Authors:
Dasun W.A.T. – 220094K
Diwakar J.S.P. – 220144P

RPAL Interpreter

Prepared by Group **Cyclone_01** of Intake-22 following CS3513
Department of Computer Science & Engineering,
University of Moratuwa.
27/05/2025

Table of Contents

Introduction

This document outlines the development of an RPAL(Right-reference Pedagogic Algorithmic Language) interpreter using Python, focusing on key phases of parsing and executing RPAL programs. The interpreter is based on the lexical conventions and syntactic structures defined in the RPAL language. The main goal of this project is to create a comprehensive software program capable of parsing RPAL code, generating an Abstract Syntax Tree (AST), converting it into a Standardized Tree (ST), and executing it to get the relevant output.

Problem Description

The objective of this project is to design and implement a complete interpreter for RPAL programming language. This includes a lexical analyzer, parser, abstract syntax tree (AST) generation, AST-to-standardized tree (ST) transformation, and a control stack environment for evaluation.

The lexical analyzer (lexer) should tokenize the input source code based on the lexical rules outlined in RPAL_Lex.pdf, identifying valid tokens such as identifiers, keywords, literals, and operators. The parser will be implemented using recursive descent or another manual method without relying on automated tools such as lex, yacc, or similar libraries. The parser will generate an Abstract Syntax Tree (AST) that represents the syntactic structure of the input RPAL program.

Following parsing, the AST must be transformed into a Standardized Tree (ST) format using a defined transformation algorithm. This transformation simplifies the AST into a more uniform and minimal form suitable for interpretation. Finally, a CSE machine will be implemented to evaluate the standardized tree according to the operational semantics of RPAL.

The system should accept an RPAL program as input from a file and produce output that matches the structure and format generated by the reference implementation rpal.exe.

About Our Solution

Development & Testing Tools: Visual Studio Code , PyCharm IDE and Command Line

The RPAL interpreter was developed as a modular tree-walking interpreter, implemented in Python. The overall system follows a structured pipeline consisting of lexical analysis, parsing, AST construction, AST standardization into a simplified syntax tree (ST), and finally, interpretation of the program by recursively evaluating the ST.

Input and Output

The program reads an input file containing RPAL program code. It accepts the -ast switch to print the Abstract Syntax Tree (AST) of the input program. In addition, -st switch prints the standardized tree which consists of lambdas and gammas.

Lexical Analysis

A custom lexical analyzer was implemented without using external tools like lex or re generators. Instead, Python's regular expressions were used to define and match token types such as identifiers, keywords, literals, operators, and punctuation.

The output of the lexical analyzer is a flat sequence of token objects, each capturing its type and value. These tokens form the input to the parser.

Parsing

The parser was implemented using the recursive descent parsing technique, without using 'lex' or 'yacc' or any similar tool, adhering closely to the original RPAL grammar. Each production rule from the grammar corresponds to a parsing function (e.g., parse_E, parse_B, parse_T, etc.).

The parser directly constructs an Abstract Syntax Tree (AST) using well-defined node classes. This structure captures the hierarchical syntactic structure of the program in a form suitable for semantic processing.

AST and Standardized Tree (ST) Generation

Each AST node class includes a standardize() method. This method transforms complex RPAL constructs (such as let, where, within, etc.) into simpler, canonical forms using only basic constructs like lambda, gamma, tau, and ->.

The result is a Standardized Tree (ST) that removes syntactic sugar and simplifies further processing. This transformation enables easier interpretation and reflects the core operational semantics of the RPAL language.

Both the AST and the ST can be printed and visually inspected, providing transparency and aiding in debugging and validation.

Interpretation

The final phase involves interpreting the standardized tree using a tree-walking approach. In this model, each node in the ST interprets itself by evaluating its children recursively and applying its semantic rule.

A dynamic environment model was implemented to handle variable scopes and closures. Function calls, lambda abstractions, and tuple access are handled using lexical scoping and nested environments, mimicking the behavior expected from the CSE machine.

Although a tree-walking approach was adopted instead of explicitly implementing a classical stack-based CSE machine, the behavioral essence of the CSE machine is preserved:

- A form of control structure exists implicitly in the recursive node calls.
- Environments are created and managed dynamically, and closures maintain references to parent environments.
- Recursive function application and evaluation order mirror the behavior of a formal CSE machine.

Hence, the tree-walking interpreter simulates the core principles of a CSE machine, even if it is not structurally stack-based. If a stricter interpretation is required, the project is modular enough to be extended with an explicit CSE machine layer using the same AST/ST structures.

Language Choice

Python was chosen as the implementation language due to:

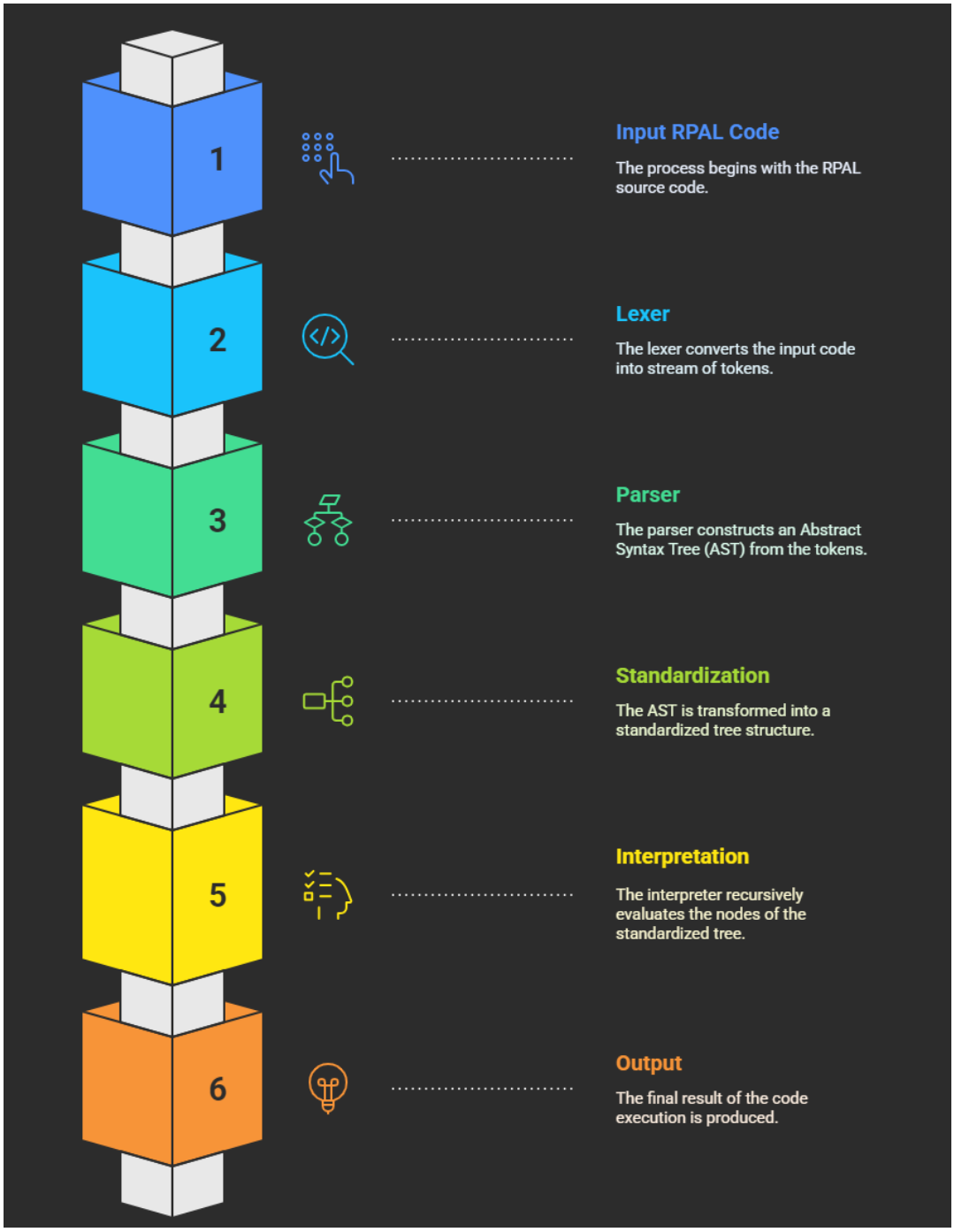
- Its readability and expressiveness, which supports rapid development and testing.
- Built-in features such as dynamic typing, object-oriented support, and regular expressions that simplify the implementation of interpreters.
- Ease of handling recursive functions and dynamic data structures, which are essential for this project.

Furthermore, Python's simplicity makes it easier to debug, maintain, and extend the project in future iterations.

Program Structure and Function Prototypes

The program goes through several stages: lexical analysis, parsing, AST generation, standardization, and interpretation. Each phase is handled by separate components organized in a modular architecture.

4.1 Program Structure



Module	Description
myrpal.py	Entry point. Handles command-line input, invokes lexer, parser, and interpreter.
Lexer.py	Contains the lexer that converts input into a stream of tokens.
parser.py	Builds the Abstract Syntax Tree (AST) from tokens using recursive descent.
nodes.py	Defines the node classes used for AST and Standardized Trees.
environment.py	Implements the environment for variable/function bindings and built-in functions.
data_types.py	Consists of classes for essential data types for the execution of a RPAL program.

4.2 Function Prototypes and Class Signatures

myrpal.py

This is the entry point of the program, coordinating the Lexer, Parser, and executing standardization and interpretation.

- **Function: read_file(filename: str) -> str**

  This reads the RPAL source file.

- **Function: main() -> None**

  Entry point that orchestrates all stages and handles the flags.

Lexer.py

This file contains the Token class for representing lexical tokens and the Lexer class for performing lexical analysis.

It tokenizes the input RPAL program based on defined lexical rules.

Functionality:

Identifies tokens based on RPAL lexical rules.

Outputs a token list containing token objects with type and value attributes.

Input:

RPAL source code as a string.

Output:

Token list (list of token objects).

- **Class: Token**
  - __init__(self, type_: str, value: str)
  - __str__(self) -> str
  - __repr__(self) -> str
- **Class: Lexer**
  - __init__(self, code: str)
  - tokenize(self) -> None

---

parser.py

This file contains the Parser class, which is responsible for building the AST from the tokens. This class got methods starting with parse_ correspond to each grammar rule of RPAL grammar.

Functionality

1. Receives the token list from the Lexical Analyzer.
2. Construct the Abstract Syntax Tree (AST) based on the provided tokens.
3. Outputs the Abstract Syntax Tree (AST).

Input

Token list generated by the Lexical Analyzer

Output

Abstract SyntaxTree (AST)

- **Class: Parser**
  - __init__(self, tokens: list)
  - peek(self) -> Token | None
  - match(self, expected: str) -> Token | None

- reversePos(self)
- expect(self, expected: str) -> Token

  methods correspond to each grammar rule :

  parse_E(self) -> Node

  parse_Ew(self) -> Node

  parse_T(self) -> Node etc.

---

nodes.py

This file defines the base Node class for the Abstract Syntax Tree (AST) and all the concrete node types. Each node type (e.g., LetNode, LambdaNode, GammaNode, ArithmeticNode, etc.) implements standardize() and interpret() based on the semantics of that construct.

Standardize(): Converts the AST to a simplified form (Standardized Tree)

For each node class , standardize method is used to standardize that node with its children nodes. It constructs the Standard Tree (ST) based on the provided string list and provide the root node of it.

Interpret(): Evaluates the node using a tree-walking interpreter.

Each node in the ST interprets itself by evaluating its children recursively and applying its semantic rule using this method.

- **Abstract Class: Node(ABC)**
  - __init__(self, type_: str, value: Any)
  - standardize(self) -> 'Node' (Abstract Method)
  - print(self, indent: int = 0) -> None (Abstract Method)
  - __str__(self) -> str
  - __repr__(self) -> str
- **Concrete Node Classes (Inherit from Node)**
  - STLambdaNode(Vb: Any, Exp: Any)
    - __init__(self, Vb, Exp)
    - standardize(self)
    - interpret(self,env)

        - print(self, indent=0)
        - __str__(self)
    - LetNode(D: 'Node', E: 'Node')
        - __init__(self, D, E)
        - standardize(self)
        - interpret(self,env)
        - print(self, indent=0)
        - __str__(self)
    - LambdaNode(Vbs: list, E: 'Node')
        - __init__(self, Vbs, E)
        - standardize(self)
        - interpret(self,env)
        - print(self, indent=0)
        - __str__(self)

    Etc.

---

environment.py

This file defines Environment ,Closure and BuiltInFunction classes implementing the environment for variable/function bindings and built-in functions.

Environment: Stores variable bindings and supports lexical scoping via parent environments.

BuiltInFunction: Handles built-in operations like Print, Order, Isinteger, etc.

Class Environment

- def __init__(self, parent: Optional[Environment] = None)
- def define(self, name: str, value: Any)
- def lookup(self, name: str) -> Any
- def defineBuiltInFunctions(self)
- def __str__(self) -> str
- def __repr__(self) -> str

Class Closure

- def __init__(self,lambdaNode,env)
- def __str__(self) -> str
- def __repr__(self) -> str

Class BuiltInFunction

- def __init__(self, name: str,arity:int,args_recieved)
- def execute(self, arg_value: Any) -> Any
- def __call_(self,*args) -> str
- def __str__(self) -> str
- def __repr__(self) -> str

---

data_types.py

This file consists of classes for essential data types for the execution of a RPAL program.

Class Tuple

- def __init__(self, elements=None):
- def __str__(self) -> str
- def __repr__(self) -> str
- def __len__(self) -> int
- def __getitem__(self, index) -> any
- def __iter__(self) -> any
- def add(self, other) -> Tuple
- def isEmpty(self) -> Boolean

Class TruthValue

- def __init__(self, value):
- def __str__(self) -> str
- def __repr__(self) -> str
- def __bool__(self):

Class Nil

- def __str__(self) -> str
- def __repr__(self) -> str
- def __bool__(self) -> boolean
- def isEmpty(self) -> boolean
- def __eq__(self, other) -> Boolean

4.3 Folder Structure

RPAL-lexical-analyzer-and-parser/

```
│
├── __pycache__/          # Python cache directory for compiled bytecode
├── .idea/                # IDE project config files
├── .pytest_cache/        # Pytest cache files
├── tests/                # Unit test directory for testing components
│
├── .gitignore            # Git ignore file for version control
├── data_types.py         # Contains classes or structures for data representations (e.g., tokens, nodes)
├── environment.py        # Defines the runtime environment
├── Lexer.py              # Implements the lexical analyzer (lexer/tokenizer)
├── Makefile              # Automates running the interpreter, printing AST/ST, testing, and cleaning build files.
├── myrpal.py             # Main driver script that connects all components (entry point)
├── nodes.py              # Defines node classes and related utility functions
├── parser.py             # Implements the recursive descent parser for RPAL
├── testcode.rpal         # Sample RPAL program used for testing and output comparison
```

Program Execution Instructions

Prerequisites

- Ensure that Python and pip are installed on your local machine.

To use the RPAL-Interpreter, follow these steps:

- Clone The repository to your local machine or download the project source code as a ZIP file.
- Navigate to the root directory of the project in the command line interface.
- Place your RPAL test programs within the Inputs folder in the root directory. An example test program is provided. Replace it with your program.
- Running the Program
- Run the main script myrpal.py with the file name as an argument:
    - *python myrpal.py [file_name]*
- To generate the Abstract Syntax Tree (AST):
    - *python myrpal.py -ast [file_name]*
- To generate the Standardized Tree:
    - *python myrpal.py -st file_name*
- To run tests:
    - *python -m pytest tests/*

Conclusion

The RPAL interpreter represents a well-structured approach to parsing and executing RPAL programs. It demonstrates a strong focus on modular design, maintainability, and extensibility, showcasing sound design choices throughout its development.

Lexical analysis utilizes regular expressions from python re library, while a recursive descent method is used during parsing to create abstract syntax trees, effectively supporting the interpretation of even intricate code structures. Each node class consist of a standardize method to convert itself from complex RPAL constructs to a standardized form. Finally, each node in the ST interprets itself by evaluating its children recursively and applying its semantic rule.

This project reflects excellent collaboration and proficient technical capabilities, making it a valuable resource for anyone new to programming language implementation. Its modular nature guarantees adaptability and the possibility of incorporating future improvements based on feedback from users or the community.

References

- RPAL_Lex.pdf
- RPAL_Grammar.pdf
- RPAL Language Documentation (https://sourceforge.net/projects/resil/)
- CS3513 Course Materials
- Python Documentation