# RPAL Interpreter Project Report


- **Name:** Jayasinghe U.H.N. **Student ID:** 220264H
- **Name:** Ekanayaka L.A.D. **Student ID:** 220155B


## Table of Contents



## 1. Introduction

This report presents a comprehensive analysis and documentation of an RPAL (Right-reference Pedagogical Algorithmic Language) interpreter implementation. RPAL is a functional programming language designed for educational purposes, emphasizing functional programming concepts and principles. The interpreter described in this report is capable of parsing, standardizing, and executing RPAL programs.

The implementation follows a classic interpreter architecture, consisting of several distinct phases: lexical analysis, syntax parsing, tree standardization, and execution using a Control-Stack-Environment (CSE) machine. Each phase is implemented as a separate module, allowing for a clean separation of concerns and making the codebase easier to understand and maintain.

This report aims to provide a detailed explanation of the interpreter's structure, the function prototypes used in each module, and the overall workflow of the system. It serves as documentation for the project and can be used as a reference for understanding the implementation details of the RPAL interpreter.

# 2. Project Overview

The RPAL interpreter project implements a complete interpreter for the RPAL programming language. The interpreter takes an RPAL source file as input, processes it through several stages, and produces the execution result as output. The project is structured into multiple modules, each responsible for a specific phase of the interpretation process.

## Key Features

The interpreter includes the following key features:

1. **Lexical Analysis:** Converts RPAL source code into tokens.
2. **Syntax Parsing:** Builds an Abstract Syntax Tree (AST) from the tokens.
3. **Tree Standardization:** Transforms the AST into a standardized tree.
4. **CSE Machine Execution:** Executes the standardized tree to produce the output.
5. **Command-Line Interface:** Provides options for displaying intermediate representations.

## Implementation Language

The interpreter is implemented in Python, a high-level, interpreted programming language known for its readability and expressiveness. Python's object-oriented features and rich standard library make it an excellent choice for implementing language processors.

## Project Structure

The project is organized into the following main directories:

1. **Lexer:** Contains the lexical analyzer module.
2. **Parser:** Contains the syntax parser and AST string converter modules.
3. **Standardizer:** Contains the tree standardization module.
4. **CSE Machine:** Contains the CSE machine implementation and related modules.

The main entry point of the interpreter is the `myrpal.py` file, which orchestrates the entire interpretation process.

# 3. Program Structure

The RPAL interpreter follows a modular architecture, with each module responsible for a specific phase of the interpretation process. This section provides an overview of the program structure, including the main components and their interactions.

## Overall Architecture

The interpreter follows a classic compiler/interpreter architecture with the following main components:

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

## Workflow

The workflow of the interpreter is as follows:

1. **Input:** The interpreter reads an RPAL source file.
2. **Lexical Analysis:**
    - The `tokenize` function in `token_analyzer.py` converts the source code into a list of tokens.
    - Each token has a category (KEYWORD, IDENTIFIER, NUMBER, TEXT, OPERATOR, PUNCTUATION, EOF) and a value.
3. **Parsing:**
    - The `SyntaxParser` class in `syntax_parser.py` builds an Abstract Syntax Tree (AST) from the tokens.
    - The parser implements a recursive descent parsing algorithm following the RPAL grammar.
    - The AST is represented as a list of `Node` objects, each with a type, value, and number of children.
4. **AST to String Conversion:**
    - The `StringAst` class in `StringAst.py` converts the AST to a string representation.
    - This string representation is used for display or further processing.
5. **Standardization:**
    - The `TreeBuilder` class in `tree_builder.py` builds a standardized tree from the string representation of the AST.
    - The standardized tree is a simplified form of the AST that is easier to execute.

6. **CSE Machine Execution:**
    - The `CSEMachineFactory` class in `factory.py` creates a CSE Machine from the standardized tree.
    - The CSE Machine consists of a control list, a stack, and an environment.
    - The `CSEMachine` class in `csemachine.py` executes the control list to produce the output.
    - The execution follows the Control-Stack-Environment (CSE) model.
7. **Output:** The interpreter prints the result of the execution.

## Directory Structure

The project is organized into the following directory structure:

```
Rpal_interpreter/
├── Lexer/
│   └── token_analyzer.py
├── Parser/
│   ├── StringAst.py
│   └── syntax_parser.py
├── Standardizer/
│   ├── tree_builder.py
│   ├── tree_node.py
│   └── tree.py
├── cse machine/
│   ├── csemachine.py
│   ├── factory.py
│   └── symbols.py
└── myrpal.py
└── Makefile
```

## Data Flow

The data flow through the interpreter is as follows:

1. The source code is read from a file and passed to the lexical analyzer.
2. The lexical analyzer produces a list of tokens.
3. The parser consumes the tokens and produces an AST.
4. The AST is converted to a string representation.
5. The standardizer consumes the string representation and produces a standardized tree.
6. The CSE machine factory creates a CSE machine from the standardized tree.
7. The CSE machine executes the standardized tree and produces the output.

### Command-Line Interface

The interpreter provides a command-line interface with the following options:

- `source_file`: Path to the RPAL source file to interpret (required).
- `-ast`: Display the Abstract Syntax Tree and exit.
- `-st`: Display the Standardized Tree and exit.

This interface allows users to interact with the interpreter and view intermediate representations of the program.

## 4. Function Prototypes

This section presents the function prototypes for the key functions and classes in each module of the RPAL interpreter. These prototypes show the structure of the programs and provide an overview of the functionality implemented in each module.

### Main Module (`myrpal.py`)

```
def main():
    """Main entry point for the RPAL interpreter."""
    # Parses command-line arguments, reads source file, and
    # orchestrates the interpretation process
```

### Lexer Module (`token_analyzer.py`)

```
class TokenCategory(Enum):
    """Enumeration of token categories in RPAL language."""
    # Defines token categories: KEYWORD, IDENTIFIER, NUMBER,
    # TEXT, OPERATOR, PUNCTUATION, EOF

class Token:
    """Represents a token in the RPAL language."""
    def __init__(self, category, value, line=0, column=0):
        """Initialize a new Token instance."""

    def __str__(self):
        """Return a string representation of the token."""

    def get_category(self):
        """Get the category of the token."""

    def get_value(self):
        """Get the string value of the token."""
```

```
    def get_position(self):
        """Get the position (line, column) of the token."""

def tokenize(source_code):
    """Convert RPAL source code into a list of tokens."""
    # Processes source code and returns a list of Token objects
```

### Parser Module (`syntax_parser.py`)

```
class NodeType(Enum):
    """Enumeration of node types in the Abstract Syntax Tree."""
    # Defines node types: let, fcn_form, identifier, integer,
    # string, etc.

class Node:
    """Represents a node in the Abstract Syntax Tree."""
    def __init__(self, node_type, value, children):
        """Initialize a new Node instance."""

class SyntaxParser:
    """Parser for RPAL language."""
    def __init__(self, tokens):
        """Initialize a new SyntaxParser instance."""

    def parse(self):
        """Parse the tokens and build an Abstract Syntax
Tree."""
        # Various parsing methods for different grammar rules

    def E(self): """Parse an expression."""
    def Ew(self): """Parse a 'where' expression."""
    def T(self): """Parse a tuple expression."""
    def Ta(self): """Parse an 'aug' expression."""
    def Tc(self): """Parse a conditional expression."""
    def B(self): """Parse a boolean expression."""
    def Bt(self): """Parse a boolean term."""
    def Bs(self): """Parse a boolean factor."""
    def Bp(self): """Parse a boolean primary."""
    def A(self): """Parse an arithmetic expression."""
    def At(self): """Parse an arithmetic term."""
    def Af(self): """Parse an arithmetic factor."""
    def Ap(self): """Parse an '@' expression."""
    def R(self): """Parse a rator-rand expression."""
    def Rn(self): """Parse a rand."""
    def D(self): """Parse a definition."""
    def Da(self): """Parse an 'and' definition."""
    def Dr(self): """Parse a 'rec' definition."""
    def Db(self): """Parse a basic definition."""
    def Vb(self): """Parse a variable binding."""
```

### AST String Converter Module (`StringAst.py`)

```
class StringAst:
    """Represents a String Abstract Syntax Tree (AST) node."""
    def __init__(self, parser):
        """Initialize a new StringAst instance."""

    def convert_ast_to_string_ast(self):
        """Convert the AST to a string representation."""

    def add_strings(self, dots, node):
        """Add string representations of nodes to the string
AST."""
```

### CSE Machine Module (`csemachine.py`)

```
class CSEMachine:
    """CSE Machine for evaluating standardized RPAL programs."""
    def __init__(self, control, stack, environment):
        """Initialize a new CSE Machine instance."""

    def execute(self):
        """Execute the CSE Machine."""

    def convert_string_to_bool(self, data):
        """Convert string representation of boolean to Python
boolean."""

    def apply_unary_operation(self, rator, rand):
        """Apply a unary operation to an operand."""

    def apply_binary_operation(self, rator, rand1, rand2):
        """Apply a binary operation to two operands."""

    def get_tuple_value(self, tup):
        """Get string representation of a tuple."""

    def get_answer(self):
        """Execute the CSE Machine and get the final result."""
```

### CSE Machine Factory Module (`factory.py`)

```
class CSEMachineFactory:
    """Factory class for creating CSE Machine instances from
standardized trees."""
    def __init__(self):
```

```
        """Initialize a new CSEMachineFactory instance."""

    def get_symbol(self, node):
        """Convert a tree node to the appropriate CSE Machine
symbol."""

    def get_b(self, node):
        """Create a B symbol (conditional block) from a tree
node."""

    def get_lambda(self, node):
        """Create a Lambda symbol from a tree node."""

    def get_pre_order_traverse(self, node):
        """Traverse a tree node in pre-order and convert to CSE
Machine symbols."""

    def get_delta(self, node):
        """Create a Delta symbol from a tree node."""

    def get_control(self, ast):
        """Create the control list for the CSE Machine from an
AST."""

    def get_stack(self):
        """Create the initial stack for the CSE Machine."""

    def get_environment(self):
        """Create the initial environment for the CSE
Machine."""

    def get_cse_machine(self, ast):
        """Create a CSE Machine instance from an AST."""
```

### Symbols Module (`symbols.py`)

```
class Symbol:
    """Base class for all symbols in the CSE machine."""
    def __init__(self, data):
        """Initialize a new Symbol instance."""

    def set_data(self, data):
        """Set the data of the symbol."""

    def get_data(self):
        """Get the data of the symbol."""

# Various symbol classes inheriting from Symbol
class Rand(Symbol): """Base class for all rand (operand)
symbols."""
```

```
class Rator(Symbol): """Base class for all rator (operator)
symbols."""
class B(Symbol): """B symbol for conditional expressions."""
class Beta(Symbol): """Beta symbol for conditional branching."""
class Bool(Rand): """Boolean value symbol."""
class Bop(Rator): """Binary operator symbol."""
class Delta(Symbol): """Delta symbol for code blocks."""
class Dummy(Rand): """Dummy value symbol."""
class E(Symbol): """Environment symbol."""
class Err(Symbol): """Error symbol."""
class Eta(Symbol): """Eta symbol for recursion."""
class Gamma(Symbol): """Gamma symbol for function
application."""
class Id(Rand): """Identifier symbol."""
class Int(Rand): """Integer value symbol."""
class Lambda(Symbol): """Lambda symbol for function
definitions."""
class Str(Rand): """String value symbol."""
class Tau(Symbol): """Tau symbol for tuple creation."""
class Tup(Rand): """Tuple value symbol."""
class Uop(Rator): """Unary operator symbol."""
class Ystar(Symbol): """Y* symbol for recursion."""
```

These function prototypes provide a comprehensive overview of the structure and functionality of the RPAL interpreter. Each module contains classes and functions that work together to implement the different phases of the interpretation process, from lexical analysis to execution.

## 5. Module Descriptions

This section provides detailed descriptions of each module in the RPAL interpreter, explaining their purpose, functionality, and implementation details.

### Main Module (`myrpal.py`)

The main module serves as the entry point for the RPAL interpreter. It orchestrates the entire interpretation process by coordinating the different phases: lexical analysis, parsing, standardization, and execution.

**Key Features:**

- Command-line argument parsing using the `argparse` module.
- File I/O operations for reading the source code.
- Error handling for file not found and other exceptions.
- Coordination of the interpretation phases.
- Display options for intermediate representations (AST and standardized tree).

The main function follows a clear workflow:

1. Parse command-line arguments.
2. Read the source file.
3. Tokenize the source code.
4. Parse the tokens into an AST.
5. Convert the AST to a string representation.
6. Display the AST if requested.
7. Build and standardize the tree.
8. Display the standardized tree if requested.
9. Build and execute the CSE machine.
10. Print the result.

This modular approach allows for easy debugging and extension of the interpreter.

## Lexer Module (`token_analyzer.py`)

The lexer module is responsible for converting the source code into a list of tokens. It implements a lexical analyzer that recognizes the various token types in the RPAL language.

**Key Features:**

- Token categorization using the `TokenCategory` enumeration.
- Token representation using the `Token` class.
- Regular expression-based pattern matching for token recognition.
- Position tracking for error reporting.
- Comment handling.
- String literal handling with escape sequence support.

The tokenization process involves scanning the source code character by character and identifying patterns that match the different token types. The lexer handles whitespace, comments, identifiers, keywords, numbers, strings, operators, and punctuation.

The `tokenize` function is the main entry point for the lexer. It takes the source code as input and returns a list of `Token` objects. Each token contains information about its category, value, and position in the source code.

## Parser Module (`syntax_parser.py`)

The parser module is responsible for building an Abstract Syntax Tree (AST) from the list of tokens produced by the lexer. It implements a recursive descent parser that follows the RPAL grammar.

**Key Features:**

- Node type categorization using the `NodeType` enumeration.
- Node representation using the `Node` class.
- Recursive descent parsing methods for different grammar rules.
- Error reporting for syntax errors.
- AST construction as a list of `Node` objects.

The parsing process involves recursively applying grammar rules to the token stream. Each grammar rule is implemented as a method in the `SyntaxParser` class. These methods consume tokens from the input stream and produce nodes in the AST.

The parser handles various language constructs, including:

- Expressions ( `E` , `Ew` , `T` , `Ta` , `Tc` , etc.)
- Boolean expressions ( `B` , `Bt` , `Bs` , `Bp` )
- Arithmetic expressions ( `A` , `At` , `Af` , `Ap` )
- Rators and rands ( `R` , `Rn` )
- Definitions ( `D` , `Da` , `Dr` , `Db` )
- Variable bindings ( `Vb` )

The `parse` method is the main entry point for the parser. It takes the tokens as input and returns the root of the constructed AST.

## Standardizer Module

The Standardizer module is responsible for transforming the Abstract Syntax Tree (AST) into a standardized tree. This standardized tree is a simplified and optimized representation of the program, making it easier for the CSE Machine to execute.

**Key Features:**

- **Tree Simplification:** Reduces the complexity of the AST by applying various transformation rules.
- **Intermediate Representation:** Creates an intermediate representation that bridges the gap between the parser's output and the CSE Machine's input.
- **Error Handling:** Ensures that the standardization process handles invalid or malformed AST structures gracefully.

The standardization process involves traversing the AST and applying a set of predefined rules to convert complex constructs into simpler, equivalent forms. This step is crucial for optimizing the execution process and ensuring consistency.

### CSE Machine Module

The CSE Machine module is the core of the RPAL interpreter, responsible for executing the standardized tree. It implements a Control-Stack-Environment (CSE) machine, which is a common model for executing functional programming languages.

**Key Features:**

- **Control List:** Manages the sequence of operations to be performed.
- **Stack:** Stores intermediate results and operands during execution.
- **Environment:** Manages variable bindings and their scopes.
- **Execution Cycle:** Implements the main execution loop, processing control list elements, manipulating the stack, and updating the environment.
- **Built-in Operations:** Supports various built-in operations, including arithmetic, logical, and comparison operations.

The CSE Machine operates by repeatedly fetching an element from the control list, interpreting it, and performing the corresponding action on the stack and environment. This process continues until the control list is empty, and the final result is left on the stack.

## 6. Conclusion

This report has provided a detailed overview of the RPAL interpreter project, covering its architecture, key features, program structure, function prototypes, and module descriptions. The interpreter successfully implements the core functionalities of an RPAL interpreter, from lexical analysis to execution using a CSE machine.

The modular design of the interpreter allows for easy maintenance, debugging, and future extensions. The use of Python as the implementation language further enhances its readability and extensibility. This project serves as a solid foundation for understanding the principles of functional programming language interpretation.

Further improvements could include:

- Adding more advanced language features to RPAL.
- Implementing a more sophisticated error reporting and recovery mechanism.
- Optimizing the CSE machine for better performance.
- Developing a graphical user interface for easier interaction.

This report concludes the documentation of the RPAL interpreter project, providing a comprehensive guide for its understanding and future development.