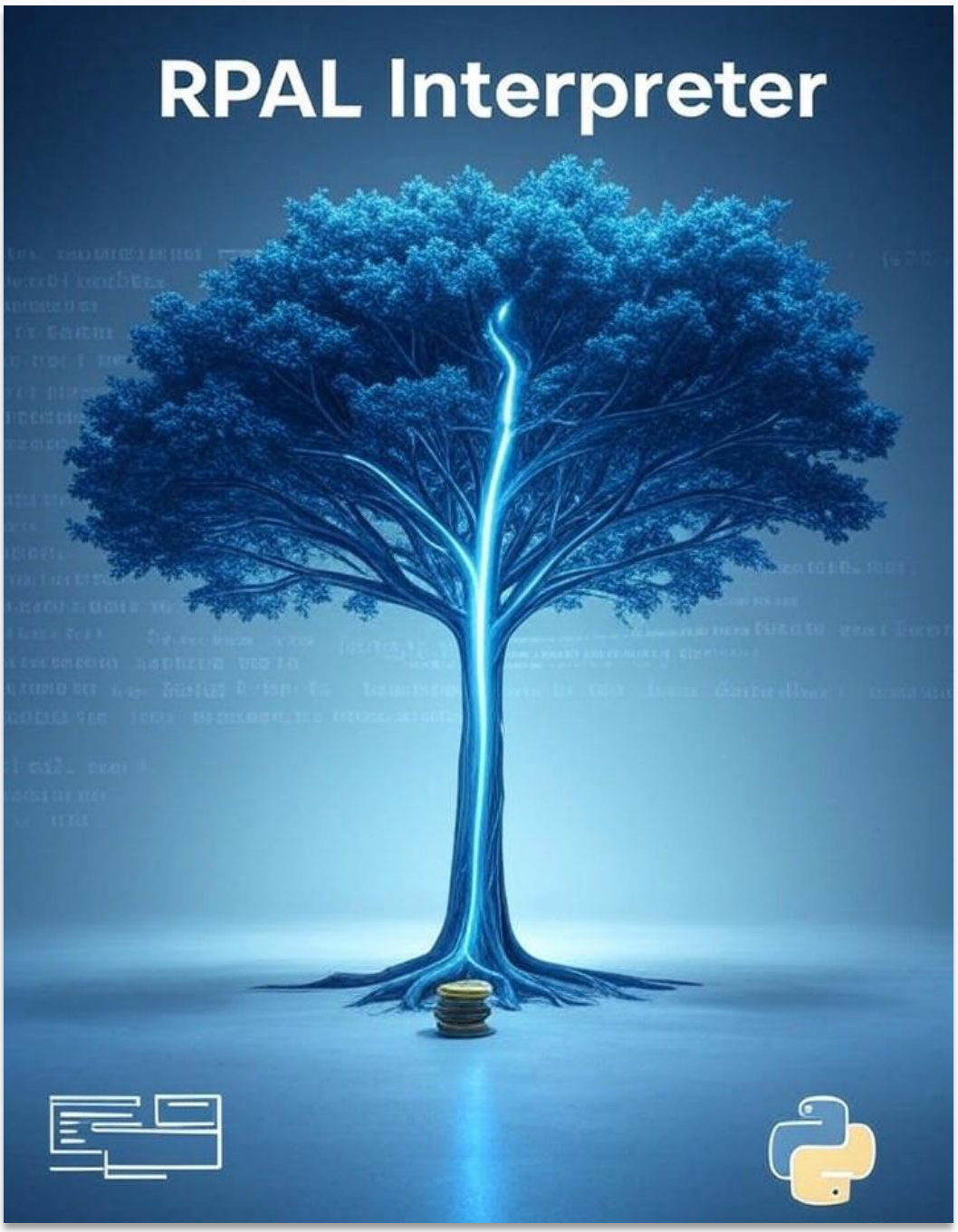

RPAL Interpreter

Team VORTEX

CS3513 | Programming Languages | 02/06/2025

Team VORTEX:

- Edirisinghe E.K.K.D.R. – 220152M
- Jayasinghe J.P.T.S – 220262B

Contents

Overview

This RPAL Interpreter is a Python-based implementation that parses, standardizes, and executes programs written in the RPAL programming language. The interpreter will go through a series of steps to process the RPAL code. The steps are tokenization, parsing to an Abstract Syntax Tree (AST), standardizing the AST, and executing through a Control Stack Environment (CSE) machine. The project supports command-line arguments for display of source code, AST, and standardized tree (ST), or execute a program and output results. It is designed as a modular implementation for maintainability and understandability and has different components for scanning, parsing, and standardizing.

Program Structure

The RPAL interpreter is organized into several Python modules. Each is responsible for a specific phase of the interpretation process. Below is the structure of the program:

- **scanner.py**
    - This handles lexical analysis by tokenizing the input RPAL code and filtering out unwanted tokens. It defines token types. It also uses regular expressions to identify tokens such as keywords, identifiers, operators, integers, strings, and punctuation.
- **parser.py**
    - This implements a recursive descent parser to construct an AST from the tokenized input. This process is based on the RPAL grammar. It processes expressions, definitions, and other constructs to build a tree structure using the Node class.
- **standardiser.py**
    - This transforms the AST into a ST by applying transformation rules (e.g.: converting let, where, function_form, and within into lambda, gamma, and other nodes). This ensures the tree is in a form suitable for CSE machine to process.
- **cse_machine.py**
    - This implements the CSE machine. It executes the ST by generating control structures and applying evaluation rules. It manages environments and a stack to handle variable bindings and computations.
- **myrpal.py**
    - This file serves as the main entry point. It carries out the entire process from reading input to producing output. It handles command-line arguments (-l, -ast, -st) to control the output (source code, AST, standardized tree, or the final result).

- **data_structures**

  This folder contains supporting classes and enums:

  - **stack.py** defines a Stack class for managing tokens during parsing and values during CSE execution.
  - **node.py** defines a Node class for representing AST nodes, with methods for tree traversal and printing.
  - **cse_enviroment.py** defines an Environment class for managing variable bindings and environment hierarchies in the CSE machine.
  - **enums.py** defines the TokenType enum for categorizing tokens.
  - **control_structures.py** defines classes (Delta, Tau, Lambda, Eta) for control structures used in the CSE machine.

The program flow is as follows:

1. **Input Reading**
   - myrpal.py reads the RPAL source code from a file specified via command-line arguments.
2. **Tokenization**
   - scanner.py converts the code into a list of tokens, filtering out whitespace and comments.
3. **Parsing**
   - parser.py builds an AST from the tokens using recursive descent parsing.
4. **Standardization**
   - standardiser.py transforms the AST into a standardized tree.
5. **Execution**
   - cse_machine.py generates control structures, applies evaluation rules, and produces the final output.
6. **Output**
   - Depending on command-line flags, myrpal.py outputs the source code, AST, standardized tree, or the execution result.

FUNCTION PROTOTYPES

Below are the function prototypes for key components across the modules. This section highlights their structure and purpose. Each prototype includes the function signature, parameters, return type, and a brief description.

scanner.py

```
def tokenize_and_screen(code: str) -> list[tuple[TokenType, str]]:
    """
    Tokenizes the input RPAL code and filters out whitespace and
comments.
    Parameters:
        code: The input RPAL source code as a string.
    Returns:
        A list of tuples, each containing a TokenType enum and the token
value.
    """
```

parser.py

```
def read(expected_token_type: TokenType, expected_token: str) -> None:
    """
    Reads the next token and verifies it matches the expected type and
value.
    Parameters:
        expected_token_type: The expected token type (from TokenType
enum).
        expected_token: The expected token value.
    Raises:
        SyntaxError: If the token does not match or input ends
unexpectedly.
    """


def build_tree(value: str, num_children: int) -> None:
    """
    Builds an AST node with the specified value and number of children.
    Parameters:
        value: The node's value (e.g., 'let', 'lambda', '<ID:name>').
        num_children: The number of child nodes to pop from the stack.
    """
```

```
def parse(token_list: list[tuple[TokenType, str]]) -> Node:
    """
    Parses a list of tokens into an AST.
    Parameters:
        token_list: List of tokens from tokenize_and_screen.
    Returns:
        The root Node of the AST.
    Raises:
        SyntaxError: If tokens are invalid or unprocessed tokens remain.
    """


def E() -> None:
    """
    Parses an Expression (E) according to RPAL grammar (e.g., 'let', 'fn',
or Ew).
    Builds corresponding AST nodes (e.g., 'let', 'lambda').
    """


def Ew() -> None:
    """
    Parses an Expression with 'where' clause (Ew).
    Builds 'where' node if applicable.
    """


def T() -> None:
    """
    Parses a Tuple expression (T).
    Builds 'tau' node for comma-separated expressions.
    """


def Ta() -> None:
    """
    Parses a Tuple atom (Ta).
    Builds 'aug' node for augmentation expressions.
    """
```

```
def Tc() -> None:
    """
    Parses a Conditional expression (Tc).
    Builds '->' node for conditional expressions.
    """

def B() -> None:
    """
    Parses a Boolean expression (B).
    Builds 'or' node for logical OR operations.
    """

def Bt() -> None:
    """
    Parses a Boolean term (Bt).
    Builds '&' node for logical AND operations.
    """

def Bs() -> None:
    """
    Parses a Boolean subexpression (Bs).
    Builds 'not' node for negation.
    """

def Bp() -> None:
    """
    Parses a Boolean primary (Bp).
    Builds nodes for comparison operators (e.g., 'gr', 'ge', 'ls', 'le',
'eq', 'ne').
    """

def A() -> None:
    """
    Parses an Arithmetic expression (A).
    Builds 'neg', '+', or '-' nodes for unary and binary operations.
```

```
    """


def At() -> None:
    """
    Parses an Arithmetic term (At).
    Builds '*' or '/' nodes for multiplication and division.
    """


def Af() -> None:
    """
    Parses an Arithmetic factor (Af).
    Builds '**' node for exponentiation.
    """


def Ap() -> None:
    """
    Parses an Arithmetic primary (Ap).
    Builds '@' node for application expressions.
    """


def R() -> None:
    """
    Parses a Rator (R).
    Builds 'gamma' node for function applications.
    """


def Rn() -> None:
    """
    Parses a Rator or Rand (Rn).
    Builds nodes for identifiers, integers, strings, booleans, nil, dummy,
or parenthesized expressions.
    """
```

```
def D() -> None:
    """
    Parses a Definition (D).
    Builds 'within' node for nested definitions.
    """


def Da() -> None:
    """
    Parses a Definition atom (Da).
    Builds 'and' node for simultaneous definitions.
    """


def Dr() -> None:
    """
    Parses a Definition recursive (Dr).
    Builds 'rec' node for recursive definitions.
    """


def Db() -> None:
    """
    Parses a Definition binding (Db).
    Builds '=' or 'function_form' nodes for bindings and function
definitions.
    """


def Vb() -> None:
    """
    Parses a Variable binding (Vb).
    Builds nodes for identifiers, parenthesized identifier lists, or empty
tuples.
    """


def Vl() -> None:
    """
    Parses a Variable list (Vl).
    Builds ',' node for comma-separated identifiers.
```

```
    """
```

standardiser.py

```
def make_standardized_tree(root: Node) -> Node:
    """
    Transforms an AST into a ST by applying transformation rules.
    Parameters:
        root: The root Node of the AST.
    Returns:
        The root Node of the standardized tree.
    """
```

cse_machine.py

```
def generate_control_structure(root: Node, i: int) -> None:
    """
    Generates control structures (Lambda, Delta, Tau) for the CSE machine.
    Parameters:
        root: The root Node of the standardized tree.
        i: The index of the control structure list.
    """


def lookup(name: str) -> Any:
    """
    Resolves tokens starting and ending with '<' and '>' (e.g.,
identifiers, integers, strings).
    Parameters:
        name: The token string (e.g., '<ID:x>', '<INT:5>').
    Returns:
        The resolved value (e.g., integer, string, boolean, or environment
variable).
    Raises:
        KeyError: If an identifier is undeclared.
    """


def built_in(function: str, argument: Any) -> None:
    """
    Executes built-in functions (e.g., Order, Print, Conc, Stern, Stem,
Isinteger, etc.).
    Parameters:
        function: The name of the built-in function.
```

```
        argument: The argument to the function.
    """


def apply_rules() -> None:
    """
    Applies CSE machine rules to evaluate the control structures and
produce the result.
    Manages the control stack, environments, and stack operations.
    """


def get_result(st: Node) -> Any:
    """
    Processes the CSE machine execution to produce the final result.
    Parameters:
        st: The root Node of the standardized tree.
    Returns:
        The execution result (or empty string if no output).
    """
```

myrpal.py

```
def main() -> None:
    """
    Main entry point for the RPAL interpreter.
    Handles command-line arguments, reads input, and orchestrates
tokenization, parsing,
    standardization, and execution.
    """
```

Conclusion

The RPAL interpreter successfully implements the RPAL language using the provided grammar. This interpreter has a modular design that separates tokenization, parsing, standardization, and execution processes. The function prototypes discussed in this report illustrate the structure and responsibilities of each component. The program supports flexible output options via command-line arguments. The use of a stack-based parser and CSE machine ensures efficient processing. The standardization step simplifies execution by transforming complex constructs into a uniform format. Overall, this project helped us to understand the design and implementation of a functional programming language interpreter.