# CS 3513: Programming Languages

**RPAL Parser and Interpreter**

**Group: Null Pointers**

**Members: Karunaratne T.K.M.C – 220309D**

**Navodi S.Y.A.C – 220419N**

**GitHu Repo Link:https://github.com/mickymarsh/RPAL-parser**

---

## 1. Introduction

This project implements an interpreter for the Right-reference Pedagogic Algorithmic Language (RPAL) in Python. The interpreter processes RPAL source code through the following phases:

1. **Lexical Analysis**
2. **Recursive Descent Parsing (AST Generation)**
3. **Standardizing the AST**
4. **Evaluation using a CSE (Control Stack Evaluation) Machine**

The goal is to evaluate RPAL programs using pure functional principles and lambda calculus concepts, without using external parser generators or interpreters.

## 2. Technologies Used

- **Language**: Python 3
- **Tools**: Custom-built components (no third-party libraries)
- **Execution Environment**: Command-line interface
- **Input Format**: .rpal files with RPAL syntax

## 3. Component Breakdown

### 3.1 Lexical Analyzer (scanner.py)

- Implements tokenization manually.
- Converts input strings into a sequence of tokens (Token objects).
- Handles keywords, identifiers, symbols, numbers, and comments.
- Uses regular expressions for token definitions.

**Output**: List of tokens
**Errors**: Invalid characters raise descriptive SyntaxError.

### 3.2 Parser (parser.py)

- A recursive descent parser that constructs an **Abstract Syntax Tree (AST)**.
- Follows RPAL grammar precisely without using external tools like yacc or ply.
- Each node in the tree is an instance of the Node class, storing data and children.

**Key Functions**:

- parse_E, parse_T, etc.: Each corresponds to a grammar rule.
- print_tree: Used to display the AST.

### 3.3 Standardizer (standardizer.py)

- Transforms the AST into a **Standardized Tree (ST)** using transformation rules.
- Introduces explicit lambda (lambda) and application (gamma) constructs.
- Simplifies various syntactic constructs into a core functional subset.

**Purpose**: Converts syntactic sugar into a minimal core language for evaluation.

### 3.4 CSE Machine (cse_machine.py)

- Evaluates the Standardized Tree using:
    - **Control Stack**: Instructions and function calls.
    - **Value Stack**: Intermediate results and values.

- **Environment Stack**: Variable bindings and closures.

**Key Concepts**:

- Lazy evaluation
- Closures and nested scopes
- Higher-order functions

**Entry Point**: CSEMachine.evaluate(tree)

# 4. Execution Instructions

### 4.1 Using direct python command

- Open the terminal and navigate to the directory where the interpreter is located.Use below command to execution.

  python main.py

### 4.2 Using the Makefile

The repository provides a simple Makefile to automate execution.

```
# Run the main Python program.This is the default target
run:

    python myrpal.py $(file)

#When the target is to generate AST (Abstract Syntax Tree) from the input file
ast:

    python myrpal.py  $(file) -ast

# When the target is to generate a Standardized tree from the input file
st:

    python myrpal.py  $(file) -st

# Lint the code using flake8 to find style and syntax issues
```

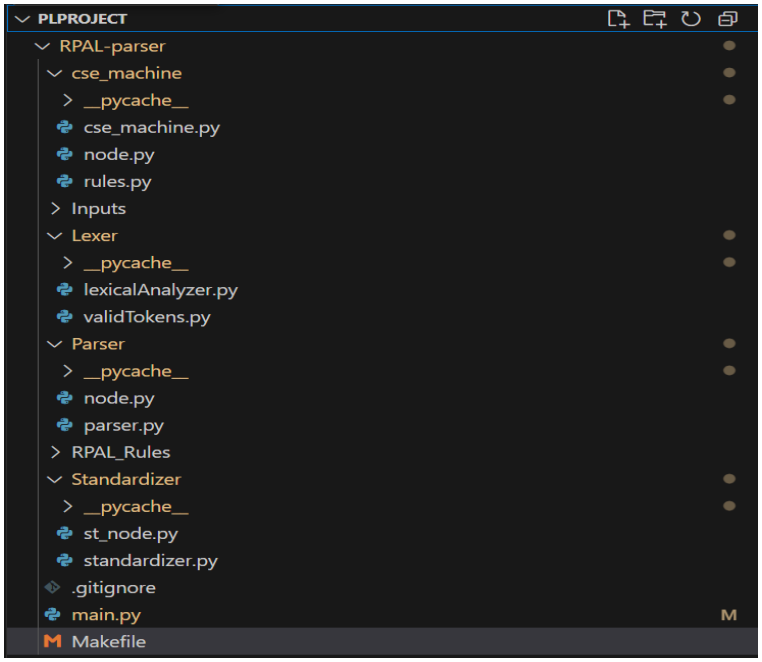
As above picture when user give

1. python myrpal.py (their file path) ➔ It will provide the ouput of the program
2. python myrpal.py (their file path) - ast ➔ It will provide both ast and the ouput of the program
3. python myrpal.py (their file path) - st ➔ It will provide both st and the ouput of the program

And here are more instructions to run

```
8
9   # Lint the code using flake8 to find style and syntax issues
10  lint:
11
12      flake8 Lexer/ Parser/ Standardizer/ main.py
13
14  # Clean up Python bytecode files and __pycache__ directories
15  clean:
16
17      find . -name "*.pyc" -delete
18      find . -name "__pycache__" -type d -exec rm -r {} +
19
20  # Run all unittests located in the 'tests' directory
21  test:
22      python -m unittest discover -s tests
23
```

## 5. Folder Structure




- **main.py**
  The main entry point of the program. It handles input reading and coordinates the full pipeline (lexing → parsing → standardizing → evaluation).

- **Makefile**
  Contains commands to compile/run the program easily using make.

- **.gitignore**
  Standard Git file to exclude __pycache__ and other unnecessary files from version control.

---

**Inputs/**

- Directory where test .txt input programs are stored. You run these as examples (e.g., Inputs/Q01.txt).

---

**Lexer/**

Handles **lexical analysis (tokenization)**.

- **lexicalAnalyzer.py**: Contains the tokenize() logic that breaks RPAL source code into tokens.

- **validTokens.py**: Defines token types or reserved words (keywords, symbols, etc.).

---

**Parser/**

Handles **syntax analysis** and AST creation.

- **parser.py**: Implements the recursive descent parser based on RPAL grammar. Builds the AST.

- **node.py**: Defines the Node class used in the AST.

---

**Standardizer/**

Converts the AST to a **Standard Tree (ST)**.

- **standardizer.py**: Applies transformation rules (e.g., converts let, where, and operators into lambdas/gammas).

- **st_node.py**: Defines specialized node structures for standardized tree.

**cse_machine/**

Executes the program using a **Control Stack Evaluation (CSE) machine**.

- **cse_machine.py**: Main logic for executing the Standardized Tree (ST).
- **rules.py**: Contains built-in rules (e.g., for arithmetic, logic).
- **node.py**: Likely reused or extended node definitions for evaluation.

---

**RPAL_Rules/**

Likely stores transformation or grammar rules specific to RPAL. Could contain helper definitions or future extensions.

---

This modular structure:

- Keeps each phase (Lexing, Parsing, Standardizing, Execution) cleanly separated
- Makes the interpreter easy to debug and extend
- Encourages reusability of Node structures and transformation logic

---

# 6. Output

We ran following program to get the outputs.

```
let Sum(A) = Psum (A,Order A )
                  where rec Psum (T,N) = N eq 0 -> 0
                                       | Psum(T,N-1)+T N
in Print ( Sum (1,2,3,4,5) )
```

Here are the outputs we got from lexical analyser, parser and cse machine

-Abstract Syntax Tree-

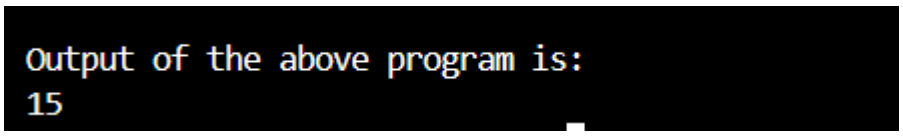
```
let
.function_form
..<ID:Sum>
..<ID:A>
..where
...gamma
....<ID:Psum>
....tau
.....<ID:A>
.....gamma
......<ID:Order>
......<ID:A>
...rec
....function_form
.....<ID:Psum>
.....,
......<ID:T>
......<ID:N>
.....->
......eq
.......<ID:N>
.......<INT:0>
......<INT:0>
......+
.......gamma
........<ID:Psum>
........tau
.........<ID:T>
.........-
..........<ID:N>
..........<INT:1>
.......gamma
........<ID:T>
........<ID:N>
.gamma
..<ID:Print>
..gamma
...<ID:Sum>
...tau
....<INT:1>
....<INT:2>
....<INT:3>
....<INT:4>
....<INT:5>
```

-Standard Tree-

```
gamma
.lambda
..<ID:Sum>
..gamma
...<ID:Print>
...gamma
....<ID:Sum>
....tau
.....<INT:1>
.....<INT:2>
.....<INT:3>
.....<INT:4>
.....<INT:5>
.lambda
..<ID:A>
..gamma
...lambda
....<ID:Psum>
....gamma
.....<ID:Psum>
.....tau
......<ID:A>
......gamma
.......<ID:Order>
.......<ID:A>
...gamma
....ystar
....lambda
.....<ID:Psum>
.....lambda
......comma
.......<ID:T>
.......<ID:N>
......cond
.......eq
........<ID:N>
........<INT:0>
.......<INT:0>
.......plus
........gamma
.........<ID:Psum>
.........tau
..........<ID:T>
..........minus
...........<ID:N>
...........<INT:1>
```

```
.....gamma
.....<ID:T>
.....<ID:N>
```

And the Final output of the program is…




# 7. Conclusion

In conclusion we implemented a lexical analyzer,parser ,standardizer to convert AST to ST and evaluation mechanism with cse machine for the RPAL language.With this project we get insight how interpreter work and internal functionalities.Finally our program using above mentioned tools and structure evaluate and produce accurate results for input RPALl codes.

**Repository**: https://github.com/mickymarsh/RPAL-parser