CS 3513: Programming Languages Programming Project - 01

Group: Byte Buddies

220131A: disanayaka D.M.D.M.
220708B: Wijekoon S.W.G.M.Y.G.D.M.

1. Introduction

In this project, we were required to develop a comprehensive system capable of interpreting and executing programs written in the RPAL programming language. We implemented a lexical analyzer and parser for the RPAL language. The parser generates an Abstract Syntax Tree (AST) for the input program. Subsequently, we developed a system that converts the Abstract Syntax Tree (AST) into a Standardized Tree (ST) and implemented a CSE machine to enable program execution with input processing.

The program focuses on the following basic stages:

- Lexical analyzer
- Parser
- Standardizer
- CSE Machine

2. Implementation

2.1 Language and Tools Used

This project was implemented using the Python programming language and does not require any additional libraries or packages to be installed before execution.

2.2 Lexical Analyzer

The Lexical Analyzer serves as the initial component of the program's front-end processing system. It scans the source code and breaks it down into individual tokens while identifying the underlying grammatical structures. This preprocessing step is essential before any subsequent analysis can occur. The Lexical Analyzer establishes the foundation for later stages in the processing pipeline by recognizing keywords, identifiers, constants, and other language-specific elements. The lexical analyzer adheres to the specifications outlined in the

RPAL_Lex.pdf document. It converts the input RPAL program into lexemes, which are subsequently forwarded to the parser for further processing.

2.3 Parser

The Parser takes the role in program execution following the preliminary work conducted by the Lexical Analyzer. The parser constructs the Abstract Syntax Tree (AST) from the lexemes produced by the lexical analyzer, adhering to the grammar specifications defined in RPAL_Grammar.pdf. For this RPAL Interpreter implementation, we implemented the parsing algorithm manually.

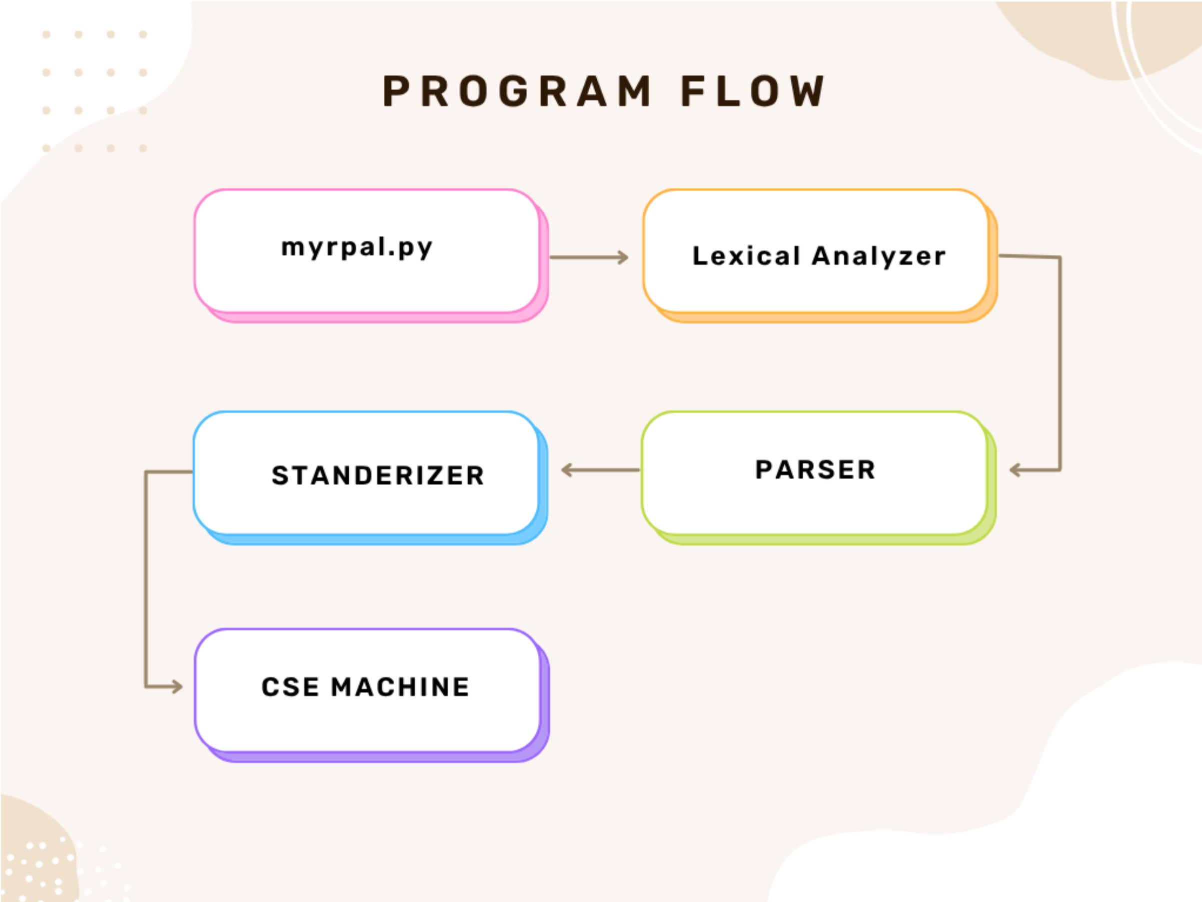
2.4 Standardizer

Following the construction of the AST, we implemented an algorithm to transform it into a Standardized Tree (ST). The ST represents the program in a format that is optimized for execution by the CSE machine.

2.5 CSE Machine

The Control Structure Evaluation (CSE) Machine serves as the final and most critical component of the interpreter system. It utilizes the processed Standardized Tree to efficiently interpret and execute the code. The CSE machine employs various computational techniques and optimizations to accurately execute the program and generate the required output. By adhering to RPAL-specific semantics and syntax rules, the CSE machine ensures consistent and reliable program execution. We implemented the CSE (Control Structure Evaluation) machine to execute RPAL programs represented by the ST. The CSE machine evaluates the program by systematically traversing the ST and performing the necessary computational operations.

3. Program Structure



3.1 Lexical Analyzer

- Input: The tokenize function expects a string as input, representing the program code to be tokenized.

- Output: It produces a list of MyToken objects, where each token contains information about its type (enumerated by TokenType) and value.

- Parameter Passing: The input string is passed to the tokenize function by value, ensuring that the function operates on a copy of the original string.

- ErrorHandling: The tokenize function handles errors by printing an error message if it encounters an unrecognized token or fails to tokenize the input string.

- ReturnValues: The return value of the tokenize function is a list of tokens, allowing the calling code to iterate over the tokens and process them further.

```
import re
```

```python
from enum import Enum


class TokenType(Enum):
    KEYWORD = 1
    ID = 2
    INT = 3
    STRING = 4
    END_OF_TOKENS = 5
    PUNCTUATION = 6
    OPERATOR = 7


# Token class to represent a single token
class Token:
    def __init__(self, token_type, value):
        if not isinstance(token_type, TokenType):
            raise ValueError("token_type must be an instance of TokenType enum")
        self.type = token_type
        self.value = value

    # Getters for type and value
    def get_type(self):
        return self.type
```

```
    def get_value(self):
        return self.value




def tokenize(input_str):
    tokens = []
    keywords = {
        # Single-line comment: starts with // and goes until the end of the line
        'COMMENT': r'//.*',
        # Keywords: exact matches for reserved words, ending on a word boundary
        'KEYWORD': r'(let|in|fn|where|aug|or|not|gr|ge|ls|le|eq|ne|true|false|nil|dummy|within|and|rec)\b',
        # String literals enclosed in single quotes, allowing escaped single quotes
        'STRING': r'\'(?:\\\'|[^\'])*\'',
        # Identifiers: start with a letter, followed by letters, digits, or underscores
        'ID': r'[a-zA-Z][a-zA-Z0-9_]*',
        # Integer literals: one or more digits
        'INT': r'\d+',
        # Operators: one or more of the specified symbols
        'OPERATOR': r'[+\-*<>&.@/:=~|$\#!%^_\[\]{}"\'?]+',
```

```
    # Whitespace: one or more spaces, tabs, or newlines
    'SPACES': r'[ \t\n]+',
    # Punctuation characters
    'PUNCTUATION': r'[();,]'
}

while input_str:
    # Flag to check if a pattern matched
    matched = False
    for key, pattern in keywords.items():
        match = re.match(pattern, input_str)
        if match:
            # print(key, match.group(0))
            if key != 'SPACES':
                if key == 'COMMENT':
                    # If it's a comment, remove the comment from input
                    comment = match.group(0)
                    input_str = input_str[match.end():]
                    matched = True
                    break
                else:
                    # Get the corresponding TokenType
                    token_type = getattr(TokenType, key)
                    if not isinstance(token_type, TokenType):
```

```
                            raise ValueError(f"Token type '{key}' is not a valid TokenType")

                        # Create and add the token to the list

                        tokens.append(Token(token_type, match.group(0)))

                        input_str = input_str[match.end():]

                        matched = True

                        break

                input_str = input_str[match.end():]

                matched = True

                break

        if not matched:

            print("Error: Unable to tokenize input")

    return tokens
```

3.2 Parser Data Formatting

- Input: The parse method takes a list of tokens as input, representing the lexemes generated by the lexical analyzer.

- Output: It produces an Abstract Syntax Tree (AST) representing the parsed program.

- Parameter Passing: The list of tokens is passed to the parse method by reference, allowing the method to directly manipulate the list during parsing.

- ErrorHandling: If parsing is unsuccessful, the parse method prints an error message and returns None, indicating failure.

- ReturnValues: The return value of the parse method is the constructed AST, enabling the calling code to further process or analyze the parsed program.

```
from enum import Enum

from Lexer.lexical_analyzer import TokenType, Token


# Enumeration defining all possible node types in the Abstract Syntax Tree
(AST)

# Each node type represents a different language construct or operation

class NodeType(Enum):

    let = 1              # Let expression binding

    fcn_form = 2         # Function form definition

    id = 3               # Identifier

    int = 4              # Integer literal

    str = 5           # String literal

    where = 6            # Where clause

    gamma = 7            # Function application

    lambda_expr = 8      # Lambda expression (anonymous function)
```

```
tau = 9                 # Tuple construction

rec = 10                # Recursive definition

aug = 11                # Augmentation operator

conditional = 12        # Conditional expression (if-then-else)

op_or = 13              # Logical OR operator

op_and = 14             # Logical AND operator

op_not = 15             # Logical NOT operator

op_compare = 16         # Comparison operators (>, <, =, etc.)

op_plus = 17            # Addition operator

op_minus = 18           # Subtraction operator

op_neg = 19             # Unary negation operator

op_mul = 20             # Multiplication operator

op_div = 21             # Division operator

op_pow = 22             # Power/exponentiation operator

at = 23                 # At operator (@)

true_value = 24         # Boolean true literal

false_value = 25        # Boolean false literal

nil = 26                # Nil/null value

dummy = 27              # Dummy value

within = 28             # Within clause

and_op = 29             # And operator for definitions

equal = 30              # Assignment/equality operator

comma = 31              # Comma operator for lists

empty_params = 32       # Empty parameter list ()
```

```python
# AST Node class representing a single node in the syntax tree
class Node:
    def __init__(self, node_type, value, children):
        self.type = node_type
        self.value = value
        self.no_of_children = children


# Main Parser class implementing a recursive descent parser
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.ast = []              # Stack-based AST construction (nodes in reverse order)
        self.string_ast = []       # String representation of the AST

    def peek_token(self):
        """Safely peek at the current token without consuming it"""
        if self.tokens:
            return self.tokens[0]
        return None

    def consume_token(self):
        """Safely consume and return the current token"""
```

```
        if self.tokens:
            return self.tokens.pop(0)
        return None

    def parse(self):
        """Main parsing entry point - parses the entire token stream"""
        # Add end-of-tokens marker to simplify parsing logic
        self.tokens.append(Token(TokenType.END_OF_TOKENS, ""))

        # Start parsing from the top-level expression rule
        self.E()

        # Check if all tokens were consumed successfully
        if self.tokens[0].type == TokenType.END_OF_TOKENS:
            return self.ast
        else:
            # Print error information if parsing failed
            print("Parsing Unsuccessful!...........")
            print("REMAINING UNPARSED TOKENS:")
            for token in self.tokens:
                print("<" + str(token.type) + ", " + token.value + ">")
            return None

    def convert_ast_to_string_ast(self):
```

```
        """Convert the stack-based AST to a readable string representation
with dot notation"""

        dots = ""      # Tracks indentation level

        stack = []     # Stack for managing node processing


        # Process nodes from the AST stack

        while self.ast:

            if not stack:

                # Base case: process leaf nodes directly

                if self.ast[-1].no_of_children == 0:

                    self.add_strings(dots, self.ast.pop())

                else:

                    # Non-leaf node: add to processing stack

                    node = self.ast.pop()

                    stack.append(node)

            else:

                if self.ast[-1].no_of_children > 0:

                    # Internal node: add to stack and increase indentation

                    node = self.ast.pop()

                    stack.append(node)

                    dots += "."

                else:

                    # Leaf node: add to stack and increase indentation

                    stack.append(self.ast.pop())
```

```
                    dots += "."


                    # Process completed nodes from stack
                    while stack[-1].no_of_children == 0:
                        self.add_strings(dots, stack.pop())
                        if not stack:
                            break
                        # Backtrack: reduce indentation and decrement parent's
child count
                        dots = dots[:-1]
                        node = stack.pop()
                        node.no_of_children -= 1
                        stack.append(node)


        # Reverse to get correct order for display
        self.string_ast.reverse()
        return self.string_ast


    def add_strings(self, dots, node):
        """Add formatted string representation of a node to string_ast"""
        # Special formatting for nodes with values
        if node.type in [NodeType.id, NodeType.int, NodeType.str,
NodeType.true_value,
                         NodeType.false_value, NodeType.nil, NodeType.dummy]:
```

```
            self.string_ast.append(dots + "<" + node.type.name.upper() + ":" +
node.value + ">")

        elif node.type == NodeType.fcn_form:

            # Special case for function form

            self.string_ast.append(dots + "function_form")

        else:

            # Standard formatting using node value

            self.string_ast.append(dots + node.value)


    # ===============================

    # EXPRESSION PARSING METHODS

    # ===============================


    # E -> 'let' D 'in' E         => 'let'      (Let expression)

    #   -> 'fn' Vb+ '.' E         => 'lambda'   (Lambda/function expression)

    #   -> Ew;                                  (Where expression)

    def E(self):

        """Parse top-level expressions: let bindings, lambda expressions, or
where expressions"""

        token = self.peek_token()

        if not token:

            print("Parse error: Unexpected end of input in E")

            return


        if token.type == TokenType.KEYWORD and token.value == "let":
```

```
            # Parse let expression: let D in E
            self.consume_token()  # Remove "let"
            self.D()              # Parse definitions

            # Expect 'in' keyword
            in_token = self.peek_token()
            if not in_token or in_token.value != "in":
                print("Parse error at E: 'in' expected")
                return
            self.consume_token()  # Remove "in"

            self.E()              # Parse body expression
            self.ast.append(Node(NodeType.let, "let", 2))

        elif token.type == TokenType.KEYWORD and token.value == "fn":
            # Parse lambda expression: fn Vb+ . E
            self.consume_token()  # Remove "fn"
            n = 0  # Count parameter bindings

            # Ensure at least one variable binding exists
            next_token = self.peek_token()
            if not next_token or (next_token.type != TokenType.ID and
next_token.value != "("):
                print("Parse error at E: At least one variable binding
expected after 'fn'")
```

```
                return

            # Parse one or more variable bindings
            while self.peek_token() and (self.peek_token().type ==
TokenType.ID or self.peek_token().value == "("):
                self.Vb()
                n += 1

            # Expect dot separator
            dot_token = self.peek_token()
            if not dot_token or dot_token.value != ".":
                print("Parse error at E: '.' expected after variable
bindings")
                return
            self.consume_token()  # Remove "."

            self.E()  # Parse function body
            self.ast.append(Node(NodeType.lambda_expr, "lambda", n + 1))
        else:
            # Default case: parse where expression
            self.Ew()

    # Ew -> T 'where' Dr          => 'where'    (Where clause)
    #    -> T;                                  (Simple tuple expression)
    def Ew(self):
```

```python
        """Parse where expressions or simple tuple expressions"""
        self.T()  # Parse tuple expression
        where_token = self.peek_token()
        if where_token and where_token.value == "where":
            self.consume_token()  # Remove "where"
            self.Dr()             # Parse recursive definitions
            self.ast.append(Node(NodeType.where, "where", 2))

    # ================================
    # TUPLE EXPRESSION PARSING
    # ================================

    # T -> Ta ( ',' Ta )+         => 'tau'      (Multi-element tuple)
    #   -> Ta ;                                 (Single element)
    def T(self):
        """Parse tuple expressions (comma-separated values)"""
        self.Ta()  # Parse first element
        n = 1      # Count elements

        # Parse additional comma-separated elements
        while self.peek_token() and self.peek_token().value == ",":
            self.consume_token()  # Remove comma(,)
```

This is a part of the code. The complete codebase is attached herewith.

3.3 AST to ST Conversion (Standardizer)

● Input: After converting the nodes that were returned by the parser to another set of nodes which have more attributes and methods (This is done by “ASTFactory”), the root node is pushed to the standardize function.

● Output: This produces the standardized AST as a set of nodes. Parameter Passing: The root node, which was returned by the ASTFactory.

● ReturnValues: Constructed standard AST object

```
    def standardize(self):

        if not self.is_standardized:

            for child in self.children:

                child.standardize()


            if self.data == "let":

                # Standardize LET node

                #        LET              GAMMA

                #      /     \           /     \

                #    EQUAL    P   ->   LAMBDA    E

                #   /   \             /    \

                #  X     E           X      P


                temp1 = self.children[0].children[1]

                temp1.set_parent(self)

                temp1.set_depth(self.depth + 1)

                temp2 = self.children[1]

                temp2.set_parent(self.children[0])
```

```
            temp2.set_depth(self.depth + 2)

            self.children[1] = temp1

            self.children[0].set_data("lambda")

            self.children[0].children[1] = temp2

            self.set_data("gamma")

        elif self.data == "where":

            #       WHERE               LET

            #       /   \             /     \

            #      P     EQUAL    ->  EQUAL    P

            #           /   \        /   \

            #          X     E      X     E


            temp = self.children[0]

            self.children[0] = self.children[1]

            self.children[1] = temp

            self.set_data("let")

            self.standardize()

        elif self.data == "function_form":


            #       FCN_FORM                EQUAL

            #       /   |   \              /     \

            #      P    V+   E    ->      P     +LAMBDA

            #                                   /     \

            #                                  V      .E
```

```
                Ex = self.children[-1]
                current_lambda = NodeFactory.get_node_with_parent("lambda",
self.depth + 1, self, [], True)
                self.children.insert(1, current_lambda)

                i = 2
                while self.children[i] != Ex:
                    V = self.children[i]
                    self.children.pop(i)
                    V.set_depth(current_lambda.depth + 1)
                    V.set_parent(current_lambda)
                    current_lambda.children.append(V)

                    if len(self.children) > 3:
                        current_lambda =
NodeFactory.get_node_with_parent("lambda", current_lambda.depth + 1,
current_lambda, [], True)

current_lambda.get_parent().children.append(current_lambda)

                current_lambda.children.append(Ex)
                self.children.pop(2)
                self.set_data("=")
            elif self.data == "lambda":
```

```
                #     LAMBDA        LAMBDA

                #      /   \    ->   /    \

                #     V++    E      V      .E


                if len(self.children) > 2:

                    Ey = self.children[-1]

                    current_lambda =
NodeFactory.get_node_with_parent("lambda", self.depth + 1, self, [], True)

                    self.children.insert(1, current_lambda)


                    i = 2

                    while self.children[i] != Ey:

                        V = self.children[i]

                        self.children.pop(i)

                        V.set_depth(current_lambda.depth + 1)

                        V.set_parent(current_lambda)

                        current_lambda.children.append(V)


                        if len(self.children) > 3:

                            current_lambda =
NodeFactory.get_node_with_parent("lambda", current_lambda.depth + 1,
current_lambda, [], True)


current_lambda.get_parent().children.append(current_lambda)
```

```
                current_lambda.children.append(Ey)

                self.children.pop(2)

        elif self.data == "within":


            #          WITHIN                   EQUAL

            #         /      \                 /     \

            #       EQUAL    EQUAL     ->     X2      GAMMA

            #      /    \   /    \                   /    \

            #     X1     E1 X2     E2                LAMBDA   E1

            #                                       /     \

            #                                      X1      E2


            X1 = self.children[0].children[0]

            X2 = self.children[1].children[0]

            E1 = self.children[0].children[1]

            E2 = self.children[1].children[1]

            gamma = NodeFactory.get_node_with_parent("gamma", self.depth +
1, self, [], True)

            lambda_ = NodeFactory.get_node_with_parent("lambda",
self.depth + 2, gamma, [], True)

            X1.set_depth(X1.get_depth() + 1)

            X1.set_parent(lambda_)

            X2.set_depth(X1.get_depth() - 1)

            X2.set_parent(self)

            E1.set_depth(E1.get_depth())
```

```
                E1.set_parent(gamma)

                E2.set_depth(E2.get_depth() + 1)

                E2.set_parent(lambda_)

                lambda_.children.append(X1)

                lambda_.children.append(E2)

                gamma.children.append(lambda_)

                gamma.children.append(E1)

                self.children.clear()

                self.children.append(X2)

                self.children.append(gamma)

                self.set_data("=")

            elif self.data == "@":


                #          AT                GAMMA

                #        / | \     ->        /    \

                #      E1  N  E2           GAMMA    E2

                #                          /    \

                #                         N      E1


                gamma1 = NodeFactory.get_node_with_parent("gamma", self.depth
+ 1, self, [], True)

                e1 = self.children[0]

                e1.set_depth(e1.get_depth() + 1)

                e1.set_parent(gamma1)
```

```
                n = self.children[1]

                n.set_depth(n.get_depth() + 1)

                n.set_parent(gamma1)

                gamma1.children.append(n)

                gamma1.children.append(e1)

                self.children.pop(0)

                self.children.pop(0)

                self.children.insert(0, gamma1)

                self.set_data("gamma")

            elif self.data == "and":


                #         SIMULTDEF              EQUAL

                #              |               /     \

                #           EQUAL++  ->      COMMA    TAU

                #           /   \              |       |

                #          X     E            X++     E++


                comma = NodeFactory.get_node_with_parent(",", self.depth + 1,
self, [], True)

                tau = NodeFactory.get_node_with_parent("tau", self.depth + 1,
self, [], True)


                for equal in self.children:

                    equal.children[0].set_parent(comma)

                    equal.children[1].set_parent(tau)
```

```
                    comma.children.append(equal.children[0])

                    tau.children.append(equal.children[1])


                self.children.clear()

                self.children.append(comma)

                self.children.append(tau)

                self.set_data("=")

            elif self.data == "rec":


                #         REC                   EQUAL

                #          |                   /     \

                #        EQUAL      ->        X      GAMMA

                #       /     \                     /    \

                #      X       E                 YSTAR   LAMBDA

                #                                       /     \

                #                                      X       E


                X = self.children[0].children[0]

                E = self.children[0].children[1]

                F = NodeFactory.get_node_with_parent(X.get_data(), self.depth
+ 1, self, X.children, True)

                G = NodeFactory.get_node_with_parent("gamma", self.depth + 1,
self, [], True)

                Y = NodeFactory.get_node_with_parent("<Y*>", self.depth + 2,
G, [], True)
```

```
                L = NodeFactory.get_node_with_parent("lambda", self.depth + 2,
G, [], True)


                X.set_depth(L.depth + 1)

                X.set_parent(L)

                E.set_depth(L.depth + 1)

                E.set_parent(L)

                L.children.append(X)

                L.children.append(E)
```

This is a part of the code. The complete codebase is attached herewith.

3.4 CSE Machine

- Input: This takes control, stack, and environment, which were returned by the "get_cse_machine," which takes the standardized AST as the input.

- Output: It produces the final output result for the RPAL program code.

- Parameter Passing:

  - Control Stack: The control attribute represents the control stack of the CSE machine, containing symbols and instructions to be executed.

  - Stack: The stack attribute represents the stack of the CSE machine, used for storing intermediate results during execution.

  - Environment: The environment attribute represents the environment of the CSE machine, containing bindings between identifiers and their values.

- ReturnValues: The get_answer method returns the final result after executing the CSE machine.

```
from .nodes import *
```

```python
class CSEMachine:

    def __init__(self, control, stack, environment):

        self.control = control

        self.stack = stack

        self.environment = environment


    def execute(self):

        # Execute the CSEMachine

        current_environment = self.environment[0]

        j = 1

        while self.control:


            # change below paths to your own paths to see how the control and stack are changing

            # self.write_control_to_file("C:\\Users\\samar\\Desktop\\PL_Project\\CSE Evaluation\\Control.txt")

            # self.write_stack_to_file("C:\\Users\\samar\\Desktop\\PL_Project\\CSE Evaluation\\Stack.txt")


            current_symbol = self.control.pop()

            if isinstance(current_symbol, Id):

                self.stack.insert(0, current_environment.lookup(current_symbol))

                # print(current_environment.lookup(current_symbol).get_data())
```

```
            elif isinstance(current_symbol, Lambda):

current_symbol.set_environment(current_environment.get_index())
                self.stack.insert(0, current_symbol)



            elif isinstance(current_symbol, Gamma):
                next_symbol = self.stack.pop(0)
                if isinstance(next_symbol, Lambda):
                    # Handle Lambda expression
                    lambda_expr = next_symbol
                    e = E(j)
                    j += 1
                    if len(lambda_expr.identifiers) == 1:
                        temp = self.stack.pop(0)
                        e.values[lambda_expr.identifiers[0]] = temp
                    else:
                        tup = self.stack.pop(0)
                        for i, id in enumerate(lambda_expr.identifiers):
                            e.values[id] = tup.symbols[i]
                    for env in self.environment:
                        if env.get_index() == lambda_expr.get_environment():
                            e.set_parent(env)
                    current_environment = e
```

```
            self.control.append(e)

            self.control.append(lambda_expr.get_delta())

            self.stack.insert(0, e)

            self.environment.append(e)

        elif isinstance(next_symbol, Tup):

            # Handle Tup expression

            tup = next_symbol

            i = int(self.stack.pop(0).get_data())

            self.stack.insert(0, tup.symbols[i - 1])

        elif isinstance(next_symbol, Ystar):

            # Handle Ystar expression

            lambda_expr = self.stack.pop(0)

            eta = Eta()

            eta.set_index(lambda_expr.get_index())

            eta.set_environment(lambda_expr.get_environment())

            eta.set_identifier(lambda_expr.identifiers[0])

            eta.set_lambda(lambda_expr)

            self.stack.insert(0, eta)

        elif isinstance(next_symbol, Eta):

            # Handle Eta expression

            eta = next_symbol

            lambda_expr = eta.get_lambda()

            self.control.append(Gamma())

            self.control.append(Gamma())
```

```
            self.stack.insert(0, eta)

            self.stack.insert(0, lambda_expr)

        else:

            # Handle other symbols

            if next_symbol.get_data() == "Print":

                pass

            elif next_symbol.get_data() == "Stem":

                # implement Stem function

                s = self.stack.pop(0)

                s.set_data(s.get_data()[0])

                self.stack.insert(0, s)

            elif next_symbol.get_data() == "Stern":

                # implement Stern function

                s = self.stack.pop(0)

                s.set_data(s.get_data()[1:])

                self.stack.insert(0, s)

            elif next_symbol.get_data() == "Conc":

                # implement Conc function

                s1 = self.stack.pop(0)

                s2 = self.stack.pop(0)

                s1.set_data(s1.get_data() + s2.get_data())

                self.stack.insert(0, s1)

            elif next_symbol.get_data() == "Order":

                # implement Order function
```

```
            tup = self.stack.pop(0)
            n = Int(str(len(tup.symbols)))
            self.stack.insert(0, n)
        elif next_symbol.get_data() == "Isinteger":
            # implement Isinteger function
            if isinstance(self.stack[0], Int):
                self.stack.insert(0, Bool("true"))
            else:
                self.stack.insert(0, Bool("false"))
            self.stack.pop(1)
        elif next_symbol.get_data() == "Null":
            # implement Null function
            pass
        elif next_symbol.get_data() == "Itos":
            # implement Itos function
            pass
        elif next_symbol.get_data() == "Isstring":
            # implement Isstring function
            if isinstance(self.stack[0], Str):
                self.stack.insert(0, Bool("true"))
            else:
                self.stack.insert(0, Bool("false"))
            self.stack.pop(1)
        elif next_symbol.get_data() == "Istuple":
```

```
            # implement Istuple function
            if isinstance(self.stack[0], Tup):
                self.stack.insert(0, Bool("true"))
            else:
                self.stack.insert(0, Bool("false"))
            self.stack.pop(1)
        elif next_symbol.get_data() == "Isdummy":
            # implement Isdummy function
            if isinstance(self.stack[0], Dummy):
                self.stack.insert(0, Bool("true"))
            else:
                self.stack.insert(0, Bool("false"))
            self.stack.pop(1)
        elif next_symbol.get_data() == "Istruthvalue":
            # implement Istruthvalue function
            if isinstance(self.stack[0], Bool):
                self.stack.insert(0, Bool("true"))
            else:
                self.stack.insert(0, Bool("false"))
            self.stack.pop(1)
        elif next_symbol.get_data() == "Isfunction":
            # implement Isfunction function
            if isinstance(self.stack[0], Lambda):
                self.stack.insert(0, Bool("true"))
```

```
                        else:
                            self.stack.insert(0, Bool("false"))
                        self.stack.pop(1)

            elif isinstance(current_symbol, E):
                # Handle E expression
                self.stack.pop(1)
self.environment[current_symbol.get_index()].set_is_removed(True)
                y = len(self.environment)
                while y > 0:
                    if not self.environment[y - 1].get_is_removed():
                        current_environment = self.environment[y - 1]
                        break
                    else:
                        y -= 1
            elif isinstance(current_symbol, Rator):
                if isinstance(current_symbol, Uop):
                    # Handle Unary operation
                    rator = current_symbol
                    rand = self.stack.pop(0)
                    self.stack.insert(0, self.apply_unary_operation(rator,
rand))
                if isinstance(current_symbol, Bop):
                    # Handle Binary operation
```

```
                rator = current_symbol

                rand1 = self.stack.pop(0)

                rand2 = self.stack.pop(0)

                self.stack.insert(0, self.apply_binary_operation(rator,
rand1, rand2))

        elif isinstance(current_symbol, Beta):

            # Handle Beta expression

            # print(self.stack[0].get_data())

            # self.print_control()

            # self.print_stack()

            # # self.control.pop(-2)

            # self.print_control()

            if (self.stack[0].get_data() == "true"):

                self.control.pop()

            else:

                self.control.pop(-2)

            self.stack.pop(0)




        elif isinstance(current_symbol, Tau):

            # Handle Tau expression

            tau = current_symbol

            tup = Tup()
```

```
            for _ in range(tau.get_n()):
                tup.symbols.append(self.stack.pop(0))
            self.stack.insert(0, tup)
        elif isinstance(current_symbol, Delta):
            # Handle Delta expression
            self.control.extend(current_symbol.symbols)
        elif isinstance(current_symbol, B):
            # Handle B expression
            self.control.extend(current_symbol.symbols)
        else:
            self.stack.insert(0, current_symbol)



# def print_stack(self):
#     print("Stack: ", end="")
#     for symbol in self.stack:
#         print(symbol.get_data(), end="")
#         if isinstance(symbol, (Lambda, Delta, E, Eta)):
#             print(symbol.get_index(), end="")
#         print(",", end="")
#     print()

# def print_control(self):
```

```python
#     print("Control: ", end="")
#     for symbol in self.control:
#         print(symbol.get_data(), end="")
#         if isinstance(symbol, (Lambda, Delta, E, Eta)):
#             print(symbol.get_index(), end="")
#         print(",", end="")
#     print()

def write_stack_to_file(self, file_path):
    with open(file_path, 'a') as file:
        for symbol in self.stack:
            file.write(symbol.get_data())
            if isinstance(symbol, (Lambda, Delta, E, Eta)):
                file.write(str(symbol.get_index()))
            file.write(",")
        file.write("\n")

def write_control_to_file(self, file_path):
    with open(file_path, 'a') as file:
        for symbol in self.control:
            file.write(symbol.get_data())
            if isinstance(symbol, (Lambda, Delta, E, Eta)):
                file.write(str(symbol.get_index()))
            file.write(",")
```

```python
        file.write("\n")


def clear_file(file_path):

    open(file_path, 'w').close()





def print_environment(self):

    # Print the environment symbols

    for symbol in self.environment:

        print(f"e{symbol.get_index()} --> ", end="")

        if symbol.get_index() != 0:

            print(f"e{symbol.get_parent().get_index()}")

        else:

            print()


def covert_string_to_bool(self, data):

    if data == "true":

        return True

    elif data == "false":

        return False


def apply_unary_operation(self, rator, rand):
```

```
        # Apply unary operation
        if rator.get_data() == "neg":
            val = int(rand.get_data())
            return Int(str(-1 * val))
        elif rator.get_data() == "not":
            val = self.covert_string_to_bool(rand.get_data())
            return Bool(str(not val).lower())
        else:
            return Err()


    def apply_binary_operation(self, rator, rand1, rand2):
        # Apply binary operation
        if rator.get_data() == "+":
            val1 = int(rand1.get_data())
            val2 = int(rand2.get_data())
            return Int(str(val1 + val2))
        elif rator.data == "-":
            val1 = int(rand1.data)
            val2 = int(rand2.data)
            return Int(str(val1 - val2))
        elif rator.data == "*":
            val1 = int(rand1.data)
            val2 = int(rand2.data)
            return Int(str(val1 * val2))
```

```python
        elif rator.data == "/":

            val1 = int(rand1.data)

            val2 = int(rand2.data)

            return Int(str(int(val1 / val2)))

        elif rator.data == "**":

            val1 = int(rand1.data)

            val2 = int(rand2.data)

            return Int(str(val1 ** val2))

        elif rator.data == "&":

            val1 = self.covert_string_to_bool(rand1.data)

            val2 = self.covert_string_to_bool(rand2.data)

            return Bool(str(val1 and val2).lower())

        elif rator.data == "or":

            val1 = self.covert_string_to_bool(rand1.data)

            val2 = self.covert_string_to_bool(rand2.data)

            return Bool(str(val1 or val2).lower())

        elif rator.data == "eq":

            val1 = rand1.data

            val2 = rand2.data

            return Bool(str(val1 == val2).lower())

        elif rator.data == "ne":

            val1 = rand1.data

            val2 = rand2.data

            return Bool(str(val1 != val2).lower())
```

```
        elif rator.data == "ls":
            val1 = int(rand1.data)
            val2 = int(rand2.data)
            return Bool(str(val1 < val2).lower())
        elif rator.data == "le":
            val1 = int(rand1.data)
            val2 = int(rand2.data)
            return Bool((val1 <= val2))
        elif rator.data == "gr":
            val1 = int(rand1.data)
            val2 = int(rand2.data)
            return Bool(str(val1 > val2).lower())
        elif rator.data == "ge":
            val1 = int(rand1.data)
            val2 = int(rand2.data)
            return Bool(str(val1 >= val2).lower())
        elif rator.data == "aug":
            if isinstance(rand2, Tup):
                rand1.symbols.extend(rand2.symbols)
            else:
                rand1.symbols.append(rand2)
            return rand1
        else:
            return Err()
```

```python
def get_tuple_value(self, tup):
    # Get the value of a tuple
    temp = "("
    for symbol in tup.symbols:
        if isinstance(symbol, Tup):
            temp += self.get_tuple_value(symbol) + ", "
        else:
            temp += symbol.get_data() + ", "
    temp = temp[:-2] + ")"
    return temp


def get_answer(self):
    # Get the answer from the CSEMachine
    self.execute()
    if isinstance(self.stack[0], Tup):
        return self.get_tuple_value(self.stack[0])
    return self.stack[0].get_data()
```

This is a part of the code. The complete codebase is attached herewith.

4. Run the Programme

The interpreter can be run in two methods.

1. Using Direct Python Commands
2. Using Makefile

4.1 Using Direct Python Commands

- Run this command for the output:

```
PS C:\Users\LENOVO LOQ\Desktop\PL project\RPAL-Project>  python myrpal.py input.txt
Output of the above program is:
15
```

- To get output as an AST, run this command:

```
PS C:\Users\LENOVO LOQ\Desktop\PL project\RPAL-Project>  python myrpal.py input.txt  -ast
```

- Output:

```
let
.function_form
..<ID:Sum>
..<ID:A>
..where
...gamma
....<ID:Psum>
....tau
.....<ID:A>
.....gamma
......<ID:Order>
......<ID:A>
...rec
....function_form
.....<ID:Psum>
.....,
......<ID:T>
......<ID:N>
.....->
......eq
.......<ID:N>
.......<INT:0>
......<INT:0>
......+
.......gamma
........<ID:Psum>
........tau
.........<ID:T>
.........-
..........<ID:N>
..........<INT:1>
.......gamma
........<ID:T>
........<ID:N>
.gamma
..<ID:Print>
..gamma
...<ID:Sum>
...tau
....<INT:1>
....<INT:2>
....<INT:3>
....<INT:4>
....<INT:5>
```

- Run the following commands for output as Standardized AST.

```
PS C:\Users\LENOVO LOQ\Desktop\PL project\RPAL-Project>  python myrpal.py input.txt  -sast
```

- Output:

```
gamma
.lambda
..<ID:Sum>
..gamma
...<ID:Print>
...gamma
....<ID:Sum>
....tau
.....<INT:1>
.....<INT:2>
.....<INT:3>
.....<INT:4>
.....<INT:5>
.lambda
..<ID:A>
..gamma
...lambda
....<ID:Psum>
....gamma
.....<ID:Psum>
.....tau
......<ID:A>
......gamma
.......<ID:Order>
.......<ID:A>
...gamma
....<Y*>
....lambda
.....<ID:Psum>
.....lambda
......,
.......<ID:T>
.......<ID:N>
......->
.......eq
........<ID:N>
........<INT:0>
.......<INT:0>
.......+
........gamma
.........<ID:Psum>
.........tau
..........<ID:T>
..........-
...........<ID:N>
...........<INT:1>
........gamma
.........<ID:T>
.........<ID:N>
```

4.2 Using Makefile

- Open the terminal and navigate to the directory where the interpreter is located.
- Run the following commands for output, AST, and Standardized AST

```
make run file=input.txt (For the output)
make ast file=input.txt (For the AST)
make sast file=input.txt (For the SAST)
```

5. Troubleshooting

- Python was not found
  If you encounter an error stating that Python was not found, please ensure that Python is installed on your system and that it is added to your system's PATH environment variable.
  If you are using the command as python, try using python3. Some Linux distributions use python3 instead of python.

6. Conclusion

Developing this mini-compiler for RPAL was aimed at understanding how programming languages are interpreted and executed at a fundamental level. Starting from tokenizing the code, building syntactic structures, and finally executing the program, each step revealed the intricate mechanisms of how interpreters operate behind the scenes. This comprehensive implementation provided valuable insights into compiler design principles and the theoretical foundations of programming language processing.

We successfully implemented a complete lexical analyzer, parser, AST to ST conversion algorithm, and CSE machine for the RPAL language. Our program demonstrates the ability to efficiently parse RPAL programs, construct accurate Abstract Syntax Trees, transform them into executable formats, and execute them using the CSE machine to produce correct computational results. The modular design of our interpreter ensures maintainability and extensibility, while the manual implementation of parsing algorithms enhanced our understanding of syntactic analysis techniques.

Through this project, we gained practical experience in implementing the core components of a programming language interpreter, bridging the gap between theoretical computer science concepts and their real-world applications in software development.

Please find the GitHub repo for the project:
https://github.com/dewminawijekoon/RPAL-Project