntaxError` is raised.
2. **Parse Statements**:
    - It initializes an empty list `statements` to store the parsed statements.
    - A `while` loop iterates through the tokens, calling `parse_statement` to parse each statement. This continues until the current token is a closing brace `}`.
3. **Expect Closing Brace**:
    - After parsing all the statements, `expect` is called again to ensure the current token is a `}`. If it is not, a `SyntaxError` is raised.
4. **Return BlockNode**:
    - Finally, the method returns a `BlockNode` containing the list of parsed statements.

```
def parse_block(self):
    self.expect('DELIMITER', '{')
    statements = []
    while self.current_token[0] != 'DELIMITER' or self.current_token[1] !=
'}':
        statements.append(self.parse_statement())
    self.expect('DELIMITER', '}')
    return BlockNode(statements)
```

Variable Declaration and Assignment Parsing

These methods parse variable declarations and assignments, ensuring that the syntax is correct and capturing the necessary information.

`parse_variable_decl` Method

This method handles the parsing of variable declarations:

1. **Skip 'let' Keyword**:
    - The method starts by advancing past the `let` keyword.
2. **Parse Identifier**:

- The current token should be an identifier representing the variable name. The identifier is captured and the method advances to the next token.

3. **Expect Colon and Parse Type**:
    - The `expect` method ensures the next token is a colon `:`. The subsequent token should be the variable type, which is captured and the method advances past it.
4. **Optional Initializer**:
    - If the next token is an equals sign `=`, the method advances and calls `parse_expression` to parse the initializer expression.
5. **Expect Semicolon**:
    - If `expect_semicolon` is `True`, the method ensures the declaration ends with a semicolon `;`.
6. **Return VariableDeclNode**:
    - The method returns a `VariableDeclNode` containing the parsed identifier, type, and optional initializer expression.

```
def parse_variable_decl(self, expect_semicolon=True):
    self.advance()  # skip 'let'
    identifier = self.current_token[1]
    self.advance()  # skip identifier
    self.expect('DELIMITER', ':')
    var_type = self.current_token[1]
    self.advance()  # skip type
    expr = None
    if self.current_token[0] == 'OPERATOR' and self.current_token[1] ==
'=':
        self.advance()  # skip '='
        expr = self.parse_expression()
    if expect_semicolon:
        self.expect('DELIMITER', ';')
    return VariableDeclNode(identifier, var_type, expr)
```

`parse_assignment` Method

This method handles the parsing of assignments:

1. **Parse Identifier**:
    - The method captures the identifier of the variable being assigned and advances to the next token.
2. **Expect Equals Sign and Parse Expression**:
    - The `expect` method ensures the next token is an equals sign `=`.

- The method then calls `parse_expression` to parse the expression being assigned to the variable.

3. **Expect Semicolon**:
    - If `expect_semicolon` is `True`, the method ensures the assignment ends with a semicolon `;`.
4. **Return AssignmentNode**:
    - The method returns an `AssignmentNode` containing the parsed identifier and expression.

```
def parse_assignment(self, expect_semicolon=True):
    identifier = self.current_token[1]
    self.advance()  # skip identifier
    self.expect('OPERATOR', '=')
    expr = self.parse_expression()
    if expect_semicolon:
        self.expect('DELIMITER', ';')
    return AssignmentNode(identifier, expr)
```

Expression Parsing

Expression parsing involves multiple methods to handle different levels of operator precedence. Each method deals with a specific level, ensuring that expressions are parsed correctly according to their precedence rules.

Purpose and Functionality

- **Purpose**: The method is designed to start the parsing process at the highest level of precedence (type casts). This ensures that operations with the highest precedence are processed first, allowing the parser to correctly handle operator precedence and associativity.
- **Functionality**: By starting with the highest precedence level and moving down to the lowest, the parser ensures that when the recursive calls unwind, the operation with the highest precedence becomes the parent node in the AST. This is because the node for the highest precedence operation is created last, and thus it naturally becomes the parent of the lower precedence operations.

`parse_expression` Method

The `parse_expression` method serves as the entry point for parsing expressions. It calls the `parse_cast` method to begin the parsing process at the highest precedence level.

```
def parse_expression(self):
    return self.parse_cast()
```

`parse_cast` Method

Handles type casts, where an expression is cast to a different type using the `as` keyword:

1. **Parse Or-Level Expression**:
    - The method starts by parsing an expression at the `or` level.
2. **Check for Casts**:
    - A `while` loop checks if the current token is the `as` keyword. If it is, the method captures the target type and continues parsing.
3. **Return CastNode**:
    - If a cast is found, the method returns a `CastNode` containing the original expression and the target type.

```
def parse_cast(self):
    left = self.parse_or()
    while self.current_token[0] == 'KEYWORD' and self.current_token[1] ==
'as':
        self.advance()  # skip 'as'
        target_type = self.current_token[1]
        self.advance()  # skip type
        left = CastNode(left, target_type)
    return left
```

`parse_or` Method

Handles logical `or` operations:

1. **Parse And-Level Expression**:
    - The method starts by parsing an expression at the `and` level.
2. **Check for Or Operators**:
    - A `while` loop checks if the current token is the `or` operator. If it is, the method captures the operator, parses the right operand, and continues parsing.
3. **Return BinaryOpNode**:
    - If `or` operators are found, the method returns a `BinaryOpNode` representing the `or` operation.

```
def parse_or(self):
    left = self.parse_and()
    while self.current_token[0] == 'OPERATOR' and self.current_token[1] ==
'or':
        operator = self.current_token[1]
        self.advance()
        right = self.parse_and()
        left = BinaryOpNode(left, operator, right)
    return left
```

`parse_and` Method

Handles logical `and` operations:

1. **Parse Equality-Level Expression**:
    - The method starts by parsing an expression at the equality level.
2. **Check for And Operators**:
    - A `while` loop checks if the current token is the `and` operator. If it is, the method captures the operator, parses the right operand, and continues parsing.
3. **Return BinaryOpNode**:
    - If `and` operators are found, the method returns a `BinaryOpNode` representing the `and` operation.

```
def parse_and(self):
    left = self.parse_equality()
    while self.current_token[0] == 'OPERATOR' and self.current_token[1] ==
'and':
        operator = self.current_token[1]
        self.advance()
        right = self.parse_equality()
        left = BinaryOpNode(left, operator, right)
    return left
```

`parse_equality` Method

Handles equality and inequality operations (`==`, `!=`):

1. **Parse Comparison-Level Expression**:
    - The method starts by parsing an expression at the comparison level.
2. **Check for Equality Operators**:
    - A `while` loop checks if the current token is an equality operator (`==`, `!=`). If it is, the method captures the operator, parses the right operand, and continues parsing.
3. **Return BinaryOpNode**:
    - If equality operators are found, the method returns a `BinaryOpNode` representing the equality operation.

```
def parse_equality(self):
    left = self.parse_comparison()
    while self.current_token[0] == 'OPERATOR' and self.current_token[1] in
('==', '!='):
        operator = self.current_token[1]
        self.advance()
        right = self.parse_comparison()
        left = BinaryOpNode(left, operator, right)
    return left
```

`parse_comparison` Method

Handles comparison operations (`<`, `>`, `<=`, `>=`):

1. **Parse Term-Level Expression**:
    - The method starts by parsing an expression at the term level.
2. **Check for Comparison Operators**:
    - A `while` loop checks if the current token is a comparison operator (`<`, `>`, `<=`, `>=`). If it is, the method captures the operator, parses the right operand, and continues parsing.
3. **Return BinaryOpNode**:
    - If comparison operators are found, the method returns a `BinaryOpNode` representing the comparison operation.

```
def parse_comparison(self):
    left = self.parse_term()
    while self.current_token[0] == 'OPERATOR' and self.current_token[1] in
('<', '>', '<=', '>='):
        operator = self.current_token[1]
        self.advance()
        right = self.parse_term()
        left = BinaryOpNode(left, operator, right)
    return left
```

`parse_term` Method

Handles addition and subtraction operations (`+`, `-`):

1. **Parse Factor-Level Expression**:
    - The method starts by parsing an expression at the factor level.
2. **Check for Addition and Subtraction Operators**:
    - A `while` loop checks if the current token is an addition or subtraction operator (`+`, `-`). If it is, the method captures the operator, parses the right operand, and continues parsing.
3. **Return BinaryOpNode**:
    - If addition or subtraction operators are found, the method returns a `BinaryOpNode` representing the term operation.

```
def parse_term(self):
    left = self.parse_factor()
    while self.current_token[0] == 'OPERATOR' and self.current_token[1] in
('+', '-'):
        operator = self.current_token[1]
        self.advance()
        right = self.parse_factor()
        left = BinaryOpNode(left, operator, right)
    return left
```

`parse_factor` Method

Handles multiplication and division operations (`*`, `/`):

1. **Parse Unary-Level Expression**:
    - The method starts by parsing an expression at the unary level.
2. **Check for Multiplication and Division Operators**:
    - A `while` loop checks if the current token is a multiplication or division operator (`*`, `/`). If it is, the method captures the operator, parses the right operand, and continues parsing.
3. **Return BinaryOpNode**:
    - If multiplication or division operators are found, the method returns a `BinaryOpNode` representing the factor operation.

```
def parse_factor(self):
    left = self.parse_unary()
    while self.current_token[0] == 'OPERATOR' and self.current_token[1] in
('*', '/'):
        operator = self.current_token[1]
        self.advance()
        right = self.parse_unary()
        left = BinaryOpNode(left, operator, right)
    return left
```

`parse_unary` Method

Handles unary operations (`-`, `not`):

1. **Check for Unary Operators**:
    - The method checks if the current token is a unary operator (`-`, `not`). If it is, the method captures the operator, advances, and parses the operand.
2. **Return UnaryOpNode**:
    - If a unary operator is found, the method returns a `UnaryOpNode` representing the unary operation.
3. **Parse Primary Expression**:
    - If no unary operator is found, the method calls `parse_primary` to handle primary expressions.

```
def parse_unary(self):
    if self.current_token[0] == 'OPERATOR' and self.current_token[1] in
('-', 'not'):
        operator = self.current_token[1]
        self.advance()
        operand = self.parse_unary()
        return UnaryOpNode(operator, operand)
    return self.parse_primary()
```

Precedence of Processes

The parser handles expressions by dividing the parsing tasks according to different levels of operator precedence. This division ensures that higher precedence operations are correctly nested within lower

precedence operations. Here is the list of the precedence levels in descending order:

1. **Type Casts**:
    - Handled by the `parse_cast` method. Type casts have the highest precedence and are parsed first to ensure correct type conversions.
2. **Logical Or**:
    - Handled by the `parse_or` method. Logical `or` operations are parsed next, after type casts.
3. **Logical And**:
    - Handled by the `parse_and` method. Logical `and` operations have higher precedence than `or` but lower than type casts.
4. **Equality and Inequality**:
    - Handled by the `parse_equality` method. These operations (`==`, `!=`) are parsed after logical operations.
5. **Comparison**:
    - Handled by the `parse_comparison` method. Comparison operators (`<`, `>`, `<=`, `>=`) have higher precedence than equality operators.
6. **Addition and Subtraction**:
    - Handled by the `parse_term` method. These arithmetic operations have higher precedence than comparison operators.
7. **Multiplication and Division**:
    - Handled by the `parse_factor` method. These operations have higher precedence than addition and subtraction.
8. **Unary Operations**:
    - Handled by the `parse_unary` method. Unary operations (`-`, `not`) have higher precedence than binary arithmetic operations.
9. **Primary Expressions**:
    - Handled by the `parse_primary` method. These are the most fundamental expressions, including literals, identifiers, and function calls, and they have the lowest precedence in the parsing hierarchy.

By following this order, the parser correctly constructs the AST, ensuring that operations with higher precedence are correctly nested within those of lower precedence.

Primary Expressions

The `parse_primary` method handles primary expressions such as literals, identifiers, and function calls. These are the simplest and most fundamental components of expressions.

1. **Literals**:
    - If the current token is a literal (`true`, `false`, number, color), the method captures the value and advances to the next token, returning a `LiteralNode`.
2. **Identifiers**:
    - If the current token is an identifier, the method captures the name and advances.
    - If the next token is an opening parenthesis `(`, it indicates a function call. The method captures the arguments by calling `parse_expression` and expects a closing parenthesis `)`. It returns a `FunctionCallNode`.
    - Otherwise, it returns an `IdentifierNode`.
3. **Parenthesized Expressions**:
    - If the current token is an opening parenthesis `(`, the method parses an expression and expects a closing parenthesis `)`. It returns the parsed expression.
4. **Special Functions**:
    - If the current token is a special function (`__print`, `__delay`, etc.), the method captures the function name and arguments, returning a `FunctionCallNode`.
5. **Error Handling**:
    - If the current token does not match any expected primary expression type, the method raises a `SyntaxError`.

```
def parse_primary(self):
    token = self.current_token
    if token[0] == 'LITERAL':
        self.advance()
        if token[1] in {"true", "false"}:
            value = True if token[1] == "true" else False
            return LiteralNode(value)
        elif token[1].startswith("#"):
            return LiteralNode(token[1])
        elif token[1].isdigit() or (token[1].replace('.', '', 1).isdigit()
and token[1].count('.') < 2):
            value = int(token[1]) if '.' not in token[1] else
float(token[1])
            return LiteralNode(value)
        else:
            raise SyntaxError(f"Unexpected literal {token[1]}")
    elif token[0] == 'IDENTIFIER':
        identifier = token[1]
        self.advance()
        if self.current_token[0] == 'DELIMITER' and self.current_token[1]
== '(':
```

```
            self.advance()  # skip '('
            args = []
            while self.current_token[0] != 'DELIMITER' or
self.current_token[1] != ')':
                args.append(self.parse_expression())
                if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ',':
                    self.advance()  # skip ','
            self.expect('DELIMITER', ')')
            return FunctionCallNode(identifier, args)
        return IdentifierNode(identifier)
    elif token[0] == 'DELIMITER' and token[1] == '(':
        self.advance()
        expr = self.parse_expression()
        self.expect('DELIMITER', ')')
        return expr
    elif token[0] == 'SPECIAL_FUNCTION':
        func_name = token[1]
        self.advance()
        args = []
        while self.current_token[0] != 'DELIMITER' or
self.current_token[1] != ';':
            args.append(self.parse_expression())
            if self.current_token[0] == 'DELIMITER' and
self.current_token[1] == ',':
                self.advance()  # skip ','
        return FunctionCallNode(func_name, args)
    raise SyntaxError(f"Unexpected token in expression: {token}")
```

Expect Method

The `expect` method ensures the current token matches the expected type and value. If the token does not match, a `SyntaxError` is raised. This method is crucial for maintaining the integrity of the parsing process by verifying that the tokens conform to the expected structure.

1. **Check Token Type and Value**:
    - The method checks if the current token's type matches the expected type. If a specific value is expected, it also checks the token's value.
2. **Raise Error if Mismatch**:
    - If the token type or value does not match the expected values, the method raises a `SyntaxError` with a descriptive message.
3. **Advance Token**:
    - If the token matches, the method advances to the next token.

```
def expect(self, token_type, value=None):
    if self.current_token[0] != token_type or (value and
```

```
self.current_token[1] != value):
        raise SyntaxError(f"Expected token {token_type} with value
{value}, but got {self.current_token}")
    self.advance()
```

By ensuring that each section is parsed correctly and handling errors appropriately, the parser maintains a robust structure for processing the PArL language, laying a solid foundation for subsequent stages in the compilation process.

Conclusion

The hand-crafted LL(k) parser processes tokens generated by the lexer and constructs an Abstract Syntax Tree (AST) that represents the syntactic structure of the PArL source code. The parser handles various language constructs, including functions, variable declarations, control structures, and expressions.

Task 3: Semantic Analysis Implementation

The semantic analyzer ensures that the syntax tree generated by the parser adheres to the semantic rules of the PArL language. It performs tasks such as type checking, scope management, and ensuring that variables are declared before they are used. The semantic analyzer uses a symbol table to keep track of declared variables and their types.

Symbol Table

The `SymbolTable` class manages variable declarations within different scopes. It supports entering and exiting scopes, declaring variables, and looking up variable types.

```
class SymbolTable:
    def __init__(self):
        self.scopes = [{}]

    def enter_scope(self):
        self.scopes.append({})

    def exit_scope(self):
        self.scopes.pop()

    def declare(self, name, type):
        if name in self.scopes[-1]:
            raise Exception(f"Variable '{name}' already declared in the
same scope")
        self.scopes[-1][name] = type

    def lookup(self, name):
        for scope in reversed(self.scopes):
            if name in scope:
                return scope[name]
        raise Exception(f"Variable '{name}' not declared")

    def is_declared(self, name):
```

```
        for scope in self.scopes:
            if name in scope:
                return True
        return False
```

- **enter_scope**: Pushes a new scope onto the stack. This is used to handle nested blocks, such as the body of a function or a conditional statement.
- **exit_scope**: Pops the current scope off the stack, effectively discarding all variable declarations in that scope.
- **declare**: Declares a variable in the current scope. If a variable with the same name already exists in the current scope, an exception is raised to prevent redeclaration.
- **lookup**: Looks up the type of a variable, searching from the innermost scope outward. If the variable is not found in any scope, an exception is raised.
- **is_declared**: Checks if a variable is declared in any scope. This is used to check if a variable is already declared in the current or any outer scope.

Semantic Analyzer

The `SemanticAnalyzer` class visits each node in the AST and performs semantic checks. It uses the visitor pattern, where each node type has a corresponding visit method.

```
class SemanticAnalyzer:
    def __init__(self):
        self.symbol_table = SymbolTable()
        self.current_function_return_type = None

    def visit(self, node):
        method_name = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception(f'No visit_{node.__class__.__name__} method')
```

- **visit**: The main method for visiting nodes. It dynamically finds and calls the appropriate visit method for the node's class.
- **generic_visit**: Raises an exception if no specific visit method is found for a node. This ensures that every node type has a corresponding visit method.

Visiting Program Nodes

The `visit_ProgramNode` method visits each statement in the program.

```
def visit_ProgramNode(self, node):
    for stmt in node.statements:
        self.visit(stmt)
```

- **Functionality**: Iterates over each statement in the program and visits it. This ensures that all top-level statements are semantically analyzed.

Visiting Function Declarations

The `visit_FunctionDeclNode` method handles function declarations, including parameter declaration and type checking for the function body.

```
def visit_FunctionDeclNode(self, node):
    self.symbol_table.enter_scope()
    for param in node.params:
        self.visit(param)
    self.current_function_return_type = node.return_type
    self.visit(node.block)
    self.symbol_table.exit_scope()
```

- **enter_scope**: Enters a new scope for the function body.
- **Parameter Declaration**: Visits each parameter to declare it in the current scope.
- **Return Type**: Sets the expected return type for the function.
- **Function Body**: Visits the function body, ensuring all statements are checked.
- **exit_scope**: Exits the function scope after checking the function body.

Visiting Parameters

The `visit_ParamNode` method declares function parameters in the current scope.

```
def visit_ParamNode(self, node):
    self.symbol_table.declare(node.identifier, node.param_type)
```

- **Functionality**: Declares the parameter in the current scope with its specified type. This ensures that the parameter can be used within the function body.

Visiting Blocks

The `visit_BlockNode` method visits each statement in a block.

```
def visit_BlockNode(self, node):
    self.symbol_table.enter_scope()
    for stmt in node.statements:
        self.visit(stmt)
    self.symbol_table.exit_scope()
```

- **enter_scope**: Enters a new scope for the block.

- **Functionality**: Iterates over each statement in the block and visits it, ensuring all block-level statements are semantically analyzed.
- **exit_scope**: Exits the block scope after checking all statements.

Visiting Variable Declarations

The `visit_VariableDeclNode` method checks the type of the initializer expression and ensures it matches the declared type.

```
def visit_VariableDeclNode(self, node):
    expr_type = self.visit(node.expr)
    self.symbol_table.declare(node.identifier, node.var_type)
    if node.var_type != expr_type:
        raise Exception(f"Type mismatch: cannot assign {expr_type} to
{node.var_type} in variable declaration of '{node.identifier}'")
```

- **Expression Type**: Visits the initializer expression to determine its type.
- **Declaration**: Declares the variable in the current scope.
- **Type Checking**: Ensures the type of the initializer expression matches the declared type of the variable. Raises an exception if there is a mismatch.

Visiting Assignments

The `visit_AssignmentNode` method checks that the type of the assigned expression matches the type of the variable.

```
def visit_AssignmentNode(self, node):
    var_type = self.symbol_table.lookup(node.identifier)
    expr_type = self.visit(node.expr)
    if var_type != expr_type:
        raise Exception(f"Type mismatch: cannot assign {expr_type} to
{var_type} in assignment to '{node.identifier}'")
```

- **Variable Lookup**: Looks up the type of the variable being assigned to.
- **Expression Type**: Visits the assigned expression to determine its type.
- **Type Checking**: Ensures the type of the assigned expression matches the type of the variable. Raises an exception if there is a mismatch.

Visiting Return Statements

The `visit_ReturnStatementNode` method ensures the return type of the expression matches the function's return type.

```
def visit_ReturnStatementNode(self, node):
    expr_type = self.visit(node.expr)
```

```
        if expr_type != self.current_function_return_type:
            raise Exception(f"Return type mismatch in function with return
type {self.current_function_return_type}: got {expr_type}")
```

- **Expression Type**: Visits the return expression to determine its type.
- **Type Checking**: Ensures the type of the return expression matches the function's declared return type. Raises an exception if there is a mismatch.

Visiting If Statements

The `visit_IfStatementNode` method checks the condition and visits both the if-block and the optional else-block.

```
def visit_IfStatementNode(self, node):
    self.visit(node.condition)
    self.visit(node.if_block)
    if node.else_block:
        self.visit(node.else_block)
```

- **Condition**: Visits the condition expression to ensure it is semantically correct.
- **If Block**: Visits the statements within the if block.
- **Else Block**: If present, visits the statements within the else block.

Visiting While Statements

The `visit_WhileStatementNode` method checks the condition and visits the loop body.

```
def visit_WhileStatementNode(self, node):
    self.visit(node.condition)
    self.visit(node.block)
```

- **Condition**: Visits the condition expression to ensure it is semantically correct.
- **Loop Body**: Visits the statements within the while loop body.

Visiting For Statements

The `visit_ForStatementNode` method visits the initialization, condition, post-expression, and loop body, ensuring each part is semantically correct.

```
def visit_ForStatementNode(self, node):
    self.symbol_table.enter_scope()
    self.visit(node.init)
    self.visit(node.condition)
    self.visit(node.post)
```

```
        self.visit(node.block)
        self.symbol_table.exit_scope()
```

- **enter_scope**: Enters a new scope for the for loop.
- **Initialization**: Visits the initialization statement.
- **Condition**: Visits the loop condition expression.
- **Post-Expression**: Visits the post-expression.
- **Loop Body**: Visits the statements within the for loop body.
- **exit_scope**: Exits the for loop scope after checking all parts of the loop.

Visiting Print Statements

The `visit_PrintStatementNode` method ensures the expression to be printed is semantically correct.

```
def visit_PrintStatementNode(self, node):
    self.visit(node.expr)
```

- **Expression**: Visits the expression to be printed, ensuring it is semantically correct.

Visiting Delay Statements

The `visit_DelayStatementNode` method ensures the delay expression is semantically correct.

```
def visit_DelayStatementNode(self, node):
    self.visit(node.expr)
```

- **Expression**: Visits the delay expression, ensuring it is semantically correct.

Visiting Write Statements

The `visit_WriteStatementNode` method ensures all arguments in the write statement are sem

antically correct.

```
def visit_WriteStatementNode(self, node):
    for arg in node.args:
        self.visit(arg)
```

- **Arguments**: Iterates over and visits each argument in the write statement, ensuring each is semantically correct.

Visiting Binary Operations

The `visit_BinaryOpNode` method checks the types of the operands and ensures they match the requirements of the binary operator.

```
def visit_BinaryOpNode(self, node):
    left_type = self.visit(node.left)
    right_type = self.visit(node.right)
    if node.operator in {'>', '<', '>=', '<=', '==', '!='}:
        if left_type != right_type:
            raise Exception(f"Type mismatch in binary operation:
{left_type} {node.operator} {right_type}")
        return 'bool'
    elif node.operator in {'+', '-', '*', '/'}:
        if left_type != right_type:
            raise Exception(f"Type mismatch in binary operation:
{left_type} {node.operator} {right_type}")
        return left_type
    elif node.operator in {'and', 'or'}:
        if left_type != 'bool' or right_type != 'bool':
            raise Exception(f"Logical operation requires boolean operands:
{left_type} {node.operator} {right_type}")
        return 'bool'
    else:
        raise Exception(f"Unsupported binary operator: {node.operator}")
```

- **Operand Types**: Visits the left and right operands to determine their types.
- **Comparison Operators**: Checks that both operands have the same type for comparison operations and returns `bool`.
- **Arithmetic Operators**: Checks that both operands have the same type for arithmetic operations and returns the operand type.
- **Logical Operators**: Ensures both operands are `bool` for logical operations and returns `bool`.
- **Unsupported Operators**: Raises an exception if the operator is not supported.

Visiting Unary Operations

The `visit_UnaryOpNode` method checks the type of the operand and returns its type.

```
def visit_UnaryOpNode(self, node):
    operand_type = self.visit(node.operand)
    return operand_type
```

- **Operand Type**: Visits the operand to determine its type and returns it. This ensures that the unary operation is semantically correct.

Visiting Literals

The `visit_LiteralNode` method determines the type of the literal value.

```
def visit_LiteralNode(self, node):
    if isinstance(node.value, bool):
        return 'bool'
    elif isinstance(node.value, int):
        return 'int'
    elif isinstance(node.value, float):
        return 'float'
    elif isinstance(node.value, str) and node.value.startswith("#"):
        return 'colour'
    else:
        raise Exception(f"Unknown literal type: {node.value}")
```

- **Type Determination**: Checks the type of the literal value and returns the corresponding type (`bool`, `int`, `float`, `colour`).
- **Unknown Types**: Raises an exception if the literal type is unknown.

Visiting Identifiers

The `visit_IdentifierNode` method looks up the type of the identifier in the symbol table.

```
def visit_IdentifierNode(self, node):
    return self.symbol_table.lookup(node.name)
```

- **Type Lookup**: Uses the symbol table to find and return the type of the identifier. This ensures that the identifier is declared and its type is known.

Visiting Function Calls

The `visit_FunctionCallNode` method visits the arguments of the function call and returns the return type of the function.

```
def visit_FunctionCallNode(self, node):
    # For simplicity, let's assume all function calls return int.
    # This would be expanded to check the function signature.
    for arg in node.args:
        self.visit(arg)
    return 'int'
```

- **Argument Types**: Iterates over and visits each argument in the function call, ensuring each is semantically correct.
- **Return Type**: Returns the assumed return type of the function (`int`). This would be expanded to handle different function signatures.

Visiting Type Casts

The `visit_CastNode` method returns the target type of the cast.

```
def visit_CastNode(self, node):
    return node.target_type
```

- **Target Type**: Returns the target type specified in the cast. This ensures that the cast is semantically correct.

Conclusion

This semantic analyzer ensures that the program is semantically correct, adhering to the rules of the PArL language. By visiting each node in the AST and performing the necessary checks, it guarantees that variables are declared before use, types are consistent, and all semantic rules are followed.

Testing - Task 1 & 2

I tested the lexer and parser with 3 PArL programs. For each program I printed the tokens generated and showed the AST tree created.

Program 1

```
fun XGreaterY(x:int, y:int) -> bool {
    let ans:bool = true;
    if (y>x) {ans = false;}
    return ans;
}

fun XGreaterY_2(x:int, y:int) -> bool {
    return x>y;
}

fun AverageOfTwo(x:int, y:int) -> float {
    let t0:int = x + y;
    let t1:float = t0 / 2 as float;
    return t1;
}

fun AverageOfTwo_2(x:int, y:int) -> float {
    return (x + y) / 2 as float;
}

fun Max(x:int, y:int) -> int {
    let m:int = x;
    if (y > x) { m = y; }
    return m;
}

__write 10, 14, #00ff00;
__delay 100;
```

```
__write_box 10, 14, 2, 2, #0000ff;

for (let i:int = 0; i < 10; i = i + 1) {
    __print i;
    __delay 1000;
}

fun Race(p1_c:colour , p2_c:colour , score_max:int) -> int {
    let p1_score:int = 0;
    let p2_score:int = 0;
    while ((p1_score < score_max) and (p2_score < score_max)) {
        let p1_toss:int = __randi 1000;
        let p2_toss:int = __randi 1000;
        if (p1_toss > p2_toss) {
            p1_score = p1_score + 1;
            __write 1, p1_score, p1_c;
        } else {
            p2_score = p2_score + 1;
            __write 2, p2_score, p2_c;
        }
        __delay 100;
    }
    if (p2_score > p1_score) {
        return 2;
    }
    return 1;
}

let c1:colour = #00ff00;
let c2:colour = #0000ff;
let m:int = __height;
let w:int = Race(c1, c2, m);
__print w;
```

Tokens

```
('KEYWORD', 'fun')
('IDENTIFIER', 'XGreaterY')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'bool')
('DELIMITER', '{')
('KEYWORD', 'let')
```

```
('IDENTIFIER', 'ans')
('DELIMITER', ':')
('KEYWORD', 'bool')
('OPERATOR', '=')
('LITERAL', 'true')
('DELIMITER', ';')
('KEYWORD', 'if')
('DELIMITER', '(')
('IDENTIFIER', 'y')
('OPERATOR', '>')
('IDENTIFIER', 'x')
('DELIMITER', ')')
('DELIMITER', '{')
('IDENTIFIER', 'ans')
('OPERATOR', '=')
('LITERAL', 'false')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'return')
('IDENTIFIER', 'ans')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'XGreaterY_2')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'bool')
('DELIMITER', '{')
('KEYWORD', 'return')
('IDENTIFIER', 'x')
('OPERATOR', '>')
('IDENTIFIER', 'y')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'AverageOfTwo')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
```

```
('KEYWORD', 'float')
('DELIMITER', '{')
('KEYWORD', 'let')
('IDENTIFIER', 't0')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('IDENTIFIER', 'x')
('OPERATOR', '+')
('IDENTIFIER', 'y')
('DELIMITER', ';')
('KEYWORD', 'let')
('IDENTIFIER', 't1')
('DELIMITER', ':')
('KEYWORD', 'float')
('OPERATOR', '=')
('IDENTIFIER', 't0')
('OPERATOR', '/')
('LITERAL', '2')
('KEYWORD', 'as')
('KEYWORD', 'float')
('DELIMITER', ';')
('KEYWORD', 'return')
('IDENTIFIER', 't1')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'AverageOfTwo_2')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'float')
('DELIMITER', '{')
('KEYWORD', 'return')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('OPERATOR', '+')
('IDENTIFIER', 'y')
('DELIMITER', ')')
('OPERATOR', '/')
('LITERAL', '2')
('KEYWORD', 'as')
('KEYWORD', 'float')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'Max')
```

```
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'let')
('IDENTIFIER', 'm')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('IDENTIFIER', 'x')
('DELIMITER', ';')
('KEYWORD', 'if')
('DELIMITER', '(')
('IDENTIFIER', 'y')
('OPERATOR', '>')
('IDENTIFIER', 'x')
('DELIMITER', ')')
('DELIMITER', '{')
('IDENTIFIER', 'm')
('OPERATOR', '=')
('IDENTIFIER', 'y')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'return')
('IDENTIFIER', 'm')
('DELIMITER', ';')
('DELIMITER', '}')
('SPECIAL_FUNCTION', '__write')
('LITERAL', '10')
('DELIMITER', ',')
('LITERAL', '14')
('DELIMITER', ',')
('LITERAL', '#00ff00')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__delay')
('LITERAL', '100')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__write_box')
('LITERAL', '10')
('DELIMITER', ',')
('LITERAL', '14')
('DELIMITER', ',')
('LITERAL', '2')
('DELIMITER', ',')
('LITERAL', '2')
('DELIMITER', ',')
```

```
('LITERAL', '#0000ff')
('DELIMITER', ';')
('KEYWORD', 'for')
('DELIMITER', '(')
('KEYWORD', 'let')
('IDENTIFIER', 'i')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '0')
('DELIMITER', ';')
('IDENTIFIER', 'i')
('OPERATOR', '<')
('LITERAL', '10')
('DELIMITER', ';')
('IDENTIFIER', 'i')
('OPERATOR', '=')
('IDENTIFIER', 'i')
('OPERATOR', '+')
('LITERAL', '1')
('DELIMITER', ')')
('DELIMITER', '{')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'i')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__delay')
('LITERAL', '1000')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'Race')
('DELIMITER', '(')
('IDENTIFIER', 'p1_c')
('DELIMITER', ':')
('KEYWORD', 'colour')
('DELIMITER', ',')
('IDENTIFIER', 'p2_c')
('DELIMITER', ':')
('KEYWORD', 'colour')
('DELIMITER', ',')
('IDENTIFIER', 'score_max')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'let')
('IDENTIFIER', 'p1_score')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '0')
('DELIMITER', ';')
```

```
('KEYWORD', 'let')
('IDENTIFIER', 'p2_score')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '0')
('DELIMITER', ';')
('KEYWORD', 'while')
('DELIMITER', '(')
('DELIMITER', '(')
('IDENTIFIER', 'p1_score')
('OPERATOR', '<')
('IDENTIFIER', 'score_max')
('DELIMITER', ')')
('OPERATOR', 'and')
('DELIMITER', '(')
('IDENTIFIER', 'p2_score')
('OPERATOR', '<')
('IDENTIFIER', 'score_max')
('DELIMITER', ')')
('DELIMITER', ')')
('DELIMITER', '{')
('KEYWORD', 'let')
('IDENTIFIER', 'p1_toss')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('SPECIAL_FUNCTION', '__randi')
('LITERAL', '1000')
('DELIMITER', ';')
('KEYWORD', 'let')
('IDENTIFIER', 'p2_toss')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('SPECIAL_FUNCTION', '__randi')
('LITERAL', '1000')
('DELIMITER', ';')
('KEYWORD', 'if')
('DELIMITER', '(')
('IDENTIFIER', 'p1_toss')
('OPERATOR', '>')
('IDENTIFIER', 'p2_toss')
('DELIMITER', ')')
('DELIMITER', '{')
('IDENTIFIER', 'p1_score')
('OPERATOR', '=')
('IDENTIFIER', 'p1_score')
('OPERATOR', '+')
('LITERAL', '1')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__write')
('LITERAL', '1')
('DELIMITER', ',')
```

```
('IDENTIFIER', 'p1_score')
('DELIMITER', ',')
('IDENTIFIER', 'p1_c')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'else')
('DELIMITER', '{')
('IDENTIFIER', 'p2_score')
('OPERATOR', '=')
('IDENTIFIER', 'p2_score')
('OPERATOR', '+')
('LITERAL', '1')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__write')
('LITERAL', '2')
('DELIMITER', ',')
('IDENTIFIER', 'p2_score')
('DELIMITER', ',')
('IDENTIFIER', 'p2_c')
('DELIMITER', ';')
('DELIMITER', '}')
('SPECIAL_FUNCTION', '__delay')
('LITERAL', '100')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'if')
('DELIMITER', '(')
('IDENTIFIER', 'p2_score')
('OPERATOR', '>')
('IDENTIFIER', 'p1_score')
('DELIMITER', ')')
('DELIMITER', '{')
('KEYWORD', 'return')
('LITERAL', '2')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'return')
('LITERAL', '1')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'let')
('IDENTIFIER', 'c1')
('DELIMITER', ':')
('KEYWORD', 'colour')
('OPERATOR', '=')
('LITERAL', '#00ff00')
('DELIMITER', ';')
('KEYWORD', 'let')
('IDENTIFIER', 'c2')
('DELIMITER', ':')
('KEYWORD', 'colour')
('OPERATOR', '=')
('LITERAL', '#0000ff')
('DELIMITER', ';')
```

```
('KEYWORD', 'let')
('IDENTIFIER', 'm')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('SPECIAL_FUNCTION', '__height')
('DELIMITER', ';')
('KEYWORD', 'let')
('IDENTIFIER', 'w')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('IDENTIFIER', 'Race')
('DELIMITER', '(')
('IDENTIFIER', 'c1')
('DELIMITER', ',')
('IDENTIFIER', 'c2')
('DELIMITER', ',')
('IDENTIFIER', 'm')
('DELIMITER', ')')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'w')
('DELIMITER', ';')
```

AST

```
ProgramNode:
  FunctionDeclNode: XGreaterY
    Params:
      ParamNode: x: int
      ParamNode: y: int
    ReturnType: bool
    BlockNode:
      VariableDeclNode: ans: bool
        LiteralNode: True
      IfStatementNode:
        Condition:
          BinaryOpNode: >
            IdentifierNode: y
            IdentifierNode: x
        IfBlock:
          BlockNode:
            AssignmentNode: ans
              LiteralNode: False
      ReturnStatementNode:
        IdentifierNode: ans
  FunctionDeclNode: XGreaterY_2
    Params:
      ParamNode: x: int
      ParamNode: y: int
```

```
  ReturnType: bool
  BlockNode:
    ReturnStatementNode:
      BinaryOpNode: >
        IdentifierNode: x
        IdentifierNode: y
FunctionDeclNode: AverageOfTwo
  Params:
    ParamNode: x: int
    ParamNode: y: int
  ReturnType: float
  BlockNode:
    VariableDeclNode: t0: int
      BinaryOpNode: +
        IdentifierNode: x
        IdentifierNode: y
    VariableDeclNode: t1: float
      CastNode:
        Expr:
          BinaryOpNode: /
            IdentifierNode: t0
            LiteralNode: 2
        TargetType: float
    ReturnStatementNode:
      IdentifierNode: t1
FunctionDeclNode: AverageOfTwo_2
  Params:
    ParamNode: x: int
    ParamNode: y: int
  ReturnType: float
  BlockNode:
    ReturnStatementNode:
      CastNode:
        Expr:
          BinaryOpNode: /
            BinaryOpNode: +
              IdentifierNode: x
              IdentifierNode: y
            LiteralNode: 2
        TargetType: float
FunctionDeclNode: Max
  Params:
    ParamNode: x: int
    ParamNode: y: int
  ReturnType: int
  BlockNode:
    VariableDeclNode: m: int
      IdentifierNode: x
    IfStatementNode:
      Condition:
        BinaryOpNode: >
          IdentifierNode: y
          IdentifierNode: x
      IfBlock:
```

```
        BlockNode:
          AssignmentNode: m
            IdentifierNode: y
    ReturnStatementNode:
      IdentifierNode: m
WriteStatementNode:
  LiteralNode: 10
  LiteralNode: 14
  LiteralNode: #00ff00
DelayStatementNode:
  LiteralNode: 100
WriteStatementNode:
  LiteralNode: 10
  LiteralNode: 14
  LiteralNode: 2
  LiteralNode: 2
  LiteralNode: #0000ff
ForStatementNode:
  Init:
    VariableDeclNode: i: int
      LiteralNode: 0
  Condition:
    BinaryOpNode: <
      IdentifierNode: i
      LiteralNode: 10
  Post:
    AssignmentNode: i
      BinaryOpNode: +
        IdentifierNode: i
        LiteralNode: 1
  Block:
    BlockNode:
      PrintStatementNode:
        IdentifierNode: i
      DelayStatementNode:
        LiteralNode: 1000
FunctionDeclNode: Race
  Params:
    ParamNode: p1_c: colour
    ParamNode: p2_c: colour
    ParamNode: score_max: int
  ReturnType: int
  BlockNode:
    VariableDeclNode: p1_score: int
      LiteralNode: 0
    VariableDeclNode: p2_score: int
      LiteralNode: 0
    WhileStatementNode:
      Condition:
        BinaryOpNode: and
          BinaryOpNode: <
            IdentifierNode: p1_score
            IdentifierNode: score_max
          BinaryOpNode: <
```

```
            IdentifierNode: p2_score
            IdentifierNode: score_max
        Block:
          BlockNode:
            VariableDeclNode: p1_toss: int
              FunctionCallNode: __randi
                LiteralNode: 1000
            VariableDeclNode: p2_toss: int
              FunctionCallNode: __randi
                LiteralNode: 1000
            IfStatementNode:
              Condition:
                BinaryOpNode: >
                  IdentifierNode: p1_toss
                  IdentifierNode: p2_toss
              IfBlock:
                BlockNode:
                  AssignmentNode: p1_score
                    BinaryOpNode: +
                      IdentifierNode: p1_score
                      LiteralNode: 1
                  WriteStatementNode:
                    LiteralNode: 1
                    IdentifierNode: p1_score
                    IdentifierNode: p1_c
              ElseBlock:
                BlockNode:
                  AssignmentNode: p2_score
                    BinaryOpNode: +
                      IdentifierNode: p2_score
                      LiteralNode: 1
                  WriteStatementNode:
                    LiteralNode: 2
                    IdentifierNode: p2_score
                    IdentifierNode: p2_c
            DelayStatementNode:
              LiteralNode: 100
      IfStatementNode:
        Condition:
          BinaryOpNode: >
            IdentifierNode: p2_score
            IdentifierNode: p1_score
        IfBlock:
          BlockNode:
            ReturnStatementNode:
              LiteralNode: 2
      ReturnStatementNode:
        LiteralNode: 1
  VariableDeclNode: c1: colour
    LiteralNode: #00ff00
  VariableDeclNode: c2: colour
    LiteralNode: #0000ff
  VariableDeclNode: m: int
    FunctionCallNode: __height
```

```
  VariableDeclNode: w: int
    FunctionCallNode: Race
      IdentifierNode: c1
      IdentifierNode: c2
      IdentifierNode: m
  PrintStatementNode:
    IdentifierNode: w
```

Program 2

```
fun SumOfTwo(x:int, y:int) -> int {
    return x + y;
}

fun ProductOfTwo(x:int, y:int) -> int {
    return x * y;
}

__print SumOfTwo(10, 20);
__delay 50;
__print ProductOfTwo(5, 6);

let counter:int = 0;
while (counter < 5) {
    __print counter;
    counter = counter + 1;
}

fun PowerOfTwo(n:int) -> int {
    let result:int = 1;
    for (let i:int = 0; i < n; i = i + 1) {
        result = result * 2;
    }
    return result;
}

__print PowerOfTwo(4);
```

Tokens

```
('KEYWORD', 'fun')
('IDENTIFIER', 'SumOfTwo')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
```

```
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'return')
('IDENTIFIER', 'x')
('OPERATOR', '+')
('IDENTIFIER', 'y')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'ProductOfTwo')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'return')
('IDENTIFIER', 'x')
('OPERATOR', '*')
('IDENTIFIER', 'y')
('DELIMITER', ';')
('DELIMITER', '}')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'SumOfTwo')
('DELIMITER', '(')
('LITERAL', '10')
('DELIMITER', ',')
('LITERAL', '20')
('DELIMITER', ')')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__delay')
('LITERAL', '50')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'ProductOfTwo')
('DELIMITER', '(')
('LITERAL', '5')
('DELIMITER', ',')
('LITERAL', '6')
('DELIMITER', ')')
('DELIMITER', ';')
('KEYWORD', 'let')
('IDENTIFIER', 'counter')
('DELIMITER', ':')
```

```
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '0')
('DELIMITER', ';')
('KEYWORD', 'while')
('DELIMITER', '(')
('IDENTIFIER', 'counter')
('OPERATOR', '<')
('LITERAL', '5')
('DELIMITER', ')')
('DELIMITER', '{')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'counter')
('DELIMITER', ';')
('IDENTIFIER', 'counter')
('OPERATOR', '=')
('IDENTIFIER', 'counter')
('OPERATOR', '+')
('LITERAL', '1')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'PowerOfTwo')
('DELIMITER', '(')
('IDENTIFIER', 'n')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'let')
('IDENTIFIER', 'result')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '1')
('DELIMITER', ';')
('KEYWORD', 'for')
('DELIMITER', '(')
('KEYWORD', 'let')
('IDENTIFIER', 'i')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '0')
('DELIMITER', ';')
('IDENTIFIER', 'i')
('OPERATOR', '<')
('IDENTIFIER', 'n')
('DELIMITER', ';')
('IDENTIFIER', 'i')
('OPERATOR', '=')
('IDENTIFIER', 'i')
```

```
('OPERATOR', '+')
('LITERAL', '1')
('DELIMITER', ')')
('DELIMITER', '{')
('IDENTIFIER', 'result')
('OPERATOR', '=')
('IDENTIFIER', 'result')
('OPERATOR', '*')
('LITERAL', '2')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'return')
('IDENTIFIER', 'result')
('DELIMITER', ';')
('DELIMITER', '}')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'PowerOfTwo')
('DELIMITER', '(')
('LITERAL', '4')
('DELIMITER', ')')
('DELIMITER', ';')
```

AST

```
ProgramNode:
  FunctionDeclNode: SumOfTwo
    Params:
      ParamNode: x: int
      ParamNode: y: int
    ReturnType: int
    BlockNode:
      ReturnStatementNode:
        BinaryOpNode: +
          IdentifierNode: x
          IdentifierNode: y
  FunctionDeclNode: ProductOfTwo
    Params:
      ParamNode: x: int
      ParamNode: y: int
    ReturnType: int
    BlockNode:
      ReturnStatementNode:
        BinaryOpNode: *
          IdentifierNode: x
          IdentifierNode: y
  PrintStatementNode:
    FunctionCallNode: SumOfTwo
      LiteralNode: 10
      LiteralNode: 20
  DelayStatementNode:
    LiteralNode: 50
```

```
PrintStatementNode:
  FunctionCallNode: ProductOfTwo
    LiteralNode: 5
    LiteralNode: 6
VariableDeclNode: counter: int
  LiteralNode: 0
WhileStatementNode:
  Condition:
    BinaryOpNode: <
      IdentifierNode: counter
      LiteralNode: 5
  Block:
    BlockNode:
      PrintStatementNode:
        IdentifierNode: counter
      AssignmentNode: counter
        BinaryOpNode: +
          IdentifierNode: counter
          LiteralNode: 1
FunctionDeclNode: PowerOfTwo
  Params:
    ParamNode: n: int
  ReturnType: int
  BlockNode:
    VariableDeclNode: result: int
      LiteralNode: 1
    ForStatementNode:
      Init:
        VariableDeclNode: i: int
          LiteralNode: 0
      Condition:
        BinaryOpNode: <
          IdentifierNode: i
          IdentifierNode: n
      Post:
        AssignmentNode: i
          BinaryOpNode: +
            IdentifierNode: i
            LiteralNode: 1
      Block:
        BlockNode:
          AssignmentNode: result
            BinaryOpNode: *
              IdentifierNode: result
              LiteralNode: 2
    ReturnStatementNode:
      IdentifierNode: result
PrintStatementNode:
  FunctionCallNode: PowerOfTwo
    LiteralNode: 4
```

Program 3

```
fun Min(x:int, y:int) -> int {
    if (x < y) {
        return x;
    }
    return y;
}

fun Square(x:int) -> int {
    return x * x;
}

__write 5, 10, #ff0000;
__delay 75;
__write_box 5, 10, 3, 3, #00ff00;

for (let j:int = 0; j < 7; j = j + 1) {
    __print j;
    __delay 500;
}

fun Factorial(n:int) -> int {
    let fact:int = 1;
    for (let k:int = 1; k <= n; k = k + 1) {
        fact = fact * k;
    }
    return fact;
}

__print Factorial(5);
```

Tokens

```
('KEYWORD', 'fun')
('IDENTIFIER', 'Min')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ',')
('IDENTIFIER', 'y')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'if')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('OPERATOR', '<')
```

```
('IDENTIFIER', 'y')
('DELIMITER', ')')
('DELIMITER', '{')
('KEYWORD', 'return')
('IDENTIFIER', 'x')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'return')
('IDENTIFIER', 'y')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'Square')
('DELIMITER', '(')
('IDENTIFIER', 'x')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'return')
('IDENTIFIER', 'x')
('OPERATOR', '*')
('IDENTIFIER', 'x')
('DELIMITER', ';')
('DELIMITER', '}')
('SPECIAL_FUNCTION', '__write')
('LITERAL', '5')
('DELIMITER', ',')
('LITERAL', '10')
('DELIMITER', ',')
('LITERAL', '#ff0000')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__delay')
('LITERAL', '75')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__write_box')
('LITERAL', '5')
('DELIMITER', ',')
('LITERAL', '10')
('DELIMITER', ',')
('LITERAL', '3')
('DELIMITER', ',')
('LITERAL', '3')
('DELIMITER', ',')
('LITERAL', '#00ff00')
('DELIMITER', ';')
('KEYWORD', 'for')
('DELIMITER', '(')
('KEYWORD', 'let')
('IDENTIFIER', 'j')
('DELIMITER', ':')
('KEYWORD', 'int')
```

```
('OPERATOR', '=')
('LITERAL', '0')
('DELIMITER', ';')
('IDENTIFIER', 'j')
('OPERATOR', '<')
('LITERAL', '7')
('DELIMITER', ';')
('IDENTIFIER', 'j')
('OPERATOR', '=')
('IDENTIFIER', 'j')
('OPERATOR', '+')
('LITERAL', '1')
('DELIMITER', ')')
('DELIMITER', '{')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'j')
('DELIMITER', ';')
('SPECIAL_FUNCTION', '__delay')
('LITERAL', '500')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'fun')
('IDENTIFIER', 'Factorial')
('DELIMITER', '(')
('IDENTIFIER', 'n')
('DELIMITER', ':')
('KEYWORD', 'int')
('DELIMITER', ')')
('ARROW', '->')
('KEYWORD', 'int')
('DELIMITER', '{')
('KEYWORD', 'let')
('IDENTIFIER', 'fact')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '1')
('DELIMITER', ';')
('KEYWORD', 'for')
('DELIMITER', '(')
('KEYWORD', 'let')
('IDENTIFIER', 'k')
('DELIMITER', ':')
('KEYWORD', 'int')
('OPERATOR', '=')
('LITERAL', '1')
('DELIMITER', ';')
('IDENTIFIER', 'k')
('OPERATOR', '<=')
('IDENTIFIER', 'n')
('DELIMITER', ';')
('IDENTIFIER', 'k')
('OPERATOR', '=')
('IDENTIFIER', 'k')
```

```
('OPERATOR', '+')
('LITERAL', '1')
('DELIMITER', ')')
('DELIMITER', '{')
('IDENTIFIER', 'fact')
('OPERATOR', '=')
('IDENTIFIER', 'fact')
('OPERATOR', '*')
('IDENTIFIER', 'k')
('DELIMITER', ';')
('DELIMITER', '}')
('KEYWORD', 'return')
('IDENTIFIER', 'fact')
('DELIMITER', ';')
('DELIMITER', '}')
('SPECIAL_FUNCTION', '__print')
('IDENTIFIER', 'Factorial')
('DELIMITER', '(')
('LITERAL', '5')
('DELIMITER', ')')
('DELIMITER', ';')
```

AST

```
ProgramNode:
  FunctionDeclNode: Min
    Params:
      ParamNode: x: int
      ParamNode: y: int
    ReturnType: int
    BlockNode:
      IfStatementNode:
        Condition:
          BinaryOpNode: <
            IdentifierNode: x
            IdentifierNode: y
        IfBlock:
          BlockNode:
            ReturnStatementNode:
              IdentifierNode: x
      ReturnStatementNode:
        IdentifierNode: y
  FunctionDeclNode: Square
    Params:
      ParamNode: x: int
    ReturnType: int
    BlockNode:
      ReturnStatementNode:
        BinaryOpNode: *
          IdentifierNode: x
          IdentifierNode: x
```

```
WriteStatementNode:
  LiteralNode: 5
  LiteralNode: 10
  LiteralNode: #ff0000
DelayStatementNode:
  LiteralNode: 75
WriteStatementNode:
  LiteralNode: 5
  LiteralNode: 10
  LiteralNode: 3
  LiteralNode: 3
  LiteralNode: #00ff00
ForStatementNode:
  Init:
    VariableDeclNode: j: int
      LiteralNode: 0
  Condition:
    BinaryOpNode: <
      IdentifierNode: j
      LiteralNode: 7
  Post:
    AssignmentNode: j
      BinaryOpNode: +
        IdentifierNode: j
        LiteralNode: 1
  Block:
    BlockNode:
      PrintStatementNode:
        IdentifierNode: j
      DelayStatementNode:
        LiteralNode: 500
FunctionDeclNode: Factorial
  Params:
    ParamNode: n: int
  ReturnType: int
  BlockNode:
    VariableDeclNode: fact: int
      LiteralNode: 1
    ForStatementNode:
      Init:
        VariableDeclNode: k: int
          LiteralNode: 1
      Condition:
        BinaryOpNode: <=
          IdentifierNode: k
          IdentifierNode: n
      Post:
        AssignmentNode: k
          BinaryOpNode: +
            IdentifierNode: k
            LiteralNode: 1
      Block:
        BlockNode:
          AssignmentNode: fact
```

```
              BinaryOpNode: *
                IdentifierNode: fact
                IdentifierNode: k
      ReturnStatementNode:
        IdentifierNode: fact
  PrintStatementNode:
    FunctionCallNode: Factorial
      LiteralNode: 5
```

Testing - Task 3

I also tested the semantic analysis section with the 3 correclty formatted programs, aswell as 3 incorrectly formatted program sections.

Results

The semantic analyser didn't raise any errors for the above 3 programs.

Semantic Test 1

```
fun MoreThan50(x:int) -> bool {
    let x:int = 23;
    if (x <= 50) {
        return false;
    }
    return true;
}
```

In this section, the analyzer should find that x is already declared in the current scope (as a function parameter) and raise an exception. However it fails to do this. This is a bug in the system that I did not have time to fix.

Semantic Test 2

```
let x:int = 45;
while (x < 50) {
    __print MoreThan50(x);
    x = x + 1;
}

let x:int = 45;
while (MoreThan50(x)) {
    __print MoreThan50(x);
    x = x + 1;
}
```

In this section the analyser successfully raises the following exception:

```
Exception: Variable 'x' already declared in the same scope
```

Semantic Test 3

```
fun AverageOfTwo(x:int, y:int) -> int {
    let t0:int = x + y;
    let t1:float = t0 / 2 as float;
    return t1;
}
```

In this section the analyser successfully raises the following exception:

```
Exception: Return type mismatch in function with return type int: got
float
```

Video Link

The video showcasing the sections running and the results obtained may be found below:

https://drive.google.com/file/d/125zxnLoYGHoOL3WC_il4a_nuk28W-DT4/view?usp=sharing