# CS 3513 – Programming Languages

## RPAL Interpreter Project Report

# Group Number: 43


## Group Members:

**JAYATHILAKE N.C. 210257F**

**KARUNATHILAKA G.M.D.S. 210277P**

## Problem Description

The project requirement was to implement a lexical analyzer and a parser for the RPAL language. The output of the parser should be the Abstract Syntax Tree (AST) for the given input program. Then an algorithm must be implemented to convert the Abstract Syntax Tree (AST) into a standardized tree (ST) and the CSE machine should be implemented. The program should be able to read an input file that contains an RPAL program and switch between with -ast and without -ast.

## Solution

| | |
|---|---|
| Lexical Analyzer | There are two stages included st; Screener and Scanner. Commence by putting in place a basic tokenization procedure. To recognise fundamental tokens such as operators, literals, keywords, and identifiers, define regular expressions. Keep track of each token's type and value. |
| Parsing | Next, concentrate on a portion of RPAL's grammar that consists of fundamental function definitions and simple expressions. Use a recursive descent parser to combine the tokens into an Abstract Syntax Tree (AST). |
| AST evaluation | Create an interpreter that evaluates expressions while navigating the AST. Begin with basic function calls and arithmetic expressions. |
| Standardised tree | Create transformation rules to simplify the AST and get rid of components like let-bindings and lambda functions if you wish to go on to the ST. |
| Exception Handling | If the input programme does not match, this programme should identify it and display an error notice. |

## Structure of the Project:

This project was coded entirely in Python. It consists of mainly 6 files. They are

1) myrpal.py
2) LexicalAnalyzer.py
3) parser.py
4) Standardizer.py
5) environment.py
6) cse_machine.py

This document outlines the purpose of each file and presents the function prototypes along with their uses.

## 1. myrpal.py

The RPAL interpreter's primary entry point is `myrpal.py. It does this by coordinating the interactions between the lexical analyzer, parser, standardizer, and control stack machine the several parts of the interpreter that make up the execution flow.

The steps followed by the interpreter as follows.

- This imports necessary modules such as ***RPALParser*** from `parser.py`, ***Standardizer*** from `Standardizer.py`, and ***CSE_machine*** from `cse_machine.py`. These modules encapsulate functionalities related to parsing RPAL code, standardizing the parsed code, and executing it using a control stack machine.
- Command Line Arguments handling- ***myrpal.py*** reads input arguments from the command line to determine the mode of operation and the RPAL source file to be processed.
- Parser Instantiation- It instantiates an RPAL parser (`RPALParser`) to parse the RPAL source code provided.

- Standardizer Instantiation - An instance of the `Standardizer` class is created to standardize the parsed RPAL code. Standardization involves transforming the parsed code into a normalized form suitable for execution.
- Control Stack Machine Instantiation - Then `myrpal.py` initializes a control stack machine `CSE_machine` to execute the standardized RPAL code.
- Depending on the command line arguments and the number of arguments provided, `myrpal.py` determines the appropriate action. It can print the abstract syntax tree (`-ast` flag), the standardized output (`-st` flag), or execute the RPAL code directly if instructed.
- Here we do error handling to manage invalid command line inputs and provide appropriate feedback to the user.

Overall, `myrpal.py` encapsulates the core functionalities required for interpreting RPAL code, offering a seamless interface for users to interact with the RPAL interpreter.

## 2. LexicalAnalyzer.py

**Token class -** The 'Token' class is designed to encapsulate the attributes of each token in the program file.Each token has its type and a value.

```
class Token:
    """
    This is for each token in the program file.
    """
    def __init__(self, type, value):
        # each token has it's type and a vlaue
        self.type = type
        self.value = value
```

**Tokenizer class -** The Lexical Analyzer, implemented in the `Tokenizer` class, plays a fundamental role in the RPAL interpreter by converting input RPAL source code into a sequence of tokens.

**reset() -** This returns the finite automaton to its initial state and retrieves the tokens, provided that they are deemed acceptable.

**trans_at_0() -** There are several transition functions upto **trans_at_22()** which depicts the transitions between two states of the finite automata.

**scanner() -** This screener function processes a list of tokens by removing unnecessary whitespace, tabs, end-of-line characters, and comments. It also converts certain tokens into keywords. After filtering out the unwanted tokens, it updates the list with the remaining tokens.

**tokenize() -** This is the overall function that,reads an input file and breaks it down into a list of tokens, identifying each token's type and value based on predefined rules.

The below are the tokenizing rules used,

- **Identifiers:** Sequences of alphanumeric characters and underscores, representing variable names, function names, or keywords.
- **Integers:** Contiguous sequences of digits.
- **Operators:** Single or multi-character operators such as arithmetic operators (+, -, *, /), comparison operators (<, >), logical operators (&, |), assignment operator (=), and others.
- **Strings:** Enclosed in single quotes ('), allowing for escape sequences (\t for tab, \n for newline, etc.).
- **Special Characters:** Parentheses (,), commas (,), and semicolons (;) are treated as individual tokens.

This iterates through the input text, extracting tokens sequentially until the entire text is processed. It employs methods such as `***get_identifier()***`, `***get_integer()***`, `***get_operator()***`, and `***get_string()***` to identify and extract tokens based on their respective types.

And for the unexpected characters gives an error.

Then the extracted tokens are collected and returned as a list, ready for further processing by subsequent components of the interpreter, such as the parser.

## 3. parser.py

- **RPALParser class -** This class is responsible for parsing RPAL source code and constructing an Abstract Syntax Tree (AST) representing the structure of the program. It utilizes a recursive descent parsing technique, breaking down the input code into smaller components according to the RPAL grammar rules.
- **token _extraction()** - This function uses a `Tokenizer` to extract tokens from the input source code, categorizing them into identifiers, integers, strings, operators, etc.
- **read_token() -** This function consumes the current token, if it matches with the required token's value otherwise throws exception saying syntax error occurred.
- **The read_token_by_type()** - This function consumes the current token, if it matches with the required token's type otherwise throws exception saying syntax error occurred.
- **build_tree()** - This function pops the nodes from the stack, makes them as children of a node and pushes parent node back to the stack.

```
def build_tree(self, value, n):
    """
    This function build pops the nodes from the stack, makes them the child of the parent
    , and pushes the parent back to stack.

    """
    parent = TreeNode(value)
    for i in range(n):
        parent.add_child(self.stack.pop())
    self.stack.append(parent)
    return
```

- **Proc_E() -** The parser implements parsing procedures for each non-terminal symbol in the RPAL grammar. There are sevaral parsing procedures Proc_E(), Proc_Ew(),..... ,Proc_vb(). These procedures recursively analyze the input tokens, adhering to the syntax rules defined for RPAL. Each represent a single grammar rule of RPAL grammar and helps the parsing process.
- **parse_file()** - This method serves as the entry point for parsing RPAL code from a file. It extracts tokens from the input file and initiates the parsing process by invoking the appropriate parsing procedures.
- **build_AST()** - This method recursively constructs the AST from the parsed tokens, organizing them into a hierarchical tree structure. The

resulting AST provides a structured representation of the input RPAL code, facilitating subsequent analysis and interpretation.

```python
def build_AST(self, node, level=0):
    """
    This function prints the built AST tree.

    """
    # level is for indicating the tree levels of each node
    self.output_AST += '.' * level + node.value + "\n"
    if len(node.children) == 0:
        return

    # when building AST children of each node is reversed.
    # So we make them reveresed again to get the correct version
    node.children.reverse()

    level += 1

    for child in node.children:
        self.build_AST(child, level)
```

- **Error Handling** - The parser includes error-handling mechanisms to detect syntax errors during the parsing process. It raises exceptions when encountering unexpected tokens or violating syntax rules, providing informative error messages to aid debugging.

The parser generates an AST representation of the input RPAL code, which can be printed or saved to a file for visualization or subsequent analysis.

```python
class TreeNode:
    """
    This class represents the AST Nodes. Each node has Parent, Children, and Value.
    """
    def __init__(self, value):
        self.parent = None
        self.children = []
        self.value = value

    def add_child(self, child):
        """
        This function is for adding children to an AST Node.
        :param child:
        :return:
        """
        child.parent = self
        self.children.append(child)

    @staticmethod
    def remove_child(child):
        child.parent.children.remove(child)
        child.parent = None
```

The above **‘TreeNode’** class is designed to represent nodes in an AST. Each node has Parent, Children and Value.

- The constructor initializes a node with a given value.
- **add_child() -** This method adds a child node to the current node.

- **remove_child()** - This static method removes a ‘child’ node from its parent.

## 4 . Standardizer.py

- **Standardizer  class** - This class provides the functionality to standardize an AST (Abstract Syntax Tree) the RPAL parser produces. Standardization involves transforming the AST into a canonical form by applying predefined rules. Here's an overview of the key components and functionalities of the `Standardizer` class:
- **Initialization** -  The class initializes with attributes `std_tree` to hold the standardized tree and `output_ST` to store the standardized tree for output.
- **Standardization Functions** - The class defines several methods, each responsible for standardizing a specific type of AST node. The below are the functions used ,
    - `*std_let*` - Standardizes a "let" node by renaming it to "gamma", swapping "P" and "E", and applying other transformations.
    - `*std_where*` - Standardizes a "where" node by renaming it to "gamma", swapping "=", renaming it to "lambda", and reordering children.
    - ‘*std_fcn_form*`-  Standardizes a "function_form" node by restructuring it into a sequence of lambda nodes.
    - `*std_multi_param_fn*` - Standardizes a lambda node representing a function with multiple parameters.
    - `*std_within*` - Standardizes a "within" node by restructuring it into a sequence of lambda and gamma nodes.
    - `*std_at_sign*` - Standardizes an "@" node by reordering children to conform to standard form.
    - `*std_and*` - Standardizes an "and" node representing simultaneous definitions.
    - `*std_rec*` -  Standardizes a "rec" node by transforming it into a lambda node with a Y* combinator.

- **Standardization_nodes()** - This method recursively traverses the AST and applies the appropriate standardization function to each node based on its type and structure.
- **build_ST()** - This method constructs a standardized tree from the AST and stores it in the `output_ST` attribute.
- **standardize()** - This method is the entry point for standardizing an AST. It takes the AST as input, applies standardization functions, and generates the standardized tree.

## 5 . environment.py

**Environment()** - This represents an environment for storing variable bindings within a programming language interpreter or compiler.

```
class Environment:
    def __init__(self, name):
        self.name = name
        self.binding = dict()
        self.parent = None
        self.children = []
```

**add_child() -** This  method allows adding a child environment to the current environment. It sets the parent of the child environment and appends it to the list of children.

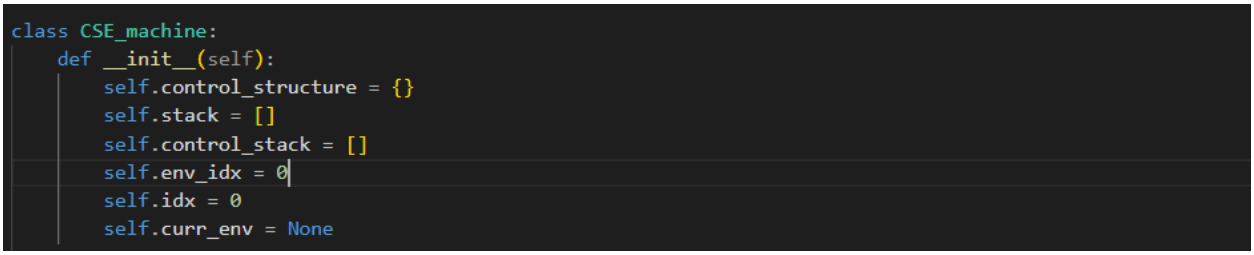
**lookup()** - This method is used to look up a variable (`var`) in the current environment's bindings. If the variable is not found in the current environment, it recursively searches in the parent environment. If the variable is not found in any ancestor environment, it exits the program with an error message indicating that the variable is not defined.

**add_binding()-**  This method adds a variable (`var`) and its corresponding value (`value`) to the environment's bindings dictionary.

Overall, this class provides a convenient way to manage variable bindings within a hierarchical environment structure, allowing for efficient variable lookup and scoping mechanisms.

## 6 . cse_machine.py

**CSE_machine()-** This class is where the made ST is being evaluated. This represents a Control Stack Machine, which is used to execute programs by interpreting control structures derived from a syntax tree (ST).



```
class CSE_machine:
    def __init__(self):
        self.control_structure = {}
        self.stack = []
        self.control_stack = []
        self.env_idx = 0
        self.idx = 0
        self.curr_env = None
```

*control_structure* -to hold control structures derived from the syntax tree
*stack* - to hold operands and intermediate results
*control_stack* - to manage control flow during execution
*env_idx* - to generate unique environment IDs
*idx* - to discriminate new control structures
*curr_env* -to keep track of the current environment

**label_lambda()** -This method labels lambda nodes in the syntax tree, assigning each lambda node a unique index to distinguish them.

**generate_control_structure()** - This method traverses the syntax tree and generates respective control structures based on the encountered nodes. It populates the `control_structure` dictionary with control structures indexed by their respective IDs.

**apply()-**This implements the application rule, which applies operations or functions to operands on the stack based on the operator or function at the top of the control stack.

**run_program()** - This executes the program by continuously applying rules until the control stack is empty. It follows the rules specified for control stack machine execution, including applying rules for operators, identifiers, lambdas, conditionals, and tuples.

**execute() -** This method serves as the entry point for executing a program. It labels lambda nodes, generates control structures, and then runs the program using the generated control structures.

Overall, the `CSE_machine` class provides a framework for executing programs using control stack machine semantics, which involves manipulating control structures and operands based on rules defined for different types of nodes in the syntax tree.

## Output of the overall Program for the given sample input:

```
Starting the CSE Machine...

Output of the above program is:
15
```

## Output of the overall Program for the given sample input- (switched to -ast)

```
Starting the CSE Machine...

The corresponding AST:

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>
```