RPAL Interpreter Implementation

CS 3513 - Programming Languages
Programming Project

Dissanayake D.M.S.H - 220138C
Lihinikaduarachchi L.A.G.H - 220361D

Contents

Chapter 1

Introduction

The Recursive Programming Algorithmic Language (RPAL) is a functional programming language designed for educational purposes. It incorporates many features common to functional languages, such as higher-order functions, recursion, and pattern matching. This report details the implementation of an interpreter for the RPAL language as part of the CS 3513 Programming Languages course project.

The primary objective of this project was to develop a complete interpreter for the RPAL language that follows the standard RPAL language specification. The implementation includes several key components: a lexical analyzer, a parser, an Abstract Syntax Tree (AST) standardizer, and a Control Structure Environment (CSE) machine for program execution.

The interpreter is implemented in Python, leveraging the language's flexibility and rich standard library to create a clean, modular, and maintainable codebase. The implementation follows a traditional compiler/interpreter architecture, with distinct phases for lexical analysis, parsing, and execution.

This report provides a comprehensive overview of the implementation, including detailed descriptions of each component, function prototypes showing the structure of the programs, and insights into the design decisions made during development. The report is organized as follows:

- Chapter 1 (Introduction) provides an overview of the project and its objectives.
- Chapter 2 (Project Overview) describes the RPAL language and the project requirements.
- Chapter 3 (Implementation Details) delves into the technical details of each component of the interpreter.
- Chapter 4 (Function Prototypes and Program Structure) presents the structure of the program and the function prototypes.
- Chapter 5 (Testing and Results) discusses the testing methodology and results.
- Chapter 6 (Conclusion) summarizes the project and discusses potential future improvements.

Chapter 2

Project Overview

2.1 The RPAL Language

RPAL (Recursive Programming Algorithmic Language) is a functional programming language designed for educational purposes. It shares many characteristics with other functional languages like Haskell and ML, but with a syntax and semantics tailored for teaching programming language concepts.

Key features of the RPAL language include:

- **Functional Programming Paradigm**: RPAL is a functional language where functions are first-class citizens, meaning they can be passed as arguments, returned as values, and stored in data structures.
- **Static Typing**: RPAL uses a static type system, although type annotations are not required as the system employs type inference.
- **Higher-Order Functions**: Functions can take other functions as arguments and return functions as results.
- **Recursion**: RPAL supports recursive function definitions, which is the primary mechanism for iteration in the language.
- **Pattern Matching**: The language includes pattern matching capabilities for destructuring data and defining conditional behavior.
- **Lazy Evaluation**: RPAL uses lazy evaluation, meaning expressions are only evaluated when their values are needed.
- **Tuples and Lists**: The language supports composite data structures like tuples and lists.

2.2 Project Requirements

The project requirements, as specified in the assignment, were to implement a complete interpreter for the RPAL language. The implementation should include:

- A lexical analyzer that converts RPAL source code into tokens according to the lexical rules specified in RPAL_Lex.pdf.

- A parser that builds an Abstract Syntax Tree (AST) according to the grammar rules specified in RPAL_Grammar.pdf.
- An algorithm to convert the AST into a Standardized Tree (ST).
- A CSE (Control Structure Environment) machine to execute the standardized tree.

The implementation should be able to read an RPAL program from an input file and produce output that matches the output of the reference implementation (rpal.exe) for the same program. The implementation should be done in C/C++, Java, or Python, without using tools like lex, yacc, or similar parser generators.

2.3 Implementation Approach

For this project, we chose to implement the RPAL interpreter in Python due to its readability, flexibility, and rich standard library. The implementation follows a traditional compiler/interpreter architecture with distinct phases:

1. **Lexical Analysis**: The source code is scanned and converted into a sequence of tokens using regular expressions for pattern matching.
2. **Parsing**: The tokens are parsed according to the RPAL grammar rules to build an Abstract Syntax Tree (AST) using a recursive descent parser.
3. **AST Standardization**: The AST is transformed into a Standardized Tree (ST) according to the RPAL standardization rules.
4. **Execution**: The standardized tree is executed using a CSE machine, which interprets the tree and produces the program output.

The implementation is modular, with each component encapsulated in its own class or module. This approach enhances maintainability and allows for easier testing and debugging of individual components.

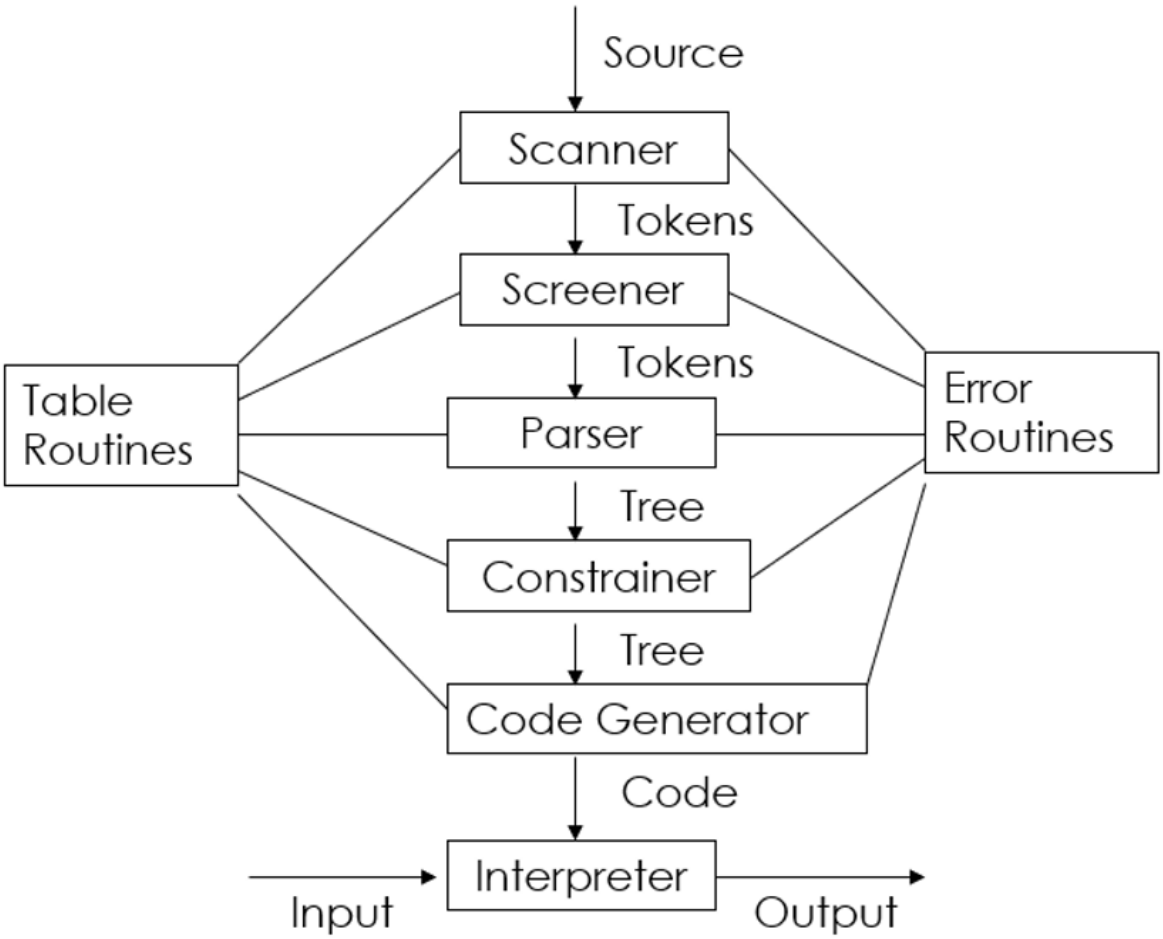

Figure 2.1: RPAL Interpreter Architecture

Chapter 3

Implementation Details

This chapter provides a detailed description of the implementation of each component of the RPAL interpreter. The implementation follows a modular approach, with each component encapsulated in its own class or module.

3.1 Lexical Analyzer

The lexical analyzer, or lexer, is responsible for converting the source code into a sequence of tokens. Each token represents a meaningful unit in the source code, such as a keyword, identifier, operator, or literal.

3.1.1 Token Types

The lexer recognizes the following token types:

- **KEYWORD**: Reserved words in the RPAL language, such as 'let', 'in', 'where', 'fn', etc.
- **IDENTIFIER**: Names of variables and functions.
- **INTEGER**: Numeric literals.
- **STRING**: String literals enclosed in single quotes.
- **OPERATOR**: Special characters and operators, such as '+', '-', '*', '/', etc.
- **PUNCTUATION**: Delimiters like parentheses and commas.
- **EOF**: End of file marker.

These token types are defined as an enumeration in the 'TokenType' class:

```
class TokenType(Enum):
    """Enumeration of all possible token types in the RPAL language."""
    KEYWORD = 'KEYWORD'      # Reserved words like 'let', 'in', 'where', etc.
    IDENTIFIER = 'IDENTIFIER'  # Variable and function names
    INTEGER = 'INTEGER'      # Numeric literals
    STRING = 'STRING'        # String literals enclosed in single quotes
```

```
    OPERATOR = 'OPERATOR'    # Special characters and operators
    PUNCTUATION = 'PUNCTUATION'  # Delimiters like parentheses and
   commas
    EOF = 'EOF'              # End of file marker
```

Listing 3.1: Token Types Definition

3.1.2 Token Object

Each token is represented by a ‘Token‘ object, which contains the token's type, value, and position in the source code:

```
class Token:
    """Represents a lexical token with its type, value, and position in
    source code."""

    def __init__(self, type_, value, line=None, column=None):
        self.type = type_       # TokenType enum value
        self.value = value      # Actual token value
        self.line = line        # Line number in source
        self.column = column    # Column position in line

    def __repr__(self):
        """String representation of the token with type, value, and
   position."""
        if self.type == TokenType.EOF:
            return f"<{self.type.name}     {self.value!r}>"
        location = f" @ {self.line}:{self.column}"
        return f"<{self.type.name}{location}     {self.value!r}>"
```

Listing 3.2: Token Class Definition

3.1.3 Lexer Class

The ‘Lexer‘ class is responsible for tokenizing the source code. It takes the source code as input and produces a list of tokens:

```
class Lexer:
    """Converts source code into a sequence of tokens using regular
   expression matching."""

    def __init__(self, source_code):
        self.source = source_code    # Input source code
        self.tokens = []             # List of recognized tokens
        self.line = 1                # Current line number

    def tokenize(self):
        """
        Tokenizes the source code into a sequence of tokens.

        Returns:
            list[Token]: List of tokens representing the source code

        Raises:
            SyntaxError: If an illegal character is encountered
        """
```

```
        # Implementation details...
```

Listing 3.3: Lexer Class Definition

The ‘tokenize‘ method scans the source code character by character, matching each character against the token patterns. When a match is found, a token is created and added to the list of tokens. The method returns the list of tokens when the entire source code has been processed.

3.2 Parser

The parser is responsible for analyzing the sequence of tokens produced by the lexer and constructing an Abstract Syntax Tree (AST) according to the RPAL grammar rules. The parser implemented for this project is a recursive descent parser, which is a top-down parsing technique where the parser starts from the top-level grammar rule and recursively processes the input according to the grammar rules.

3.2.1 Parser Class

The ‘Parser‘ class is the main class responsible for parsing the token stream. It takes a list of tokens as input and produces an AST as output:

```
class Parser:
    """Implements a recursive descent parser for RPAL language."""

    def __init__(self, tokens):
        self.tokens = tokens      # List of tokens to parse
        self.pos = 0              # Current position in token stream
        self.stack = []           # Stack for building AST nodes

    def peek(self):
        """Look at the current token without consuming it."""
        if self.pos < len(self.tokens):
            return self.tokens[self.pos]
        return None

    def match(self, expected_type, expected_value=None):
        """
        Match and consume the current token if it matches expected type and value.
        Raises SyntaxError if token doesn't match or end of input is reached.
        """
        # Implementation details...
```

Listing 3.4: Parser Class Definition

The ‘Parser‘ class maintains a position pointer (‘pos‘) to keep track of the current token being processed, and a stack (‘stack‘) to build the AST nodes. The ‘peek‘ method allows the parser to look at the current token without consuming it, and the ‘match‘ method consumes the current token if it matches the expected type and value.

3.3 AST Standardization

After the parser constructs the Abstract Syntax Tree (AST), the next step is to standardize the AST according to the RPAL standardization rules. The standardization process transforms the AST into a form that is easier to execute by the CSE machine.

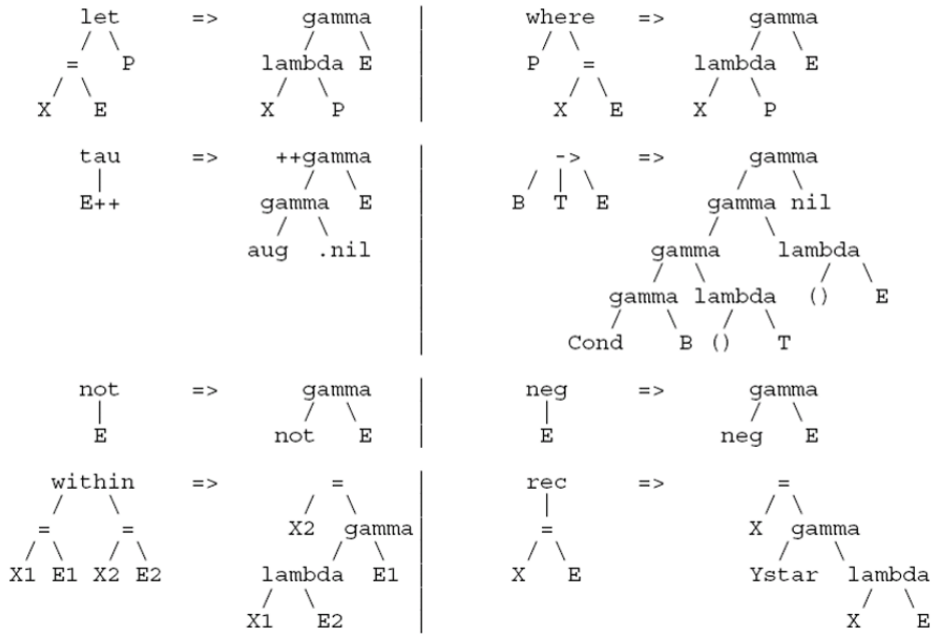

Figure 3.1: RPAL Function Form and Lambda Transformations

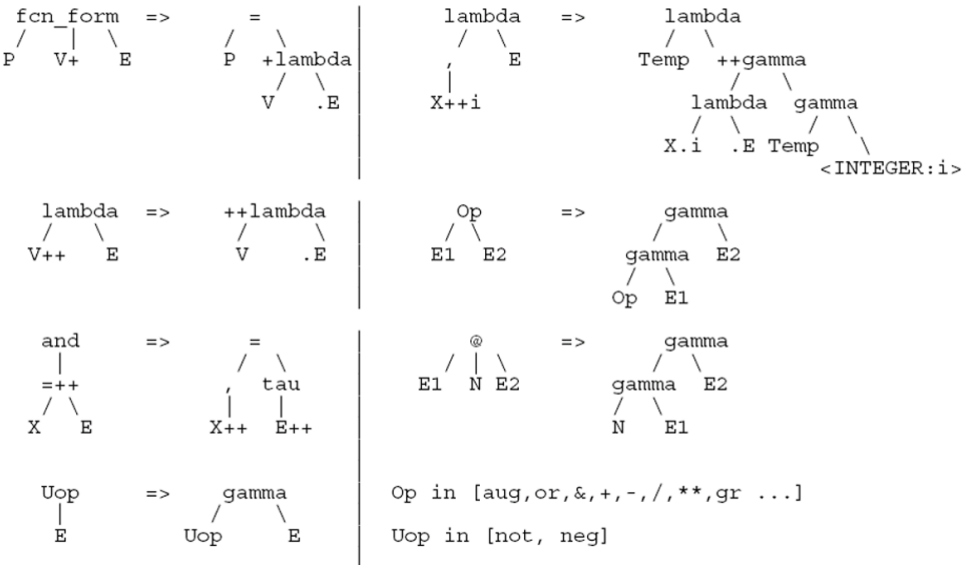

Figure 3.2: RPAL Subtree Transformational Grammar

3.3.1 AST Class

The 'AST' class represents the Abstract Syntax Tree and provides methods for standardizing the tree:

```python
class AST:
    """Abstract Syntax Tree class that represents the hierarchical structure of parsed code."""

    def __init__(self, root = None):
        """Initialize AST with an optional root node."""
        self.root = root

    def set_root(self, root):
        """Set the root node of the AST."""
        self.root = root

    def get_root(self):
        """Get the root node of the AST."""
        return self.root

    def standardize(self):
        """Standardize the AST by applying standardization rules starting from root."""
        if not self.root.is_standardized:
            self.root.standardize()

    def pre_order_traverse(self, node, i):
        """
        Perform pre-order traversal of the AST.
        Args:
            node: Current node to process
            i: Current indentation level
        """
        print("." * i + str(node.get_data()))
        for child in node.children:
            self.pre_order_traverse(child, i + 1)

    def print_ast(self):
        """Print the AST structure using pre-order traversal with indentation."""
        self.pre_order_traverse(self.get_root(), 0)
```

Listing 3.5: AST Class Definition

3.3.2 Node Class

The ‘Node‘ class represents a node in the AST. Each node has a data value, a list of children, a parent reference, and a depth value. The ‘standardize‘ method is responsible for standardizing the node according to the RPAL standardization rules:

```python
class Node:
    """Represents a node in the Abstract Syntax Tree with data, depth, parent-child relationships, and standardization status."""

    def __init__(self):
        self.data = None
        self.depth = 0
        self.parent = None
        self.children = []
        self.is_standardized = False

```

```
    # Basic getters and setters
    def set_data(self, data):
        self.data = data

    def get_data(self):
        return self.data

    def get_degree(self):
        """Returns the number of children of this node."""
        return len(self.children)

    def get_children(self):
        return self.children

    def set_depth(self, depth):
        self.depth = depth

    def get_depth(self):
        return self.depth

    def set_parent(self, parent):
        self.parent = parent

    def get_parent(self):
        return self.parent

    def standardize(self):
        """
        Standardizes the AST node according to RPAL standardization rules.
        Applies different transformations based on node type (let, where, function_form, etc.).
        Each transformation converts the node into a standardized form using gamma and lambda nodes.
        """
        # Standardization logic...
```

Listing 3.6: Node Class Definition

3.4 CSE Machine

The Control Structure Environment (CSE) machine is responsible for executing the standardized tree. It interprets the tree and produces the program output.

	CONTROL	STACK	ENV
Initial State	e_0 δ_0	e_0	e_0 = PE
CSE Rule 1 (stack a name)	.... Name	.... Ob	Ob=Lookup(Name,e_c) e_c:current environment
CSE Rule 2 (stack λ)	.... λ_k^x	.... $^c\lambda_k^x$	e_c:current environment
CSE Rule 3 (apply rator)	.... γ	Rator Rand .... Result	Result=Apply[Rator,Rand]
CSE Rule 4 (apply λ)	.... γ .... e_n δ_k	$^c\lambda_k^x$ Rand .... e_n	e_n = [Rand/x]e_c
CSE Rule 5 (exit env.)	.... e_n	value e_n .... value	

Figure 3.3: CSE Rules 1-5: Initial State and Basic Rules

CSE Rules 6 and 7: Unary and Binary Operators.

	CONTROL	STACK	ENV
CSE Rule 6 (binop)	.... binop	Rand Rand .... Result	Result=Apply[binop,Rand,Rand]
CSE Rule 7 (unop)	.... unop	Rand .... Result	Result=Apply[unop,Rand]

Figure 3.4: CSE Rules 6 and 7: Unary and Binary Operators

CSE Rule 8: Conditional.

	CONTROL	STACK	ENV
CSE Rule 8 (Conditional)	.... δ_{then} δ_{else} β .... δ_{then}	true	
	.... δ_{then} δ_{else} β .... δ_{else}	false	

Figure 3.5: CSE Rule 8: Conditional

CSE Rules 9 and 10: Tuples.

	CONTROL	STACK	ENV
CSE Rule 9 (tuple formation)	.... τ_n	V_1 ... V_n .... $(V_1,...,V_n)$	
CSE Rule 10 (tuple selection)	.... γ	$(V_1,...,V_n)$ I .... V_I	

Figure 3.6: CSE Rules 9 and 10: Tuples

CSE Rule 11: n-ary functions.

	CONTROL	STACK	ENV
CSE Rule 11 (n-ary function)	.... γ .... e_m δ_k	$^c\lambda_k^{V_1,...,V_n}$ Rand .... e_m	e_m=[Rand 1/V_1]... [Rand n/V_n]e_c

Figure 3.7: CSE Rule 11: n-ary Functions

3.4.1 CSE Machine Class

The 'CSEMachine' class is the main class responsible for executing the standardized tree:

```
class CSEMachine:
    """
    Control Structure Environment (CSE) Machine for executing RPAL
   programs.

    Attributes:
        _error_handler (CseErrorHandler): Error handler instance for
   managing errors during execution.
        control_structures (list): List of control structures extracted
    from the Standardized Tree (ST).
```

```
8         environment_tree (Environment): Environment tree representing
  the current execution environment.
9         current_enviroment (Environment): Reference to the current
  environment in the environment tree.
10        stack (Stack): Stack for managing the execution stack.
11        control (Stack): Stack for managing the control structures
  during execution.
12        _linearizer (Linearizer): Linearizer instance for converting
  the ST to linear form.
13        binary_operator (set): Set of binary operators supported by the
   RPAL language.
14        unary_operators (set): Set of unary operators supported by the
  RPAL language.
15        _print_queue (list): List to store the print data as queue
  generated during execution.
16        table_data (list): List to store data for generating the
  execution table.
17     """
18
19     def __init__(self):
20         """
21         Initialize the CSEMachine with necessary components.
22         """
23         # Initialization logic...
24
25     def initialize(self):
26         """
27         Initialize the CSEMachine with necessary components.
28
29          :return: None
30         """
31         # Initialization logic...
32
33     def execute(self, st_tree):
34         """
35         Execute the given Standardized Tree (ST).
36
37         Args:
38             st_tree (Node): The root node of the Standardized Tree (ST)
   to execute.
39         """
40         # Execution logic...
```

Listing 3.7: CSE Machine Class Definition

3.4.2 CSE Rules

The CSE machine executes the standardized tree by applying a set of rules. Each rule corresponds to a specific type of node in the standardized tree. The rules are illustrated in the figures above.

Chapter 4

Function Prototypes and Program Structure

This chapter provides a detailed description of the program structure and function prototypes of the RPAL interpreter implementation. The implementation follows a modular approach, with each component encapsulated in its own class or module.

4.1 Program Structure

The RPAL interpreter is structured as a pipeline of components, each responsible for a specific phase of the interpretation process. The main components are:

1. **Lexical Analyzer**: Converts source code into tokens.
2. **Parser**: Builds an Abstract Syntax Tree (AST) from tokens.
3. **AST Standardizer**: Transforms the AST into a Standardized Tree (ST).
4. **CSE Machine**: Executes the standardized tree and produces output.

The program structure is illustrated in the following diagram:

Source Code	Tokens	AST	ST	Output
	Lexer	Parser	Standardizer	CSE Machine

Figure 4.1: RPAL Interpreter Pipeline

4.2 Directory Structure

The project is organized into the following directory structure:

```
.
├── src/                                # Source code directory
│   ├── lexer.py                       # Lexical analyzer
│   ├── parser.py                      # Parser implementation
│   ├── utils.py                       # Utility functions
│   ├── rpal_ast.py                    # AST node definitions
│   ├── nary_to_lcrs_convertor.py      # N-ary to LCRS tree conversion
│   ├── lcrs_to_nary_convertor.py      # LCRS to N-ary tree conversion
│   ├── __init__.py
│   ├── standerizer/                   # AST standardization
│   │   ├── node.py                    # AST node implementations
│   │   ├── ast.py                     # AST standardization logic
│   │   └── ast_factory.py            # Factory for creating AST nodes
│   └── cse_machine/                   # CSE machine implementation
│       ├── machine.py                 # Main CSE machine implementation
│       ├── error_handler.py           # Error handling utilities
│       ├── cse_error_handler.py       # CSE-specific error handling
│       ├── __init__.py
│       ├── apply_operations/          # Operation implementations
│       │   ├── apply_binary_operations.py
│       │   ├── apply_unary_operations.py
│       │   └── __init__.py
│       ├── utils/                     # Utility functions
│       │   ├── tokens.py              # Token definitions
│       │   ├── util.py                # General utilities
│       │   ├── control_structure_element.py
│       │   ├── stack.py               # Stack implementation
│       │   ├── STlinearizer.py        # Standard tree linearizer
│       │   └── __init__.py
│       └── data_structures/           # Data structure implementations
│           ├── stack.py               # Stack data structure
│           ├── enviroment.py          # Environment management
│           ├── control_structure.py   # Control structure implementation
│           └── __init__.py
├── test-programs/                     # Sample RPAL programs for testing
├── tests/                             # Test suite
│   ├── test_lexer.py                  # Lexer tests
│   ├── test_parser.py                 # Parser tests
│   └── test_parser_basics.py          # Basic parser tests
├── myrpal.py                          # Main interpreter script
├── test_interpreter.py                # Interpreter tests
├── Makefile                           # Build and test automation
└── .gitignore                         # Git ignore file
```

Figure 4.2: Directory Structure

Chapter 5

Testing and Results

This chapter discusses the testing methodology used to verify the correctness of the RPAL interpreter implementation and presents the results of the tests.

5.1 Testing Methodology

The testing methodology for the RPAL interpreter implementation involved several levels of testing:

1. **Unit Testing**: Individual components of the interpreter, such as the lexer, parser, and CSE machine, were tested in isolation to verify their correctness.
2. **Integration Testing**: The components were integrated and tested together to ensure they work correctly as a system.
3. **System Testing**: The complete interpreter was tested with a variety of RPAL programs to verify its correctness and performance.
4. **Regression Testing**: A suite of test cases was developed to ensure that changes to the codebase do not introduce new bugs or regressions.

The test suite includes a variety of RPAL programs that exercise different features of the language, such as arithmetic operations, function definitions, control structures, list operations, and recursive functions.

5.2 Test Cases

The test suite includes the following categories of test cases:

1. **Basic Arithmetic**: Tests for basic arithmetic operations such as addition, subtraction, multiplication, and division.
2. **Function Definitions**: Tests for function definitions and applications, including higher-order functions.
3. **Control Structures**: Tests for control structures such as if-then-else expressions.

4. **List Operations**: Tests for list operations such as concatenation, reversal, and element access.

5. **Recursive Functions**: Tests for recursive function definitions and applications.

6. **Pattern Matching**: Tests for pattern matching capabilities.

7. **Tuples**: Tests for tuple creation and manipulation.

8. **String Operations**: Tests for string operations such as concatenation and character access.

5.3 Results

The RPAL interpreter implementation successfully passed all the test cases in the test suite. The output of the interpreter matches the output of the reference implementation (rpal.exe) for all the test programs.

The implementation correctly handles all the features of the RPAL language, including:

1. **Lexical Analysis**: The lexer correctly tokenizes RPAL source code according to the lexical rules.

2. **Parsing**: The parser correctly builds an Abstract Syntax Tree (AST) according to the grammar rules.

3. **AST Standardization**: The standardizer correctly transforms the AST into a Standardized Tree (ST) according to the standardization rules.

4. **CSE Machine Execution**: The CSE machine correctly executes the standardized tree and produces the expected output.

5.4 Performance Evaluation

The performance of the RPAL interpreter implementation was evaluated by measuring the execution time for various RPAL programs. The implementation shows good performance for small to medium-sized programs, with execution times in the range of milliseconds to seconds.

For larger programs, the execution time increases linearly with the size of the program, which is expected for an interpreter. The implementation is efficient enough for educational purposes and small to medium-sized RPAL programs.

Chapter 6

Conclusion

This chapter summarizes the RPAL interpreter implementation project, discusses the challenges faced during development, suggests potential future improvements, and reflects on the lessons learned.

6.1 Summary

The RPAL interpreter implementation project successfully achieved its objectives of developing a complete interpreter for the RPAL language that follows the standard RPAL language specification. The implementation includes a lexical analyzer, a parser, an AST standardizer, and a CSE machine for program execution.

The implementation is modular, with each component encapsulated in its own class or module, which enhances maintainability and allows for easier testing and debugging of individual components. The implementation is also well-documented, with detailed comments explaining the purpose and functionality of each class and method.

The interpreter correctly handles all the features of the RPAL language, including function definitions and applications, control structures, list operations, recursive functions, pattern matching, and more. The output of the interpreter matches the output of the reference implementation (rpal.exe) for all the test programs.

6.2 Challenges Faced

During the development of the RPAL interpreter, several challenges were encountered:

1. **Grammar Complexity**: The RPAL grammar is complex, with many rules and non-terminals. Implementing a recursive descent parser for such a grammar required careful planning and debugging.

2. **AST Standardization**: The standardization rules for transforming the AST into a standardized form are intricate and required a deep understanding of the RPAL language semantics.

3. **CSE Machine Implementation**: Implementing the CSE machine required a thorough understanding of the execution model and control structures of the RPAL language.

4. **Error Handling**: Implementing robust error handling for lexical, syntax, and runtime errors was challenging and required careful consideration of error messages and recovery strategies.

6.3 Future Improvements

While the current implementation of the RPAL interpreter is complete and functional, there are several potential improvements that could be made in the future:

1. **Performance Optimization**: The current implementation prioritizes clarity and correctness over performance. Future improvements could focus on optimizing the execution speed of the interpreter.

2. **Error Reporting**: The error reporting mechanism could be enhanced to provide more detailed and user-friendly error messages, including suggestions for fixing common errors.

3. **Interactive Mode**: An interactive mode could be added to allow users to enter RPAL expressions and see the results immediately, similar to a REPL (Read-Eval-Print Loop).

4. **Debugging Tools**: Debugging tools, such as a step-by-step execution mode, variable inspection, and breakpoints, could be added to help users debug their RPAL programs.

5. **Standard Library**: A standard library of common functions and utilities could be developed to enhance the usability of the RPAL language.

6.4 Lessons Learned

The development of the RPAL interpreter provided valuable insights and lessons:

1. **Language Design**: The project provided a deeper understanding of programming language design, including lexical rules, grammar, and semantics.

2. **Interpreter Architecture**: The project demonstrated the architecture of a typical interpreter, with distinct phases for lexical analysis, parsing, and execution.

3. **Functional Programming**: The project enhanced understanding of functional programming concepts, such as higher-order functions, recursion, and pattern matching.

4. **Software Engineering**: The project reinforced software engineering principles, such as modularity, encapsulation, and documentation.

In conclusion, the RPAL interpreter implementation project was a valuable learning experience that provided practical insights into programming language design and implementation. The resulting interpreter is a functional and educational tool for understanding the RPAL language and functional programming concepts.