Project Report

- RPAL Interpreter –

CS3513 – Programming Languages

Group Name: AST

Group Members:

220098C – De Silva A.D.D.T.
220186U – Gunarathna H.R.A.

Table of Contents

1. Introduction

1.1 What is RPAL?

RPAL is a subset of PAL, the **Pedagogic Algorithmic Language**. There are three versions of PAL: RPAL, LPAL, and JPAL. The ‘R’ in RPAL stands for **right-reference**, as opposed to ‘L’ (LPAL), which stands for ‘left-reference’. RPAL is a functional language. Every RPAL program is nothing more than an expression, and ‘running’ an RPAL program consists of nothing more than evaluating the expression, yielding one result.

1.2 Project Overview

The objective of this project is to build an interpreter for the RPAL language. The Interpreter consists of four core components.

1. Lexical Analysis: The lexical analyzer (lexer) reads a source file containing an RPAL program and breaks it into valid tokens.
2. Parsing: The parser takes the list of tokens and constructs the Abstract Syntax Tree (AST).
3. Standardization: The AST is then transformed into a Standardized Tree (ST).
4. CSE Machine Implementation: Finally, the Standardized Tree is executed using a Control Stack Environment (CSE) machine.

2. Development Details

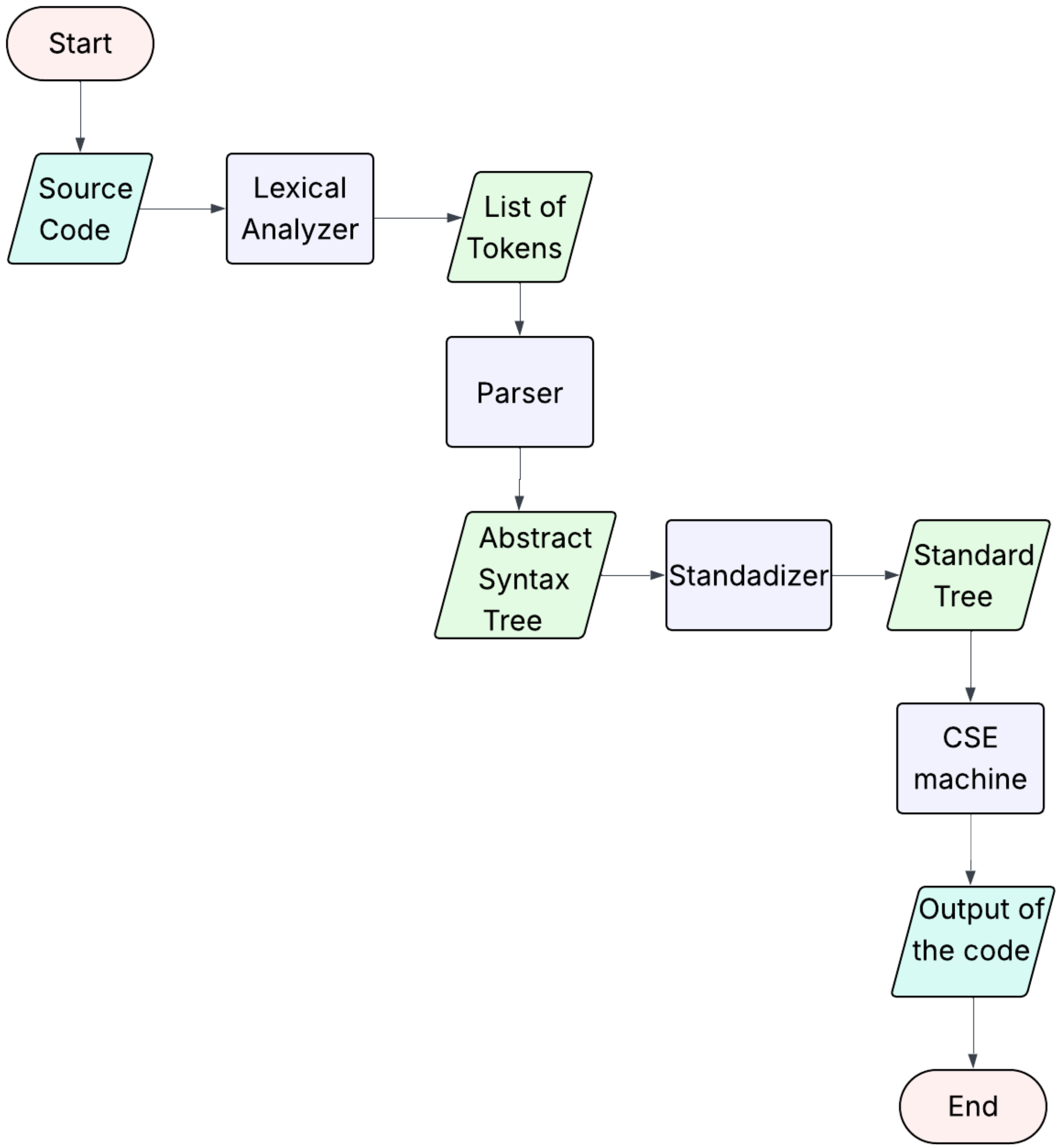
2.1 Language and environment used

For the implementation of the RPAL interpreter, we used the **Python** programming language due to its simplicity, readability, and rich set of built-in data structures, which are especially useful for handling tokens and tree structures.

We developed the project using **Visual Studio Code** (VSCode) as the primary integrated development environment (IDE). For version control, we used **GitHub** to manage our codebase, collaborate, and maintain a history of changes.

2.2 Program Structure

The program follows a modular structure to promote clarity and maintainability. The flow of the execution is illustrated below.



2.3 Folder Structure

The folder structure and the tasks performed by each file are described below.

```
├── Lexer/
  └── tokenizer.py          Reads input RPAL code and generates a list of tokens
├── Parser/
  └── parser.py             Parses the token list and build an AST
├── Standardizer/
  └── ast_builder.py        Converts the string format AST to AST data structure
  └── ast.py                Defines the AST structure with methods to standardize and traverse
  └── node.py               Defines node structure with standardization logic for each node type
├── CSEM/
  └── control_structures.py Generates the control structures for the program from the ST
  └── csemachine.py         Executes the CSE Machine to generate the final output
  └── symbols.py            Defines the classes for all symbols used in the CSE Machine
├── .gitignore
├── input.txt               Sample input file
├── Makefile                Used to run and manage the project
├── myrpal.py               Main entry point
├── README.md
```

2.4 Functional Modules

The program is divided into several functional modules, each handling a specific part of the RPAL interpretation pipeline.

- Lexer
- Parser
- Standardizer
- CSE Machine

Lexer

Purpose:

Performs lexical analysis by reading the input RPAL source code and converting it into a list of tokens according to the rules defined in the `RPAL_Lex` document.

Key Functionalities:

- The `tokenizer` function reads the RPAL source code and identifies tokens such as keywords, identifiers, operators, comments, strings, and integers.
- Removes unwanted tokens like comments and white spaces
- Returns a list of token objects where each token has two attributes: `token_type` and `token_value`.

```
import re

class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def get_type(self):
        return self.type

    def get_value(self):
        return self.value

# Define token type constants
KEYWORD = 'KEYWORD'
IDENTIFIER = 'IDENTIFIER'
INTEGER = 'INTEGER'
OPERATOR = 'OPERATOR'
STRING = 'STRING'
SPACES = 'SPACES'
COMMENT = 'COMMENT'
PUNCTUATION = 'PUNCTUATION'

token_types = {
    COMMENT: r'//.*',
    KEYWORD:
r'\b(let|in|fn|where|aug|or|not|gr|ge|ls|le|eq|ne|true|false|nil|dummy|within|and
|rec)\b',
    IDENTIFIER: r'[a-zA-Z][a-zA-Z0-9_]*',
    INTEGER: r'\d+',
```

```python
    STRING: r"'(?:\\t|\\n|\\\\|\\'|\\\"|[();, a-zA-Z0-9+\-
*/<>&.@/:=˜|$!#%ˆ_\[\]{}\"`?])*'",
    OPERATOR: r'[+\-*<>&.@/:=~|$!#%^_\[\]{}"\'?]+',
    SPACES: r'\s+',
    PUNCTUATION: r'[();,]'
}

# Combine into named regex groups
token_regex = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in
token_types.items())
compiled_regex = re.compile(token_regex)

def tokenizer(code):
    tokens = []
    pos = 0
    length = len(code)

    while pos < length:
        match = compiled_regex.match(code, pos) #scanning tokens
        if not match:
            raise SyntaxError(f"Unrecognized token at position {pos}:
{code[pos]!r}")

        kind = match.lastgroup
        value = match.group()

        if kind not in ['SPACES', 'COMMENT']: #screening unwanted tokens
            if kind == 'PUNCTUATION':
                kind = value
            tokens.append(Token(kind, value))

        pos = match.end()

    return tokens
```
[lexer.py]

Parser

Purpose:

Takes the list of tokens and constructs the Abstract Syntax Tree (AST) based on the grammar provided in the `RPAL_Grammar` document. The AST accurately represents the hierarchical syntactic structure of the input program.

Key Functionalities:

- The parser reads a list of tokens (produced by `tokernizer.py`) and constructs an abstract syntax tree. The AST is built in a bottom-up manner.
- `Node Class` represents a node in the AST, storing its type value and the number of children.
- The `parser class` uses mutually recursive methods to match the grammar rules, which follow the language's syntax. Each method (e.g., `E, T, A`) corresponds to a non-terminal in the grammar
- `print_ast()` allows for visualizing the AST in a readable format.

```
class Node:
    def __init__(self,node_type:NodeType, value,children_count):
        self.node_type=node_type
        self.value=value
        self.children_count=children_count
```

[Node Class]

```python
class Parser:
    def __init__(self, token_list):
        self.token_list = token_list
        self.string_ast=[]
        self.ast = []
    #========================printing part=======================
    def print_ast(self):
        dots = ""
        stack = []

        while self.ast:
            if not stack:
                if self.ast[-1].children_count == 0:
                    self.add_strings(dots, self.ast.pop())
                else:
                    node = self.ast.pop()
                    stack.append(node)
            else:
                if self.ast[-1].children_count > 0:
                    node = self.ast.pop()
                    stack.append(node)
                    dots += "."
                else:
                    stack.append(self.ast.pop())
                    dots += "."
                    while stack[-1].children_count == 0:
                        self.add_strings(dots, stack.pop())
                        if not stack:
                            break
                        dots = dots[:-1]
                        node = stack.pop()
                        node.children_count -= 1
                        stack.append(node)
```

[Parser Class]

```python
B   -> B ’or’ Bt       => ’or’
    -> Bt
def B(self):
    self.Bt()
    while self.token_list[0].get_value()=='or':
        self.token_list.pop(0)
        self.Bt()
        self.ast.append(Node(NodeType.OR,'or',2))

Bt  -> Bt ’&’ Bs     => ’&’
    -> Bs
def Bt(self):
    self.Bs()
    while self.token_list[0].get_value()=='&':
        self.token_list.pop(0)
        self.Bs()
        self.ast.append(Node(NodeType.AND,'&',2))
```

[some example methods that used for grammar rules]

Standardizer

Purpose:

Given the Abstract Syntax Tree(AST), generate the Standardized Abstract Syntax Tree (SAST) according to the RPAL Subtree Transformational Grammars.

Key Functionalities:

- Reads the input provided as `string_ast`—a string representation of the AST as specified in the project.
- Converts `string_ast` into a tree data structure.
- Feeds the root of this tree to the `standardize()` function, which recursively standardizes the tree in a bottom-up manner.
- Produces and outputs a standardized tree representation (SAST) of the original AST.

```python
def standardize(self):
    """
    Standardize this node while recursively standardizing all children first.
    """
    if self.is_standardized:
        return

    # First standardize all children
    for child in self.children:
        child.standardize()

    # Apply standardization rules based on node type
    if self.data == "let":
        self._standardize_let()
    elif self.data == "where":
        self._standardize_where()
    elif self.data == "function_form":
        self._standardize_function_form()
    elif self.data == "lambda":
        self._standardize_lambda()
    elif self.data == "within":
        self._standardize_within()
    elif self.data == "@":
        self._standardize_at_operator()
    elif self.data == "and":
        self._standardize_simultaneous_def()
    elif self.data == "rec":
        self._standardize_recursive_def()

    self.is_standardized = True
```

```python
def _standardize_let(self):
    """
    Standardize LET node:
         LET              GAMMA
        /    \           /     \
      EQUAL   P   ->   LAMBDA   E
     /   \             /    \
    X     E           X      P
    """

    expr = self.children[0].children[1]
    expr.set_parent(self)
    expr.set_depth(self.depth + 1)

    p_node = self.children[1]
    p_node.set_parent(self.children[0])
    p_node.set_depth(self.depth + 2)

    self.children[1] = expr
    self.children[0].set_data("lambda")
    self.children[0].children[1] = p_node
    self.set_data("gamma")

def _standardize_where(self):
    """
    Standardize WHERE node:
          WHERE               GAMMA
          /   \              /     \
         P     EQUAL   ->  LAMBDA    E
               /   \       /   \
              X     E     X     P
    """
    self.children[0], self.children[1] = self.children[1], self.children[0]
    self.set_data("let")
    self.standardize()
```

Code snippet of the standardization process in the `ASTNode` *class*

CSE Machine

Purpose:

Given a Standardized Abstract Syntax Tree (SAST), this module executes the program to generate the final output according to the rules of the CSE Machine.

Key Functionalities:

- Reads the input, which is a tree data structure representing the Standardized Abstract Syntax Tree (SAST).
- Performs a pre-order traversal of the tree to generate the control structures of the program.
- Initializes the stack and environment, and feeds these along with the generated control structures to the CSE Machine.
- The CSE Machine then updates the control, stack, and environment according to its operational rules.
- Produces the final output of the input program as a result.

```python
def create_control_structure(self, ast):
    root_delta = self.create_delta(ast.get_root())
    return [self.env0, root_delta]

def create_stack(self):
    return [self.env0]

def create_environment(self):
    return [self.env0]

def create_cse_machine(self, ast):
    """
    Creates and returns a new CSE machine instance initialized with the control,
    stack, and environment structures.
    """
    control = self.create_control_structure(ast)
    stack = self.create_stack()
    environment = self.create_environment()
    return CSEMachine(control, stack, environment)
```

```python
def create_lambda(self, node):
    """
    Handles the creation of a new control structure and setting the lambda index and identifiers.
    """
    lambda_structure = Lambda(self.lambda_index)
    self.lambda_index += 1

    lambda_structure.set_delta(self.create_delta(node.get_children()[1]))

    param_node = node.get_children()[0]
    if param_node.get_data() == ",":
        # Multiple parameters
        for child in param_node.get_children():
            identifier_name = child.get_data()[12:-1]
            lambda_structure.identifiers.append(Id(identifier_name))
    else:
        # Single parameter
        identifier_name = param_node.get_data()[12:-1]
        lambda_structure.identifiers.append(Id(identifier_name))

    return lambda_structure

def pre_order_traverse(self, node):
    symbols = []
    if node.get_data() == "lambda":
        # Handling Lambda encounters
        symbols.append(self.create_lambda(node))
    elif node.get_data() == "->":
        # Handling conditional expressions (->)
        symbols.append(self.create_delta(node.get_children()[1]))
        symbols.append(self.create_delta(node.get_children()[2]))
        symbols.append(Beta())
        symbols.append(self.create_B(node.get_children()[0]))
    else:
        # Standard nodes
        symbols.append(self.map_ast_node_to_symbol(node))
        for child in node.get_children():
            symbols.extend(self.pre_order_traverse(child))
    return symbols

def create_delta(self, node):
    delta_structure = Delta(self.delta_index)
    self.delta_index += 1
    delta_structure.symbols = self.pre_order_traverse(node)
    return delta_structure
```

Code snippet of the generation of control structures in the `ControlStructure` *class*

```python
class CSEMachine:
    def __init__(self, control, stack, environment):
        self.control = control
        self.stack = stack
        self.environments = environment

    def execute(self):
        """Main execution loop of the CSE machine."""
        current_env = self.environments[0]
        new_env_index = 1

        while self.control:
            symbol = self.control.pop()

            # Identifier: push value from current environment
            if isinstance(symbol, Id):
                self.stack.insert(0, current_env.lookup(symbol))

            # Lambda: push Lambda and set its environment index
            elif isinstance(symbol, Lambda):
                symbol.set_environment(current_env.get_index())
                self.stack.insert(0, symbol)

            # Gamma application: apply to Lambda, Tup, Ystar, Eta, or built-in function
            elif isinstance(symbol, Gamma):
                next_symbol = self.stack.pop(0)
                if isinstance(next_symbol, Lambda):

                    lambda_expr = next_symbol
                    new_env = E(new_env_index)
                    new_env_index += 1

                    if len(lambda_expr.identifiers) == 1:
                        arg = self.stack.pop(0)
                        new_env.values[lambda_expr.identifiers[0]] = arg
                    else:
                        tup = self.stack.pop(0)
                        for i, ident in enumerate(lambda_expr.identifiers):
                            new_env.values[ident] = tup.symbols[i]

                    for env in self.environments:
                        if env.get_index() == lambda_expr.get_environment():
                            new_env.set_parent(env)
```

Code snippet of the execution of the CSE machine in the `CSEMachine` *class*

3. Command-Line Usage

The RPAL interpreter is executed via the command line using Python. The usage follows one of the formats depending on the desired output.

`python myrpal.py file_name` : Executes the full program and produces final output

`python myrpal.py file_name -ast` : Displays only the Abstract Syntax Tree (AST)

4. Example input & output

Here is a sample RPAL code:

```
let x=3 in print(x, x**2)
```

Example output when using -ast switch:

```
let
.=
..<IDENTIFIER:x>
..<INTEGER:3>
.gamma
..<IDENTIFIER:print>
..tau
...<IDENTIFIER:x>
...**
....<IDENTIFIER:x>
....<INTEGER:2>
```

Example output without using -ast switch:

```
Output of the above program is:
(3, 9)
```

5. Conclusion

Through this project, we explored the core stages of building a programming language interpreter, including lexical analysis, parsing, AST generation, standardization, and execution via a CSE machine. This hands-on experience highlighted the challenges and importance of implementing language tools manually and gave us a solid foundation for more advanced topics in compilers and interpreters.

6. Appendix

- GitHub Repository Link: https://github.com/Dinara-De-Silva/PL_Project.git

7. References

- RPAL_Lex.pdf
- RPAL_Grammar.pdf