CS 3513 Programming Language

RPAL Programming Project

Group Synalyze

Group members

- Mannage K.M.K.K 220384B
- Jayakody J.A.U.C 220248M

Contents

1. Introduction

1.1 Problem Description

This project implements a complete interpreter for the RPAL (Right-reference Pascal) programming language. The interpreter consists of four major components:

1. **Lexical Analyzer** – Tokenizes the input source code based on the lexical rules defined in `RPAL_Lex.pdf`.

2. **Parser** – Constructs an Abstract Syntax Tree (AST) from the token stream using the grammar rules provided in `RPAL_Grammar.pdf`.

3. **AST Standardizer** – Transforms the AST into a Standardized Tree (ST) by eliminating syntactic sugar and normalizing expressions for evaluation.

4. **CSE Machine (Control Stack Environment Machine)** – Evaluates the ST using functional evaluation semantics and produces the final result of the program.

The interpreter reads RPAL source code from an input file and provides two modes of output:

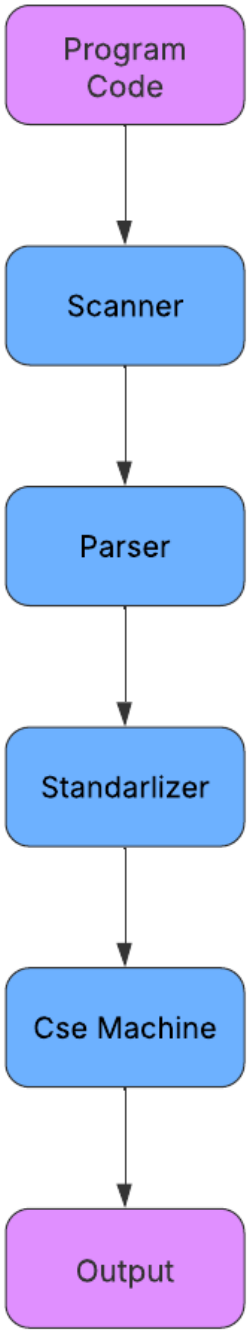
- When invoked with the `-ast` flag, it prints the Abstract Syntax Tree.

- Without the `-ast` flag, it evaluates the program and prints the result.

This system demonstrates key principles of compiler construction including lexical analysis, parsing, tree transformation, and functional program evaluation.

1.2 Implementation process

The following diagram shows a rough sketch of our implementation process.



2. Modules

2.1 Lexical Analyzer (Lexer)

Objective: Convert source code into a stream of meaningful tokens.

Responsibilities:

- **Input:** Raw RPAL source code (from a file).
- **Processing Steps:**
    - Read the input file character by character.
    - Segment the input into **tokens** using lexical rules from RPAL_Lex.pdf.
    - **Classify** each token into one of the categories:
        - Identifiers
        - Integers
        - Reserved Keywords (e.g., let, in, where, function_form)
        - Operators (e.g., +, -, <=, :=)
        - Special symbols (e.g., ;, ,, (,))
    - **Ignore** comments and whitespace.
    - **Return:** A cleaned list of meaningful tokens for the parser.

2.2 Parser

Objective: Generate an Abstract Syntax Tree (AST) from the token stream.

Responsibilities:

- **Input:** Token list from the Lexer.
- **Parsing Method: Recursive Descent Parsing**
    - Use production rules from RPAL_Grammar.pdf to define parsing functions.
    - Each non-terminal symbol corresponds to a recursive parsing function.
- **Construct:** AST nodes representing the syntactic structure of the RPAL program.

Output: AST that reflects the structure but not yet optimized or normalized.

2.3 Standardizer

Objective: Transform the AST into a simplified and normalized form (Standardized Tree - ST).

Responsibilities:

- **Input:** Abstract Syntax Tree (AST).
- **Processing Steps:**
    - Traverse the AST and apply standardization rules.
    - Replace syntactic sugar and complex forms with:
        - lambda nodes (for function abstraction)
        - gamma nodes (for function application)
        - Explicit tau, and Y* constructs
    - Remove let, where, within, rec ,fcn form and @ by transforming them into lambda-based structures.
- **Output:** Standardized Tree (ST) ready for evaluation by the CSE machine.

2.4 Control Stack Environment (CSE) Machine

Objective: Evaluate the standardized tree and compute the result.

Responsibilities:

- **Input:** Standardized Tree (ST).
- **Evaluation Steps:**
    - **Flatten the ST** into a **Control Structure**
    - Use:
        - **Control Stack** (sequence of operations to perform)

        - **Environment Stack** (manages scopes and variable bindings)
        - **Value Stack** (stores intermediate results)
    - **Evaluate** using operational semantics rules.
        - Handle primitive operations, function applications, tuples, conditionals, recursion, etc.
- **Output:** Final result of the RPAL program (integer, boolean, tuple, or closure).

3. File Structure

1. **CSEMachine.java**: Likely implements a Control Stack Evaluation Machine, a mechanism used to execute RPAL expressions by managing control stacks, environments, and continuations.

2. **CSEMachineFactory.java, EvaluatorFactory.java**: These are factory classes that create instances of the CSEMachine and Evaluator, promoting modular design and reusability in the execution process.

3. **Evaluator.java**: Responsible for evaluating RPAL expressions, possibly by interpreting or compiling them into executable form using the CSEMachine.

4. **Lexical_Analyser.java, TestLexicalAnalyser.java**: The lexical analyzer tokenizes the RPAL source code into meaningful symbols (e.g., keywords, identifiers), with a test class to verify its functionality.

5. **Token.java, TokenType.java**: Define the structure and types of tokens generated by the lexical analyzer, essential for parsing.

6. **Parser.java, TestParser.java**: The parser constructs an Abstract Syntax Tree (AST) from the token stream, with a test class to ensure correct parsing of RPAL syntax.

7. **Node.java, NodeType.java, ParserJava**: Represent nodes in the AST and their types, with ParserJava possibly being a specific implementation or helper for parsing.

8. **Standardizer.java, TestStandardizer.java**: Standardizes the AST into a canonical form, a step in RPAL processing to simplify evaluation, with a test class for validation.

9. **AST.java, BuildAST.java, NodeFactory.java**: Handle the construction and management of the AST, with a factory for creating AST nodes.

10. **Symbols, Beta.java, Bool.java, etc.**: These files likely define symbol tables and specific node types (e.g., Beta reduction, Boolean values) used in RPAL's functional programming paradigm, including Greek letters and other constructs like Delta, Eta, Gamma, etc., which may correspond to RPAL's special forms or operators.
11. **input.txt**: Contains sample RPAL code or input data to be processed by the interpreter/compiler.

12. **README.md**: Provides documentation or instructions for the project.

13. **rpal.java**: Possibly the main entry point or a package/class orchestrating the RPAL implementation.

4. Implementation Details

4.1 How Implement an RPAL Program

- Lexical Analysis: Lexical_Analyser.java breaks the RPAL source code into tokens.
- Parsing: Parser.java builds an AST from these tokens.
- Standardization: Standardizer.java transforms the AST into a standard form.
- Evaluation: Evaluator.java and CSEMachine.java execute the standardized AST, leveraging the control stack and environment.
- Support Structures: Files like Node.java, Token.java, and symbol-related classes provide the necessary data structures and types.

4.2 Main Program Execution

The main program is implemented in rpal.java, a Java class that serves as the entry point for an RPAL (Recursive Programming Language) interpreter or compiler. It processes command-line arguments to determine the input file and desired output options, then delegates the evaluation to the Evaluator class in the Engine package.

Command-Line Input Handling

- **Input Format**:
  java rpal <switches> file_name

    - The program expects the user to provide a filename and optional switches (-ast, -st) via the command line.
    - <switches>: Optional flags that control the output behavior (e.g., printing the Abstract Syntax Tree or Standardized Tree).
    - file_name: The name of the file containing the RPAL program to process (defaults to input.txt if not specified).

    - <switches>: If no switches then Rpal program gives only the output.

Switches and Their Behavior

The program supports four modes of operation based on the presence and type of switches:

1. **No Switch**:
    - **Function Called**: Evaluvator.evaluvate(file_name, isPrintAST, isPrintST) in Engine.Evaluvator
    - **Output**: The evaluated result of the input program.
    - **Description**:
        - When no switches are provided, the program defaults to reading from input.txt and enables printing of both the Abstract Syntax Tree (AST) and Standardized Tree (ST).
        - The evaluvate method in Evaluvator (likely a typo for evaluate) processes the RPAL program through lexical analysis, parsing, standardization, and evaluation using a Control Stack Evaluation Machine (CSE Machine), returning the final result.
2. **Switch: -ast** (Abstract Syntax Tree):
    - **Function Called**: Evaluvator.evaluvate(file_name, isPrintAST, isPrintST) with isPrintAST = true
    - **Output**: Prints the AST for the input program and the corresponding output.
    - **Description**:

        - When -ast is specified (alone or with -st), the program reads the RPAL program from the specified file_name, parses it to construct an AST, and prints the AST.
        - The evaluvate method handles the parsing (via Parser.java) and enables AST printing based on the isPrintAST flag.
        - This mode is useful for debugging the syntactic structure of the program.
3. **Switch: -st** (Standardized Tree):
    - **Function Called**: Evaluvator.evaluvate(file_name, isPrintAST, isPrintST) with isPrintST = true
    - **Output**: Prints the Standardized Tree (ST) for the input program and the corresponding output .
    - **Description**:
        - When -st is specified (alone or with -ast), the program reads the RPAL program, parses it to generate an AST, standardizes it into an ST using Standardizer.java, and prints the ST.
        - The evaluvate method manages the standardization process and enables ST printing based on the isPrintST flag.
        - This mode helps verify the transformation of the AST into a canonical form for evaluation.
4. **Combination: -ast -st or -st -ast**:
    - **Function Called**: Evaluvator.evaluvate(file_name, isPrintAST, isPrintST) with isPrintAST = true and isPrintST = true
    - **Output**: Prints both the AST and ST for the input program, followed by the evaluation result.
    - **Description**:
        - When both switches are provided (in any order), the program prints the AST, the ST, and the final result of the evaluation.
        - This mode combines debugging output with the program's execution result.

4.3 Lexical Analyzer

Implemented in **Lexical_Analyzer**

The lexical analyzer, implemented in LexicalAnalyser.java, is a core component of the RPAL interpreter, responsible for tokenizing input RPAL programs. It reads the input file (default: input.txt) line by line and uses regular expressions to identify tokens such as keywords (e.g., let, fn), identifiers, integers, operators, strings, and punctuation. The Token.java class defines the token structure with a type (from TokenType.java enum: KEYWORD, IDENTIFIER, INTEGER, OPERATOR, STRING, PUNCTUATION, etc.) and a value. The analyzer skips spaces and comments (using //.* pattern), categorizes tokens, and stores them in a list. The TestLexicalAnalyzer.java class tests this functionality by scanning the input file and printing tokens in the format <type, value>. If

an unrecognized character is encountered, it throws an exception with the line number and character details, ensuring robust error handling during lexical analysis.

4.4 Parser

The parser is a critical component of the RPAL (Recursive Programming Language) interpreter, responsible for constructing an Abstract Syntax Tree (AST) from tokenized input. The implementation is distributed across several Java files in the Parser package, with supporting classes from the Lexical_Analyser package. Below are the details of each file and their roles in the parsing process.

NodeType.java

- **Description**: Defines an enum (NodeType) that specifies the types of nodes used in the AST. These types represent RPAL constructs such as let, fcn_form, identifier, integer, string, where, gamma, lambda, tau, rec, aug, conditional, and various operators (op_or, op_and, op_not, etc.), along with constants like true_value, false_value, nil, and dummy.
- **Purpose**: Provides a standardized set of node types to classify the structure of RPAL expressions, ensuring consistent AST construction.
- **Key Features**: Includes a comprehensive list of RPAL-specific constructs, enabling the parser to differentiate between control structures, operators, and literals.

Node.java

- **Description**: Implements the Node class, which represents a node in the AST. Each node has:
    - type: A NodeType enum value indicating the node's category.
    - value: A String holding the node's specific value (e.g., an identifier name or integer).
    - noOfChildren: An int representing the number of child nodes, with the root node's depth set to 0.
- **Purpose**: Serves as the building block for the AST, allowing the parser to construct a hierarchical representation of the RPAL program.
- **Key Features**: The constructor initializes a node with its type, value, and child count, facilitating tree traversal and manipulation.

Parser.java

- **Description**: Implements the Parser class, which processes a list of Token objects (generated by LexicalAnalyser) to build an AST. It includes methods to parse RPAL expressions according to a recursive descent parsing strategy based on the RPAL grammar.

- **Purpose**: Converts tokenized input into a structured AST, which is later used for standardization and evaluation.
- **Key Features**:
    - **Initialization**: Takes a List<Token> as input and initializes an AST (ArrayList<Node>) and stringAST (ArrayList<String>) to store the tree and its string representation.
    - **Parsing Logic**: The parse() method initiates the parsing process with the E() method, which handles the top-level expression grammar rules (e.g., let D in E, fn Vb+ . E, or Ew). It appends a TokenType.EndOfTokens token to detect parsing completion.
    - **Grammar Rules**: Implements recursive methods (E, Ew, T, Ta, Tc, B, Bt, Bs, Bp, A, At, Af, Ap, R, Rn, D, Da, Dr, Db, Vb, Vl) corresponding to RPAL's grammar, building the AST by adding nodes with appropriate NodeType, value, and noOfChildren.
    - **Error Handling**: Reports parsing errors (e.g., missing in,), or unexpected tokens) and returns null if parsing fails, printing unparsed tokens for debugging.
    - **String Representation**: The convertAST_toStringAST() method transforms the AST into a dotted string format (e.g., <ID:value>, <INT:value>), using a stack-based approach to traverse the tree and generate a readable output.

TestParser.java

- **Description**: Implements the TestParser class, a test driver that demonstrates the parser's functionality. It reads tokens from input.txt using LexicalAnalyser, parses them into an AST, and prints the tokens, AST nodes, and string representation.
- **Purpose**: Verifies the correctness of the lexical analyzer and parser by processing a sample RPAL file and displaying the results.
- **Key Features**:
    - **Setup**: Initializes a LexicalAnalyser with input.txt and retrieves tokens via scan().
    - **Execution**: Creates a Parser instance with the tokens, calls parse() to build the AST, and prints each node in the format <type, value, noOfChildren>.
    - **Output**: Converts the AST to a string format using convertAST_toStringAST() and prints the result, aiding in visual inspection of the tree structure.
    - **Error Handling**: Catches and displays exceptions from the scanning or parsing process.

4.5 Standardizer

The Standardizer is a key component of the RPAL interpreter, responsible for transforming the raw Abstract Syntax Tree (AST) produced by the Parser into a standardized form. This transformation simplifies downstream processing (such as interpretation or code generation)

by ensuring all syntactic sugar and alternative forms are converted into a consistent, minimal core representation.
The implementation involves several Java files: Standardizer.java, Node.java, and NodeType.java (shared with the Parser), and it works in coordination with the AST and BuildAST classes.

Standardizer.java

- **Description**:Contains the core logic for standardizing the AST. It recursively traverses the tree and applies transformation rules to specific node types such as let, where, lambda, rec, and within.
- **Purpose:**To convert high-level RPAL constructs into a simplified and uniform structure, making interpretation and evaluation more straightforward.
- **Key Features:**
    - **Entry Point:** The standardize(Node node) method is the entry point, invoked on the root of the AST. It uses pre-order traversal to standardize child nodes before applying transformations to the current node.
    - **Utility Methods:** Includes helpers like createNode, makeLambda, makeGamma, and setChildren to create new tree structures easily.
    - **Preservation:** The original structure is not overwritten; instead, transformations create new nodes with updated relationships.

Node.java

- **Description:**Defines the Node class, used by the Standardizer to navigate and rebuild parts of the AST. Each node contains its type (NodeType), value (String), list of children (ArrayList<Node>), depth, and a reference to the parent node.
- **Purpose:**To encapsulate all structural and contextual information needed for standardization operations.
- **Key Features:**
    - Nodes can dynamically update their children and parent relationships during the transformation.
    - Provides methods to add and manipulate children (addChild, setChildren), which are crucial for rewriting subtrees.

AST.java and BuildAST.java

- AST.java provides the standardizeAST() method which calls Standardizer.standardize() on the root node of the AST and updates the internal representation.

- BuildAST.java constructs the initial AST from a string-formatted version (e.g., dotted notation). After building, the AST is passed to the Standardizer for transformation.

TestStandardizer.java

- **Description:** A test driver class that demonstrates the entire pipeline: scanning, parsing, AST building, standardization, and final output.
- **Purpose:**To verify that standardization rules are applied correctly and to provide visual inspection of the transformed tree.
- **Key Features:**
    - Loads input from input.txt, processes it through all interpreter stages, and prints the standardized AST.

4.6 CSE Machine

Evaluvator.java

Purpose: This is the main entry point for evaluating an RPAL program. It orchestrates the entire process, from lexical analysis to the execution of the CSE Machine, and returns the final result.

Key Components:

- **Input**: Takes a filename (the RPAL program to evaluate) and two boolean flags: isPrintAST (to print the AST) and isPrintST (to print the standardized AST).
- **Process**:
    - **Lexical Analysis**: Uses LexicalAnalyser to tokenize the input RPAL program. If no tokens are produced (empty program), it outputs a message(*Empty PROGRAM!!!!!!!!*) and returns an empty string.
    - **Parsing**: Uses Parser to parse the tokens into an AST, represented as a list of Node objects. If isPrintAST is true, it prints the AST as strings.
    - **AST Standardization**: Uses BuildAST to convert the string-based AST into a standardized AST (using the AST class). If isPrintST is true, it prints the standardized AST.
    - **CSE Machine Execution**: Creates a CSEMachineFactory to generate a CSEMachine instance from the standardized AST, then calls getAnswer() to execute the machine and retrieve the result.
- **Error Handling**: Catches exceptions during processing and prints error messages, returning null if an error occurs.
- **Output**: Returns the result of the program as a string, obtained from the CSE Machine.

Role in RPAL: Acts as the high-level coordinator, tying together the lexical analysis, parsing, AST standardization, and CSE Machine execution phases.

CSEMachineFactory.java

Purpose: This class is responsible for constructing the components of the CSE Machine, including the control structure, stack, and environment, based on the standardized AST.

Key Components:

- **Instance Variables**:
  - e0: The initial environment (index 0).
  - i: A counter for assigning unique indices to Lambda nodes.
  - j: A counter for assigning unique indices to Delta nodes.
- **Key Methods**:
  - **getSymbol(Node)**: Maps an AST node to a corresponding Symbol object based on its type:
    - **Unary Operators**: not, neg (returns Uop).
    - **Binary Operators**: +, -, *, /, **, &, or, eq, ne, ls, le, gr, ge, aug (returns Bop).
    - **Special Constructs**: gamma (returns Gamma), tau (returns Tau with the number of children), <Y*> (returns Ystar for recursion).
    - **Operands**: Identifiers (<ID:>), integers (<INT:>), strings (<STR:>), nil, true, false, dummy (returns Id, Int, Str, Tup, Bool, or Dummy).
    - **Error Case**: Returns Err for unrecognized nodes.
  - **getLambda(Node)**: Creates a Lambda symbol for a lambda node, assigning it a unique index, extracting identifiers (single or multiple), and associating it with a Delta (body of the lambda).
  - **getDelta(Node)**: Creates a Delta symbol with a unique index, containing symbols from a preorder traversal of the node.
  - **getB(Node)**: Creates a B symbol (for conditional branches) containing symbols from a preorder traversal.
  - **getPreOrderTraverse(Node)**: Performs a preorder traversal of the AST node to generate a list of Symbol objects, handling lambda, conditional (->), and other nodes.
  - **getControl(AST)**: Initializes the control structure with the initial environment (e0) and a Delta for the root node.
  - **getStack()**: Initializes the stack with the initial environment (e0).
  - **getEnvironment()**: Initializes the environment list with e0.
  - **getCSEMachine(AST)**: Constructs a CSEMachine instance with the control, stack, and environment.

Role in RPAL: Translates the standardized AST into the control structures, stack, and environment needed for the CSE Machine to evaluate the program. It handles the creation of symbols for operators, operands, and special constructs like lambdas and conditionals.

CSEMachine.java

Purpose: Implements the CSE Machine, which executes the RPAL program by processing the control structure, stack, and environment according to predefined rules.

Key Components:

- **Instance Variables**:
  - control: List of Symbol objects representing the control structure (instructions to execute).
  - stack: List of Symbol objects representing the evaluation stack.
  - environment: List of E (environment) objects for variable bindings.
- **Execution (execute)**: The execute method runs the CSE Machine by processing symbols from the control structure until it is empty. It implements the following rules:
  - **Rule 1 (Id)**: Looks up an identifier in the current environment and pushes its value onto the stack.
  - **Rule 2 (Lambda)**: Pushes a lambda (with its environment index) onto the stack.
  - **Rule 3, 4, 10, 11, 12, 13 (Gamma)**: Handles function application:
    - **Lambda (Rule 4, 11)**: Creates a new environment, binds identifiers to stack values (single or tuple), links to the parent environment, and adds the lambda's delta to the control.
    - **Tuple (Rule 10)**: Selects the i-th element from a tuple based on an index on the stack.
    - **Ystar (Rule 12)**: Creates an Eta for recursive functions, linking to a lambda.
    - **Eta (Rule 13)**: Handles recursive function application by adding gamma and lambda symbols.
    - **Built-in Functions**: Handles functions like Print, Stem, Stern, Conc, Order, Isinteger, Isstring, Istuple, Isdummy, Istruthvalue, Isfunction.
  - **Rule 5 (E)**: Removes an environment from the stack and updates the current environment to the most recent non-removed environment.
- **Rule 6, 7 (Rator)**: Applies unary (neg, not) or binary operators (+, -, *, /, **, &, or, eq, ne, ls, le, gr, ge, aug) by popping operands from the stack and pushing the result.
- **Rule 8 (Beta)**: Implements conditional execution by selecting the appropriate branch (then or else) based on a boolean value on the stack.
- **Rule 9 (Tau)**: Creates a tuple by popping n elements from the stack and pushing a Tup symbol.
- **Delta and B**: Expands Delta or B symbols by adding their symbols to the control structure.
- **Default**: Pushes other symbols (e.g., constants) onto the stack.

Helper Methods:

- **applyUnaryOperation**: Implements unary operations (neg, not).
- **applyBinaryOperation**: Implements binary operations, including arithmetic, logical, comparison, and tuple augmentation (aug).
- **getTupleValue**: Converts a Tup symbol into a string representation (e.g., (1, 2, 3)).
- **getAnswer**: Runs execute and returns the top stack element as a string (tuple or single value).
- **printControl, printStack, printEnvironment**: Debugging methods to print the state of the CSE Machine.

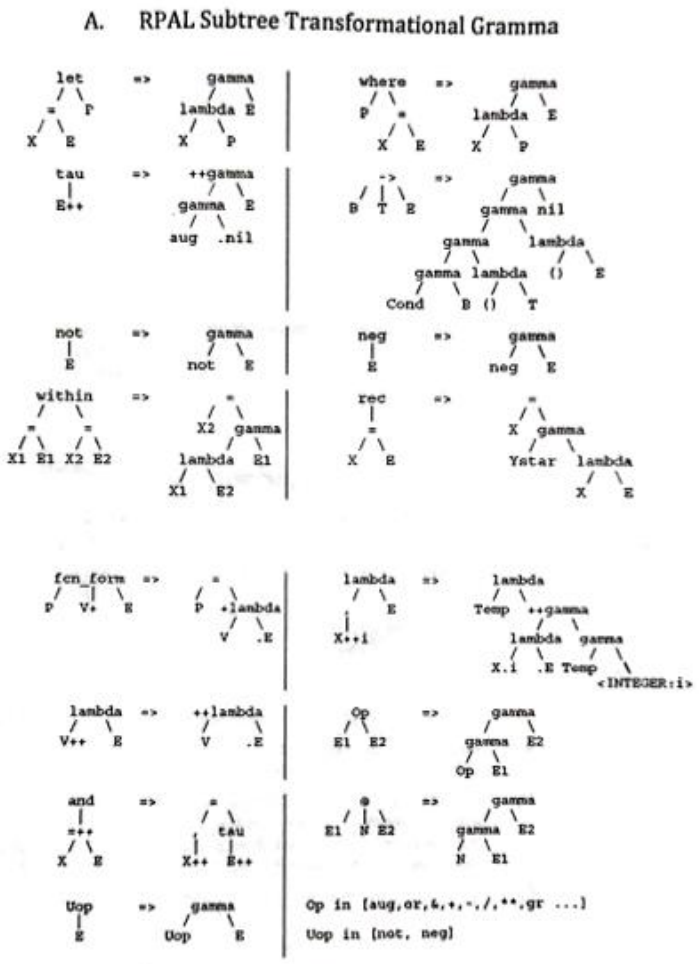
Overall Description

The RPAL CSE Machine implementation processes an RPAL program through the following stages:

1. **Lexical Analysis and Parsing**: Evaluvator.java uses LexicalAnalyser and Parser to convert the program into an AST.
2. **AST Standardization**: The AST is standardized using BuildAST to ensure a consistent structure for evaluation.
3. **CSE Machine Setup**: CSEMachineFactory.java constructs the control structure, stack, and environment from the standardized AST, mapping nodes to symbols (e.g., operators, lambdas, tuples).
4. **Execution**: CSEMachine.java evaluates the program by applying rules to process symbols, handling function applications, conditionals, operators, and tuples, and producing the final result.

5. References

https://github.com/KevithMannage/programming_language.git



RPAL's LEXICON:

```
Identifier -> Letter (Letter | Digit | '_')*                  => '<IDENTIFIER>';

Integer    -> Digit+                                         => '<INTEGER>';

Operator   -> Operator_symbol+                               => '<OPERATOR>';

String     -> ''''
              ( '\' 't' | '\' 'n' | '\' '\' | '\' ''''
              | '('     | ')'     | ':'     | '.'
              | ' '
              | Letter | Digit | Operator_symbol
              )* ''''                                        => '<STRING>';

Spaces     -> ( ' ' | ht | Eol )+                            => '<DELETE>';

Comment    -> '//'
              ( '''' | '(' | ')' | ':' | '.' | '\' | ' '
                | ht | Letter | Digit | Operator_symbol
              )* Eol                                         => '<DELETE>';

Punction   -> '('                                           => '('
           -> ')'                                            => ')'
           -> ':'                                            => ':'
           -> '.'                                            => '.';

Letter     -> 'A'..'Z' | 'a'..'z';

Digit      -> '0'..'9';

Operator_symbol
           -> '+' | '-' | '*' | '<' | '>' | '&' | '.'
            | '@' | '/' | ':' | '=' | '~' | '|' | '$'
            | '!' | '#' | '%' | '^' | '_' | '[' | ']'
            | '{' | '}' | '"' | '`' | '?';
```

RPAL's Phrase Structure Grammar:

```
# Expressions ###############################################

E    -> 'let' D 'in' E                          => 'let'
     -> 'fn'  Vb+ '.' E                         => 'lambda'
     ->  Ew;
Ew   -> T  'where' Dr                           => 'where'
     -> T;

# Tuple Expressions #########################################

T    -> Ta ( ',' Ta )+                          => 'tau'
     -> Ta ;
Ta   -> Ta 'aug' Tc                             => 'aug'
     -> Tc ;
Tc   -> B '->' Tc '|' Tc                        => '->'
     -> B ;

# Boolean Expressions #######################################

B    -> B 'or' Bt                               => 'or'
     -> Bt ;
Bt   -> Bt '&' Bs                               => '&'
     -> Bs ;
Bs   -> 'not' Bp                                => 'not'
     -> Bp ;
Bp   -> A ('gr' | '>' ) A                       => 'gr'
     -> A ('ge' | '>=') A                       => 'ge'
     -> A ('ls' | '<' ) A                       => 'ls'
     -> A ('le' | '<=') A                       => 'le'
     -> A 'eq' A                                => 'eq'
     -> A 'ne' A                                => 'ne'
     -> A ;

# Arithmetic Expressions ####################################

A    -> A '+' At                                => '+'
     -> A '-' At                                => '-'
     ->   '+' At
     ->   '-' At                                => 'neg'
     -> At ;
At   -> At '*' Af                               => '*'
     -> At '/' Af                               => '/'
     -> Af ;
Af   -> Ap '**' Af                              => '**'
     -> Ap ;
Ap   -> Ap '@' '<IDENTIFIER>' R                 => '@'
     -> R ;

# Rators And Rands ##########################################

R    -> R Rn                                    => 'gamma'
     -> Rn ;
Rn   -> '<IDENTIFIER>'
     -> '<INTEGER>'
     -> '<STRING>'
     -> 'true'                                  => 'true'
     -> 'false'                                 => 'false'
     -> 'nil'                                   => 'nil'
     -> '(' E ')'
     -> 'dummy'                                 => 'dummy' ;
```

```
# Definitions ################################################

D     -> Da 'within' D                              => 'within'
      -> Da ;
Da    -> Dr ( 'and' Dr )+                           => 'and'
      -> Dr ;
Dr    -> 'rec' Db                                   => 'rec'
      -> Db ;
Db    -> Vl '=' E                                   => '='
      -> '<IDENTIFIER>' Vb+ '=' E                   => 'fcn_form'
      -> '(' D ')' ;

# Variables ##################################################

Vb    -> '<IDENTIFIER>'
      -> '(' Vl ')'
      -> '(' ')'                                    => '()';
Vl    -> '<IDENTIFIER>' list ','                    => ','?;
```