CS 3513: Programming Languages

Programming Languages Project

RATHNAYAKA R.L.M.P. | **220520P**

LAKRUWAN R. W. S | **220352C**

RPAL Interpreter Implementation

1. Overview

This project involves the development of an interpreter for the Right-reference Pedagogic Algorithmic Language (RPAL). The implementation is broken down into several core modules, each handling a specific part of the interpretation pipeline.

The interpreter consists of the following major components:

- A lexical analyzer
- A parser
- A standardizer
- A CSE (Control Structure Evaluation) machine

2. Development Details

2.1 Technology Stack

The interpreter was entirely developed using the Python programming language. No third-party packages or dependencies are required, ensuring straightforward execution on any Python-supported environment.

2.2 Lexical Analysis

Lexical analysis is carried out in compliance with the guidelines defined in the RPAL_Lex.pdf. This module scans the input source code and splits it into recognizable tokens, which are then passed on to the parser.

2.3 Parsing Mechanism

The parser takes the list of tokens from the lexical analyzer and generates an Abstract Syntax Tree (AST), adhering to the grammar provided in RPAL_Grammar.pdf. The parsing process is implemented from scratch — no external parser generators like lex or yacc are used.

2.4 Standardization Process

After constructing the AST, the next step is to convert it into a Standardized Tree (ST). This process reorganizes the AST into a simpler, uniform structure using lambda and gamma nodes, which are better suited for interpretation by the CSE machine.

2.5 CSE Machine Execution

The CSE machine interprets the standardized tree by navigating through its structure and performing the necessary control and computation tasks. It serves as the final execution layer of the RPAL interpreter.

2.6 Program Input and Output

The program reads an input file containing RPAL source code and supports the following Makefile targets:

run – Execute the RPAL interpreter with an input file

tokens – Print lexical tokens

ast – Print the Abstract Syntax Tree (AST)

sast – Print the Standardized AST (with lambdas and gammas)

cs – Display control structures for the CSE machine

3. Program Structure

3.1 Lexical Analyzer

The lexer (lexical analyzer) processes the source code (as a string) and produces a **sequence of tokens**, each representing a meaningful element like keywords, identifiers, operators, etc.

```python
class MyToken:⋯

def tokenize(input_str):
    tokens = []
    keywords = {
        'COMMENT': r'//.*',
        'KEYWORD': r'(let|in|fn|where|aug|or|not|gr|ge|ls|le|eq|ne|true|false|nil|dummy|within|and|rec)\b',
        'STRING': r'\'(?:\\\'|[^\'])*\'',
        'IDENTIFIER': r'[a-zA-Z][a-zA-Z0-9_]*',
        'INTEGER': r'\d+',
        'OPERATOR': r'[+\-*<>&.@/:=~|$\#!%^_\[\]{}"\'?]+',
        'SPACES': r'[ \t\n]+',
        'PUNCTUATION': r'[();,]'
    }

    while input_str:
        matched = False
        for key, pattern in keywords.items():
            match = re.match(pattern, input_str)
            if match:
                # print(key, match.group(0))
                if key != 'SPACES':
                    if key == 'COMMENT':
                        comment = match.group(0)
                        input_str = input_str[match.end():]
                        matched = True
                        break
                    else:
                        token_type = getattr(TokenType, key)  # Get TokenType enum value
                        if not isinstance(token_type, TokenType):
                            raise ValueError(f"Token type '{key}' is not a valid TokenType")
                        tokens.append(MyToken(token_type, match.group(0)))
                        input_str = input_str[match.end():]
                        matched = True
                        break
                input_str = input_str[match.end():]
                matched = True
                break
        if not matched:
            print("Error: Unable to tokenize input")
    return tokens

def show_tokens(tokens):⋯
```

Find the full function code [here]()

Function Description – Lexer Module

- ### Input:

  The `tokenize(input_str)` function takes a **string input**, which represents the complete source code of an RPAL program. This input includes keywords, identifiers, literals, operators, and other syntactic elements.

- ### Output:

  The function returns a **list of `MyToken` objects**, where each object contains:

  - `type`: an enumerated value from `TokenType` (e.g., `KEYWORD`, `IDENTIFIER`)
  - `value`: the exact string matched from the input (e.g., `"let"`, `"x"`, `"42"`)

- **Parameter Passing**:

  The input string is passed by value to the `tokenize` function. Internally, the string is processed sequentially, and matched segments are removed as tokens are generated.

- **Error Handling**:

  If a segment of the input does not match any known token patterns, an error message (`"Error: Unable to tokenize input"`) is printed. However, the function does not raise exceptions or terminate execution—it continues processing.

- **Return Values**:

  The function returns a complete list of valid tokens in the order they appear in the source code. This list can be used by other components like the parser for syntax analysis.

3.2 Parser

This parser is responsible for constructing an **Abstract Syntax Tree (AST)** from a sequence of **lexically analysed tokens** of the RPAL language. It follows a **recursive descent parsing** strategy based on the RPAL grammar rules, enabling structured interpretation of the source code into syntactic constructs.

```
    #  Bp   -> A ('gr' | '>' ) A => 'gr'
    #           -> A ('ge' | '>=') A => 'ge'
    #           -> A ('ls' | '<' ) A => 'ls'
    #           -> A ('le' | '<=') A => 'le'
    #           -> A 'eq' A => 'eq'
    #           -> A 'ne' A => 'ne'
    #           -> A ;


    def parse_boolean_primary(self):
        self.parse_arithmetic_expression()
        current_token = self.tokens[0]
        if current_token.value in [">", ">=", "<", "<=", "gr", "ge", "ls", "le", "eq", "ne"]:
            self.tokens.pop(0)
            self.parse_arithmetic_expression()
            if current_token.value == ">":
                self.syntax_tree.append(ASTNode(ASTNodeType.op_compare, "gr", 2))
            elif current_token.value == ">=":
                self.syntax_tree.append(ASTNode(ASTNodeType.op_compare, "ge", 2))
            elif current_token.value == "<":
                self.syntax_tree.append(ASTNode(ASTNodeType.op_compare, "ls", 2))
            elif current_token.value == "<=":
                self.syntax_tree.append(ASTNode(ASTNodeType.op_compare, "le", 2))
            else:
                self.syntax_tree.append(ASTNode(ASTNodeType.op_compare, current_token.value, 2))

```

Find the full function code [here]()

Function Description – Phaser Module

Input

- A **list of tokens**, each with type and lexeme, produced by the lexical analyzer.

Parsing Logic

- Implements a **recursive descent parser** based on RPAL grammar.
- Applies **top-down parsing** rules to recognize valid syntactic structures.
- Builds a **hierarchical AST** with internal nodes as non-terminals and leaves as terminal symbols (tokens).

Output / Return Values

- Returns the **root node of the Abstract Syntax Tree (AST)**.
- If parsing fails, returns **None**.

Parameter Passing

- Tokens are passed **by reference** to the `parse` method, allowing the parser to consume and manipulate the token stream during recursion.

Error Handling

- On encountering unexpected or invalid tokens:
  - Prints a **descriptive error message**.
  - May include **line/column information** if available.
  - **Terminates parsing early** by returning None.

Purpose

- Converts flat token sequence into a **structured representation of the program** (AST), enabling further stages like semantic analysis or code generation.

3.3 AST to ST Conversion (Standardizer)

Transforms the Abstract Syntax Tree (AST) into a Standardized Tree (ST) by restructuring and simplifying nodes.

```python
def standardize_tree(node):
    for i in range(len(node.children)):
        node.children[i] = standardize_tree(node.children[i])

    if node.value == "let":
        node.value = "gamma"
        P = node.children[1]
        E = node.children[0].children[1]
        node.children[1] = E
        node.children[0].children[1] = P
        node.children[0].value = "lambda"

    elif node.value == "where":
        P = node.children[0]
        where_child = node.children[1]
        if where_child.value == "=":
            X = where_child.children[0]
            E = where_child.children[1]
            lambda_node = TreeNode("lambda")
            lambda_node.add_child(X)
            lambda_node.add_child(P)
            node.value = "gamma"
            node.children = [lambda_node, E]

    elif node.value == "within":
        X1 = node.children[0].children[0]
        E1 = node.children[0].children[1]
        X2 = node.children[1].children[0]
        E2 = node.children[1].children[1]
        node.value = "="
        lamda = TreeNode("lambda")
        lamda.children = [X1, E2]
        gamma = TreeNode("gamma")
        gamma.children = [lamda, E1]
        node.children[0] = X2
        node.children[1] = gamma

```

Find the full function code here

Function Description

Input:

- Indented string representing a tree structure (tokens per line).

Output:

- Standardized Abstract Syntax Tree (`TreeNode`).

Functions:

- `build_tree`: Parses input string into a tree of `TreeNode` based on indentation.
- `standardize_tree`: Recursively transforms nodes (e.g., `let` → `lambda/gamma`), simplifying syntax.

Parameter Passing:

- Input string passed by value to `build_tree`.
- `TreeNode` passed by reference to `standardize_tree`.

Error Handling:

- None; assumes valid input.

Return:

- `build_tree` returns the root node of the raw tree.
- `standardize_tree` returns the root node of the standardized tree.

Purpose:

- Build and simplify the AST for further processing

3.4 CSE Machine

3.4.1 Control structure generator

Implements a Control Stack Environment (CSE) machine for executing functional programming language constructs through step-by-step evaluation of control expressions.

```python
    def _traverse(self, node):
        result = []

        # —— handle lambda ——
        if node.value == "lambda":
            delta_name = f"delta{self.delta_counter + 1}"
            self.delta_counter += 1

            # Check if first child is a comma node
            if len(node.children) >= 2 and node.children[0].value == ",":
                comma_node = node.children[0]
                body = node.children[1]

                # Get the children of the comma node
                if len(comma_node.children) >= 2:
                    T = comma_node.children[0].value
                    N = comma_node.children[1].value
                    lam_name = f"lambda5{T},{N}"
                else:
                    # Fallback if comma doesn't have enough children
                    lam_name = f"lambda{self.delta_counter}"
            else:
                # Original lambda handling
                var = node.children[0].value
                body = node.children[1]
                lam_name = f"lambda{self.delta_counter}{var}"

            # build the body-structure as its own delta
            body_struct = self._traverse(body)
            self.deltas[delta_name] = body_struct
            result.append(lam_name)

```

Find the full function code here

3.4.2 executes the program

```python
    # Truth value operations
    def _builtin_or(self, val1, val2):
        """Logical OR operation"""
        def is_truthy(val):
            if isinstance(val, bool):
                return val
            if isinstance(val, str):
                return val.lower() in ['true', '1', 'yes'] or (val.isdigit() and int(val) != 0)
            if isinstance(val, (int, float)):
                return val != 0
            return bool(val)

        result = is_truthy(val1) or is_truthy(val2)
        print(f"OR operation: {val1} or {val2} = {result}")
        return result

    def _builtin_not(self, value):
        """Logical NOT operation"""
        def is_truthy(val):
            if isinstance(val, bool):
                return val
            if isinstance(val, str):
                return val.lower() in ['true', '1', 'yes'] or (val.isdigit() and int(val) != 0)
            if isinstance(val, (int, float)):
                return val != 0
            return bool(val)

        result = not is_truthy(value)
        print(f"NOT operation: not {value} = {result}")
        return result

    def _builtin_ne(self, val1, val2):
        """Not equal comparison"""
        result = not self._builtin_eq(val1, val2)
        print(f"NE comparison: {val1} != {val2} = {result}")
        return result

```

Find the full function code [here]

Function Description

- **Input**: Takes initial control stack, evaluation stack, and environment mappings as starting state for program execution
- **Output**: Produces the final computed result value after complete program evaluation
- **Parameter Passing**:
    - **Control Stack**: Contains program instructions, operators, function applications, and control flow constructs (lambda expressions, conditionals, tuple operations) awaiting execution
    - **Evaluation Stack**: Stores intermediate computation results, function arguments, and operand values during expression evaluation
    - **Environment Mappings**: Maintains variable bindings, function definitions, and lexical scope information across different execution contexts

Core Functionality

- **Step-by-Step Execution**: Processes control elements sequentially using pattern matching and rule-based evaluation
- **Function Application**: Handles lambda function creation, parameter binding, and recursive function calls through Y-combinator support
- **Control Flow Management**: Implements conditional branching (beta rule) and structured programming constructs
- **Data Structure Operations**: Supports tuple creation, indexing, and manipulation with proper element ordering
- **Environment Management**: Creates nested scopes, variable bindings, and maintains lexical closure semantics

Return Values

The execution engine returns the final computed result after:

- Complete control stack exhaustion
- All intermediate calculations resolved
- Final value extracted from evaluation stack top
- Proper cleanup of temporary environments and bindings

Error Handling

- Type conversion and compatibility checking for operations
- Stack underflow protection during binary operations
- Environment lookup validation with fallback to literal interpretation

4. Usage

The RPAL interpreter can be executed in two ways:

1. Using Python Commands: Directly Open a terminal and navigate to the project directory.

   Use the following commands based on the required output:

   ```
   python3 myrpal.py input.txt           # Execute the program
   ```

   ```
   python3 myrpal.py input.txt -tokens   # Print Tokens
   ```

   ```
   python3 myrpal.py input.txt -ast      # Print the Abstract Syntax Tree
   ```

   ```
   python3 myrpal.py input.txt -sast     # Print the Standardized AST
   ```

```
python3 myrpal.py input.txt -cs.      # Print control structures
```

2. Using the Makefile: Open a terminal and navigate to the project directory.

   Use the corresponding make commands:

```
make run file=input.txt     # Execute the program

make tokens file=input.txt  # Print tokens

make ast file=input.txt     # Print the Abstract Syntax Tree

make sast file=input.txt    # Print the Standardized AST

make cs file=input.txt      # Print control structures
```

5. Conclusion

To summarize, we successfully developed a complete RPAL interpreter, consisting of a lexical analyzer, a parser, an AST-to-ST transformation module, and a CSE machine. The interpreter is capable of efficiently analyzing RPAL source code, generating abstract and standardized syntax trees, and executing programs accurately through the CSE evaluation process.

You can access the full project and source code on GitHub at the following link:

https://github.com/Sachintha-Lakruwan/RPAL-Compiler