# CS3515 Programming Languages
# RPAL Interpreter Analysis Project Report

**by**

**Group Mmbu Roomies**

**210017V Abineyan R.  210554M Sajeev K.**

## 1. Problem Description

You are required to implement a lexical analyzer and a parser for the RPAL language. Refer RPAL_Lex.pdf for the lexical rules and RPAL_Grammar.pdf for the grammar details. You should not use ‘lex’. ‘yacc’ or any such tool. Output of the parser should be the Abstract Syntax Tree (AST) for the given input program. Then you need to implement an algorithm to convert the Abstract Syntax Tree (AST) into Standardized Tree (ST) and implement CSE machine. Your program should be able to read an input file which contains a RPAL program. Output of your program should match the output of “rpal.exe“ for the relevant program.

## 2. Program Components

**Tokenizer**

The Tokenizer serves as a lexical analyzer using a hard-coded scanner. It screens the input file and generates tokens, merging the necessary ones. It identifies reserved keywords, removes comments and white spaces, and ultimately returns an array of tokens.

**AST Parser**

The AST parser processes the token sequence, constructing an Abstract Syntax Tree (AST) using recursive descent parsing. This method ensures that the AST accurately represents the structure of the source code.

**Standardizer**

The Standardizer performs a pre-order traversal of the AST, standardizing each node to ensure consistency and correctness in the tree's representation.

**Control Structure Generation**

During Control Structure Generation, a pre-order traversal of the AST is performed. A First-In-First-Out (FIFO) queue is maintained to facilitate the generation of control structures from the nodes of the AST.

**Control Structure Environment Evaluation**

In the Control Structure Environment Evaluation phase, a control structure array and a stack are maintained. Each element in the control structure array is popped and a rule is executed based on the top of the stack and the current environment. This step ensures that the control structures operate correctly within the given context.

## 3. Program Process

### Tokenization Process:

1. Initialization in myrpal.py:

- The tokenization process starts in the `myrpal.py` file.
- The program collects command line arguments, initializes a token list, and repeatedly calls the `get_next_token()` method from `LexicalAnalyzer.py`, adding each token to the list.

2. Tokenization in LexicalAnalyzer.py:

I. Scanner:

- The scanner breaks down the RPAL program text into tokens using a hard-coded lexical analyzer, classifying each token by type (e.g., keyword, identifier).
- The `TokenType` enum is utilized to identify each token's type.

II. Screening:

- The `Screener` class in `LexicalAnalyzer.py` is responsible for screening the tokens.
- The `merge_tokens()` method combines specific tokens such as `**`, `->`, `=>` and `=<`.
- The `remove_comments()` method removes comments from the code.
- The `screen_reserved_keywords()` method identifies reserved keywords.

## Parsing and AST Generation:

Files: myrpal.py

Implements a recursive descent parsing methodology to create an AST from the identified tokens. The AST Parser class handles the parsing process, starting by invoking the `procE()` method. The ASTParser object includes various methods that are components of the recursive descent parser: `procE`, `procEw`, `procT`, `procTa`, `procTc`, `procB`, `procBt`, `procBs`, `procBp`, `procA`, `procAt`, `procAf`, `procAp`, `procR`, `procRn`, `procD`, `procDa`, `procDr`, `procDb`, `procVb`, and `procVL`.

## AST to Standardized Tree Transformation:

File: AbstractSyntaxTreeNode.py

The `standardize()` method in the AbstractSyntaxTreeNode class is responsible for standardizing the AST. Using the RPAL Subtree Transformational Grammar, the method standardizes the following nodes: `let`, `where`, `within`, `function_form`, `and`, `rec`, and `@`.

## Control Structure Generation:

File: controlStructure.py

- The `ControlStructureGenerator` class is responsible for generating control structures.

- Utilizes two data structures:
    1. A binary AST node.
    2. A FIFO queue.

- The process begins with the `generate_control_structures()` method:
    1. Traverses the standardized tree in a pre-order fashion.
    2. Treats lambda nodes as leaves and pushes the right child of each lambda to the FIFO queue.

- After the initial traversal:
    1. The algorithm iterates through the queue.
    2. Traverses the nodes.
    3. Generates control structures.
    4. Assign them to the `map_ctrl_structs` dictionary, using the control structure ID as the key and the list of nodes as values for each control structure.

### CSE Machine Execution:

File: cseMachine.py

The CSE Machine class in `cseMachine.py` is responsible for executing control structures. The execution process begins within the `execute()` method.

1. Initialization:
   - The method starts by pushing the `e0` environment variable, containing empty key values, onto both the control structure list and the stack.

2. Control Structure Setup:
   - Subsequently, it adds the contents of the 0th control structure to the control structure list.

3. Execution:
   - Based on the top elements of the control structure and the stack, the CSE Machine performs operations following the 11 CSE Machine Rules.

## 4. Utility Classes

### 1. Environment:

- File: Environment.py
- Purpose: Environment.py hosts the Environment class, designed to store environment variables as a linked list. This class facilitates traversing the linked list from the tail until the specific variable is located.

### 2. Beta:

- File: controlStructure.py
- Purpose: Within controlStructure.py, the Beta class is employed to represent the beta node in the control structure.

3. **Eta:**

- File: cseMachine.py
- Purpose: Found in cseMachine.py, the Eta class serves to represent the Eta node in the control structure. This node is particularly useful in handling recursive functions.

4. **Tau:**

- File: controlStructure.py
- Purpose: Also situated in controlStructure.py, the Tau class is utilized to represent the n-ary elements within the control structures.

## Conclusion

In conclusion, we were tasked with implementing a lexical analyzer and parser for the RPAL language, referring to RPAL_Lex.pdf for the lexical rules and RPAL_Grammar.pdf for the grammar details. We avoided using tools like and developed our lexer and parser from scratch. Our goal was to produce an Abstract Syntax Tree (AST) as the output of the parser for a given input program. Additionally, we implemented an algorithm to convert the AST into a Standardized Tree (ST) and created a Control Structure Environment (CSE) machine. Throughout the project, we ensured that our program's output matched that of "rpal.exe" for the relevant program. We ran test codes and debugged the system to ensure its accuracy and functionality. This comprehensive approach allowed us to demonstrate proficiency in lexical analysis, parsing, tree transformation, and CSE machine execution.