

# Programming Languages

## CS 3513

### Project Report

Group 48

#### **Members**

Udesh P.V.D	210662R
-------------	---------

Wickramasinghe V.L.A	210710N
----------------------	---------

## Content

Content.....	2
Problem Introduction.....	3
Solution.....	4
Implementation.....	5
Function Prototypes.....	7
Execution.....	12
References.....	12

## Problem Introduction

The purpose of this report is to document the development and implementation of a lexical analyzer and a parser for the RPAL language. The project involves creating these components without using automated tools like 'lex' or 'yacc'. The output of the parser is an Abstract Syntax Tree (AST), which will be further transformed into a Standardized Tree (ST) using a custom algorithm. Additionally, a CSE machine will be implemented to execute the standardized RPAL programs.

RPAL is a functional programming language known for its concise and expressive syntax. The language specification includes a set of lexical rules and a grammar that define its structure and permissible constructs. Implementing a compiler for RPAL involves breaking down the process into several key components: lexical analysis, parsing, AST generation, AST standardization, and execution using a CSE machine.

## Solution

To address the problem of implementing a lexical analyzer and parser for the RPAL language, a multi-step approach was adopted, ensuring a systematic development process. The first step involved designing a lexical analyzer that reads the input RPAL program and converts it into a stream of tokens based on the lexical rules outlined in ``RPAL_Lex.pdf``. This tokenizer was made to handle various RPAL-specific constructs such as keywords, identifiers, operators, and literals.

Following tokenization, a parser was developed to construct the Abstract Syntax Tree (AST) using the grammar rules provided in ``RPAL_Grammar.pdf``. The parser employs recursive descent techniques to ensure a correct hierarchical representation of the input program. Once the AST is generated, an algorithm is implemented to transform this tree into a Standardized Tree (ST), optimizing it for execution. Finally, a CSE machine is designed to interpret and execute the standardized RPAL programs.

# Implementation

- Design and Implement the Lexical Analyzer

Read and understand the lexical rules from RPAL\_Lex.pdf.

Develop a tokenizer that scans the input program and converts it into a sequence of tokens.

Handle different types of tokens such as keywords, identifiers, operators, literals, and delimiters.

Ensure proper error handling for invalid tokens.

- Develop the Parser

Study the grammar rules from RPAL\_Grammar.pdf.

Choose an appropriate parsing technique (ex: recursive descent).

Write parsing functions for each grammatical construct to build the Abstract Syntax Tree (AST).

Implement error detection and recovery mechanisms to handle syntax errors.

- Generate the Abstract Syntax Tree (AST)

Define the data structures to represent nodes of the AST.

Construct the AST during the parsing process, ensuring it accurately represents the hierarchical structure of the input program.

Include necessary metadata in the AST nodes for subsequent processing stages.

- Convert AST to Standardized Tree (ST)

Develop an algorithm to transform the AST into a Standardized Tree (ST).

Ensure the ST involves to the standardized form required for execution by the CSE machine.

Optimize the tree where possible to improve execution efficiency.

- Implement the CSE Machine

Design the architecture of the CSE machine, including the control stack and environment stack.

Write the execution logic to interpret the standardized RPAL programs.

Ensure the CSE machine can handle various control structures and function calls correctly.

Implement necessary built-in functions and operators.

# Function Prototypes

## Analyzer

- Lexical Analyzer

### public Analyzer(String inputFileName)

The Analyzer constructor initializes a new instance of the Analyzer class. It sets up the input file name and prepares a list to hold the tokens that will be identified during the lexical analysis process.

**Regex Patterns and Matchers:** Within the tokenizeLine method (called by scan), various regex patterns are defined to match different token types such as identifiers, integers, operators, strings, and punctuation. The Pattern and Matcher classes from the [java.util.regex](https://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html) package are used for this purpose

**Patterns:** Regular expressions are compiled into Pattern objects for different token types.

**Matchers:** These patterns are used to create Matcher objects which scan the input line and identify substrings that match the defined patterns.

- Parser

### Recursive Parsing Functions

E(), Ew(), T(), Ta(), Tc(), B(), Bt(), Bs(), Bp(), A(), At(), Af(), Ap(), R(), Rn(), D(), Da(), Dr(), Db(), Vb(), Vl()

These functions are part of the parser for the RPAL language. Each function is responsible for parsing different components of the language according to its grammar rules. The functions collectively build the Abstract Syntax Tree (AST) from the input tokens, handling expressions, declarations, boolean operations, arithmetic operations, conditionals, tuples, and function applications.

### **void addStrings(String dots, Node node)**

This function takes string dots representing the indentation or hierarchical level and a Node object representing a node in the Abstract Syntax Tree (AST). It adds a formatted string representation of the node to the stringAST list. The function formats different types of nodes (identifiers, integers, strings, keywords, and function forms) accordingly and uses the dots string to visually indicate the structure of the tree.

- Abstract Syntax tree

### **AST Class**

**AST(Node root):** Constructor to initialize the AST with the given root node.

**void setRoot(Node root):** Sets the root node of the AST.

**Node getRoot():** Returns the root node of the AST.

**void standardize():** Standardizes the AST if it is not already standardized.



**void preOrderTraverse(Node node, int depth):** Performs a pre-order traversal of the AST, printing each node's data with indentation corresponding to its depth.

**void printAst():** Initiates a pre-order traversal from the root node to print the AST structure

### **ASTFactory Class**

**ASTFactory():** Constructor for the ASTFactory class.

**AST getAbstractSyntaxTree(ArrayList<String> data):**  
Creates and returns an AST from a list of string data representing the nodes and their hierarchical structure.

- CSE machine

### **CSEMachine Class**

**CSEMachine(ArrayList<Symbol> control, ArrayList<Symbol> stack, ArrayList<E> environment):**  
Constructor to initialize the CSE machine with control, stack, and environment.

**void setControl(ArrayList<Symbol> control):** Sets the control list of the CSE machine.

**void setStack(ArrayList<Symbol> stack):** Sets the stack list of the CSE machine.

**void setEnvironment(ArrayList<E> environment):** Sets the environment list of the CSE machine.

**void execute():** Executes the CSE machine's instructions based on the control, stack, and environment.

**void printControl():** Prints the current state of the control list.

**void printStack():** Prints the current state of the stack list.

**void printEnvironment():** Prints the current state of the environment.

**Symbol applyUnaryOperation(Symbol rator, Symbol rand):** Applies a unary operation to the given operand.

**Symbol applyBinaryOperation(Symbol rator, Symbol rand1, Symbol rand2):** Applies a binary operation to the given operands.

**String getTupleValue(Tup tup):** Returns the string representation of a tuple's value.

**String getAnswer():** Executes the CSE machine and returns the result as a string.

### **CSEMachineFactory Class**

**public Symbol getSymbol(Node node):** Retrieves the symbol from the given node in the abstract syntax tree (AST).

**public Lambda getLambda(Node node):** Constructs a Lambda symbol from the given node in the AST.

**private ArrayList<Symbol> getPreOrderTraverse(Node node):** Traverses the AST in pre-order and constructs a list of symbols.

**public Delta getDelta(Node node):** Constructs a Delta symbol from the given node in the AST.

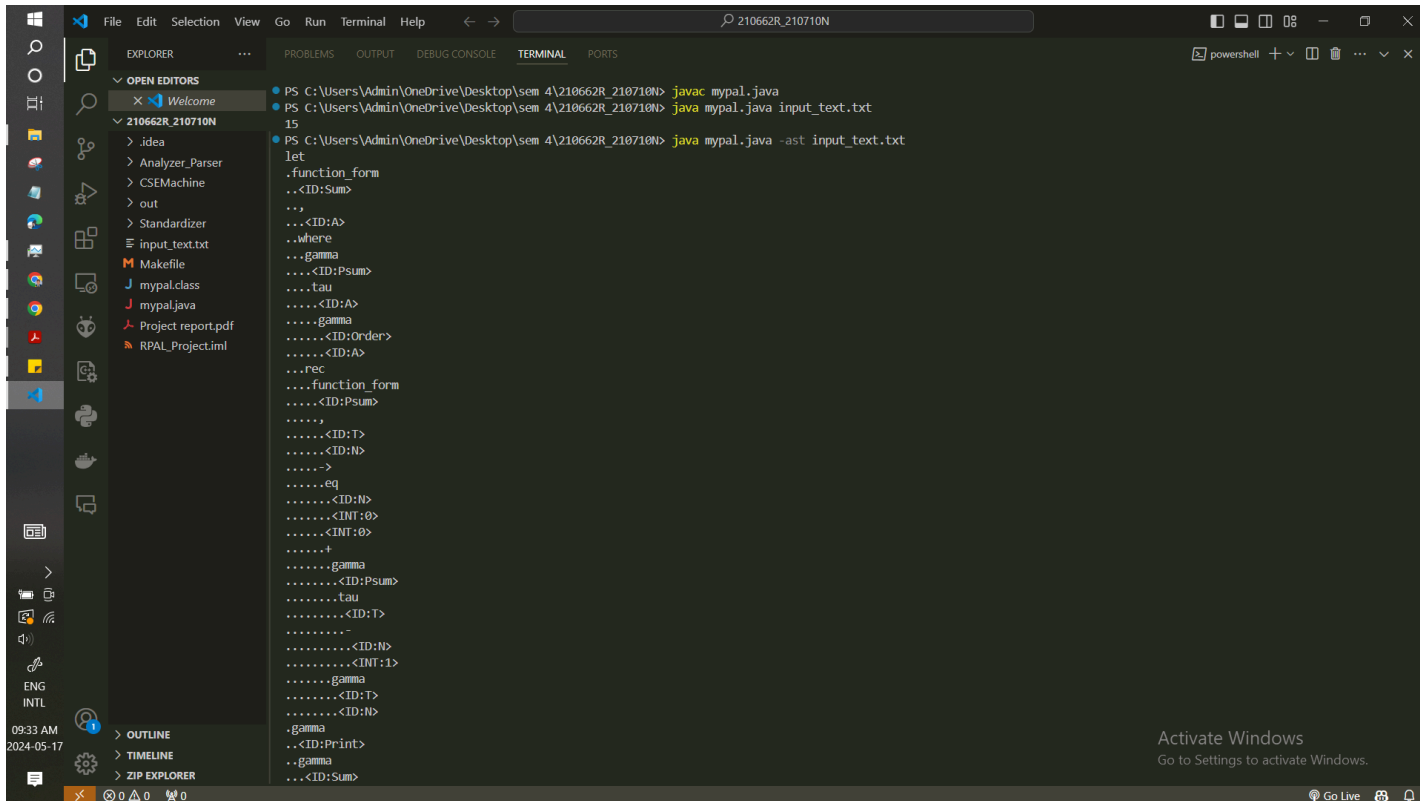
**public ArrayList<Symbol> getControl(AST ast):**  
Constructs the initial control list for the CSE machine based on the AST.

**public ArrayList<Symbol> getStack():** Constructs the initial stack for the CSE machine.

**public ArrayList<E> getEnvironment():** Constructs the initial environment for the CSE machine.

**public CSEMachine getCSEMachine(AST ast):** Constructs a CSE machine with the given AST.

# Execution



```
PS C:\Users\Admin\OneDrive\Desktop\sem 4\210662R_210710N> javac mypal.java
PS C:\Users\Admin\OneDrive\Desktop\sem 4\210662R_210710N> java mypal.java input_text.txt
15
PS C:\Users\Admin\OneDrive\Desktop\sem 4\210662R_210710N> java mypal.java -ast input_text.txt
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>
...<ID:N>
...eq
...<ID:N>
...<ID:N>
...<ID:N>
...+
...gamma
...<ID:Psum>
...tau
...<ID:T>
...<ID:T>
...<ID:N>
...gamma
...<ID:T>
...<ID:N>
...gamma
...<ID:Print>
...gamma
...<ID:Sum>
```

## References

[rpal grammer](#)

[rpal\\_lex](#)

[project file](#)

