CS3513: Programming Languages
Programming Languages Project Report

Group TechWhiz

220185P:Gunrathna D.M.N.S

220213D:Harinakshi W.B.S.S.

1. Project Description

This project implements an interpreter for RPAL (Right-reference Practical Algorithmic Language) in Python, fulfilling the course project requirements by developing the entire pipeline from source code to execution. The key stages are:

1. Lexical Analysis (Lexer) — tokenizes raw RPAL source code
2. Syntax Analysis (Parser) — builds an Abstract Syntax Tree (AST) from tokens
3. Standardization (Standardizer) — converts AST into a canonical Standardized Tree (ST)
4. Evaluation (CSE Machine) — executes the standardized program using a stack-based abstract machine

The interpreter supports command-line switches to output the AST (-ast) and the Standardized Tree (-st), aiding debugging and correctness verification against the official RPAL implementation (rpal.exe). The design focuses on modularity, clarity, and adherence to RPAL semantics.

2. Components Overview

2.1 Lexical Analyzer (Lexer)

Purpose:
Transforms raw RPAL source text into a sequence of tokens, each representing the smallest meaningful unit such as identifiers, integers, operators, and end of line.

Implementation Details (from lexer.py):

Token class: Represents tokens with type (e.g., IDENTIFIER, INTEGER) and value.
Lexer class: Reads the input source and generates tokens.
* advance() and peek() methods navigate characters.
*skip_whitespace() and skip_comment() clean input.

Token-specific methods:
get_identifier(): Recognizes identifiers and keywords.
get_number(): Extracts integer literals.
get_operator(): Handles RPAL operators (e.g., +, *, ->).
get_punction(): Parses punctuation like (,), ;, ,.
get_string(): Processes string literals, including escape sequences (\n, \t, etc.).
get_next_token(): Main driver returning the next token.
tokenize(): Produces the complete token list until EOF.

Key Features:

*Skips comments starting with //.

*Supports escape sequences in strings.
*Properly differentiates between identifiers and reserved keywords.
*Generates tokens compatible with RPAL syntax rules.

Example Token Output:

[<IDENTIFIER:let>, <IDENTIFIER:x>, <OPERATOR:=>, <INTEGER:10>]

The token stream feeds directly into the parser.

2.2 Parser

Purpose:
Converts the token stream into an Abstract Syntax Tree (AST), representing the syntactic structure of the RPAL program in a hierarchical tree form.

Design (Assumed / Typical):

*Uses recursive descent parsing techniques based on the RPAL grammar.
*Parses expressions, function definitions, applications, conditionals, and operators based on RPAL grammar.
*Constructs tree nodes with labels and children corresponding to syntactic constructs such as lambda, gamma (application), let, -> (conditional), etc.
*Ensures syntax correctness and meaningful error reporting.

Parser Function Prototypes

The parser.py module implements a recursive descent parser for RPAL. Key function prototypes include:

Grammar-based recursive descent functions

```
def E(): pass       # Expression
def Ew(): pass      # Where-clause
def T(): pass       # Tuple
def Ta(): pass
def Tc(): pass
def B(): pass       # Boolean
def Bt(): pass
def Bs(): pass
def Bp(): pass
def A(): pass       # Application
def At(): pass
def Af(): pass
def Ap(): pass
def R(): pass       # Recursion
def Rn():pass

Declaration-related functions:
def D(): pass
def Dr(): pass
```

```
def Db(): pass
def Vb():pass
def Vl(): pass

Entry point :
def parse(): pass
```

Output:
A tree structure where each node represents a language construct or token, which can be printed or traversed for further processing.

Usage:

```
make ast file=path/to/your/input.txt
```

To print the Abstract Syntax Tree (AST) with the output, use the ast target.

2.3 Standardizer

Purpose:
Transforms the AST into a Standardized Tree (ST) by applying RPAL-specific standardization rules that simplify and canonicalize program structure. This eases later evaluation and guarantees uniform representation

Typical Transformations:

*Converts let x = E1 in E2 constructs into lambda applications: gamma(lambda x. E2) E1.
* Converts infix operators into prefix form.
*Normalizes recursion and conditional syntax.
*Flattens or restructures certain nodes to fit the evaluation model.

Standardizer Function Prototypes:

The standadizer.py module defines the following key components:

```
class Node:
    def __init__(self, value: str, children: Optional[List['Node']] = None) -> None
```

Constructs a tree node used by the standardizer with a value and optional list of children.

```
def convert_tree_to_node(tree_node) -> Node
```

Converts a parser-generated tree node (TreeNode or LeafNode) into the internal Node structure for standardization.

```
def standardize(tree_root) -> Node
```

Applies structural transformations on the input AST node recursively to generate a standardized abstract syntax tree.

```
def print_tree(node: Node, dots: int = 0) -> None
```

Prints the standardized tree structure in a human-readable form for debugging or visualization purposes.

```
def parse_and_standardize(source_tokens: List[str]) -> Node
```

Orchestrates the parsing of source tokens and standardizes the resulting AST, returning the root of the standardized tree.

Output

A transformed and standardized version of the input program's abstract syntax tree (AST), conforming to the expected structure of the CSE machine.

Usage:

```
make st file=path/to/your/input.txt
```

To print the standardized Abstract Syntax Tree (ST) with the output, use the st target.

2.4 CSE Machine (Control Structure Evaluator)

Purpose:
Evaluates the standardized RPAL program via an abstract machine approach, simulating the operational semantics of the language.

Key Features:

Data Structures:

- Control List: Instructions or symbols to be evaluated.
- Stack: Temporarily stores intermediate values and environments.
- Environment Stack: Maps variable identifiers to their bound values.

Symbolic Representation:
Uses classes representing RPAL constructs (Symbol, Rand, Rator, Lambda, Delta, E, Beta, etc.) implemented in nodes.py.

cse_factory.py:
Converts the standardized tree nodes into machine-understandable symbols and manages indexing and environment setup.

Evaluation Mechanics:
The machine repeatedly applies operational rules to the control list and stack:
- Function application (gamma symbol) triggers environment changes.
- Lambda abstraction (lambda) captures closures.
- Variables are looked up in environment frames.
- Control constructs like beta and tau manage conditional branching and tuple processing.

Execution:
Processes input until the control list is exhausted, yielding the final result of the RPAL program.

Overall Flow:
AST → Standardized Tree → Factory-generated symbols → CSE Machine evaluation → Output.

Function prototypes:

```
apply_rules(control_list, stack_list, environment_list)
```

Main evaluator function. Iteratively applies the 13 CSE machine rules to evaluate the program based on control, stack, and environment lists.

```
built_in(function, argument, stack_symbol_2, stack)
```

Handles built-in operations like arithmetic (+, -), logical (and, or), comparison (eq, gr), and tuple operations (aug, etc.).

```
generate_control_structure(root, count)
```

Recursively traverses the ST to generate control structures, including lambda expressions and delta closures.

```
add_delta(delta_num, control_structure)
```

Adds a generated delta (sub-control structure) to the global list of deltas with a unique number for reference.

class Stack

Custom stack implementation with enhanced indexing and inspection support for managing the machine stack.

```
class Stack:
    def __init__(self):
        # Initialize an empty list to represent the stack
        self.items = []

    def push(self, item):
        # Push an item onto the top of the stack
        self.items.append(item)

    def pop(self):
        # Pop and return the top item from the stack
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("pop from empty stack")

    def peek(self):
        # Return the top item without removing it
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError("peek from empty stack")

    def is_empty(self):
        # Check if the stack is empty
        return len(self.items) == 0

    def __getitem__(self, index):
        # Enable access to stack elements via indexing (e.g., stack[0])
        return self.items[index]

    def __setitem__(self, index, value):
        # Allow setting a stack element at a specific index
        self.items[index] = value

    def __len__(self):
        # Return the number of items in the stack
        return len(self.items)
```

These are the control structures used in structures.py.

```
class Delta:
    def __init__(self, number):
        # Delta represents a control structure index (typically for function or block)
        self.number = number


class Tau:
    def __init__(self, number):
```

```
        # Tau is used to represent a tuple or structured expression with a given arity
(number of elements)
        self.number = number


class Lambda:
    def __init__(self, number):
        # Lambda represents a function abstraction with an identifier number
        self.number = number

        # bounded_variable refers to the variable bound by this lambda (e.g., in lambda x.
E, x is the bounded variable)
        self.bounded_variable = None

        # environment captures the environment in which this lambda was defined (used for
closures)
        self.environment = None


class Eta:
    def __init__(self, number):
        # Eta is used for eta-reduction or transformation, similar to Lambda
        self.number = number

        # bounded_variable refers to the variable bound in eta abstraction (like lambda)
        self.bounded_variable = None

        # environment stores the environment context for the eta abstraction
        self.environment = None
```

3. Makefile for Build and Execution Automation

To facilitate easy and consistent execution of the RPAL interpreter and its various components, a Makefile is provided. This file automates common tasks such as running the interpreter, generating lexer output, producing the Abstract Syntax Tree (AST), displaying the symbol table, and cleaning up generated files.

Makefile Contents:

```
PYTHON = python3
APP = myrpal.py

run:
	$(PYTHON) $(APP) $(file)

lexer:
	$(PYTHON) $(APP) -l $(file)

ast:
	$(PYTHON) $(APP) -ast $(file)

st:
	$(PYTHON) $(APP) -st $(file)
```

```
clean:
	rm -rf __pycache__ *.pyc

.PHONY: run lexer ast st clean
```

Description of Targets:

- run: Executes the interpreter on the specified input file.
- lexer: Runs the lexical analysis phase to output tokens.
- ast: Generates and displays the Abstract Syntax Tree for the input program.
- st: Produces the symbol table, useful for semantic analysis.
- clean: Removes Python cache files and directories to keep the workspace clean.

Usage:

To use the Makefile, specify the input source file with the file variable. For example, to run the interpreter on example.rpal, use:

```
make run file=example.rpal
```

Other commands follow the same pattern:

```
make lexer file=example.rpal
make ast file=example.rpal
make st file=example.rpal
make clean
```

This automation enhances developer productivity by reducing manual command entry and maintaining consistency in running tests and builds.

4. Summary

This project faithfully implements the RPAL language interpreter by constructing a robust pipeline from source code to final output:

*The Lexer cleanly tokenizes input, handling all RPAL syntax and literals.
* The Parser builds a syntactic AST reflecting the program structure.
*The Standardizer prepares a canonical, evaluation-ready tree.
*The CSE Machine uses symbolic evaluation techniques to execute RPAL programs correctly and efficiently.

This modular design enables independent development and testing of components while ensuring conformance to RPAL semantics and the official interpreter output.

5. Conclusion

In conclusion, we successfully implemented a lexical analyzer, parser, AST to ST conversion algorithm, and CSE machine for the RPAL language. Our program can efficiently parse RPAL

programs, construct ASTs, execute them using the CSE machine, and produce accurate results.

Please find the GitHub repository link below for the project:

https://github.com/NipunSachintha/RPAL-interpreter

References:

https://github.com/neerajrao/rpal-interpreter.git