Report

GROUP 44

PREPARED BY :

210329E - Laksara K.Y
210523T - Ranaweera H.K

1. Introduction

RPAL (Right-reference Pedagogic Algorithmic Language) is a functional programming language. It is designed for teaching and experimenting with concepts related to functional programming, including higher-order functions, recursion, pattern matching, and lambda calculus.

The RPAL language follows a formal grammar and lexical rules outlined in RPAL_Lex.pdf and RPAL_Grammar.pdf documents. These documents define the syntax and semantics of RPAL programs, including tokenization rules, grammar productions, and language constructs.

This report presents the design and implementation details of a lexical analyzer, parser, Abstract Syntax Tree (AST), Standardized Tree (ST), and CSE Machine for the RPAL language. The implementation is done using python.

2. Problem Statement

The task at hand involves the development of a lexical analyzer and parser for the RPAL language, adhering strictly to the lexical rules and grammar specifications provided in the documents RPAL_Lex.pdf and RPAL_Grammar.pdf, respectively. The implementation should not rely on external tools such as 'lex' or 'yacc'. The primary objective is to produce an Abstract Syntax Tree (AST) from the input RPAL program, following the parsing process. Subsequently, an algorithm must be devised to transform the AST into a Standardized Tree (ST), which will serve as the basis for implementing the Control Stack Environment (CSE) machine. The program should be capable of reading RPAL programs from input files and generating output that matches the output produced by "myrpal.exe" for the corresponding program.

The report will detail the design, implementation, and functionality of the lexical analyzer, parser, AST generation, ST conversion algorithm, and CSE machine implementation, ensuring compliance with the specified requirements and achieving parity with the expected output of "myrpal.exe."

3. Program Execution Instructions

The following sequence of commands is applicable within the **root directory** of the project for compiling the program and executing RPAL programs.

Note: The development of this project was undertaken in a Windows environment, and a sample demonstration is provided separately in the following section.

- **<u>To compile the program and execute RPAL programs:</u>**

  ```
  > make
  > ./myrpal file_name
  ```

  *Note:Upon executing the above mentioned commands, the **output of the lexical analyzer,** which represents the tokenized input RPAL program, is also directed to an `**output_token_sequence.txt**` file.*

- **<u>To compile all components of the program:</u>**

  ```
  > make all
  ```

  This command compiles all components or targets specified in the `Makefile`.

- **<u>To compile the program and then execute it:</u>**

  ```
  > make run
  ```

  This command performs a complete build of the program and then runs the resulting executable, typically configured in the `Makefile`.

- **<u>To compile the program and generate the Tokenized Sequence:</u>**

  ```
  > make lex
  ```

  This command compiles the entire program and then runs it with the `-lex` option to generate the Tokenized Sequence. The tokenized sequence is saved to an `***output_token_sequence.txt***` file.

- **<u>To compile the program and generate Abstract Syntax Tree (AST) module:</u>**

  ```
  > make ast
  ```

This command compiles the entire program and then runs it with the `-ast` option to generate the Abstract Syntax Tree.

- **<u>To compile the program and generate Standardized Tree (ST) module:</u>**

  ```
  > make st
  ```

  This command compiles the entire program and then runs it with the `-st` option to generate the Standardized Tree.

- **<u>To generate the Tokenized Sequence:</u>**

  ```
  > ./myrpal -lex input.txt
  ```

- **<u>To generate the Abstract Syntax Tree:</u>**

  ```
  > ./myrpal -ast file_name
  ```

- **<u>To generate the Standardized Tree:</u>**

  ```
  > ./myrpal -st file_name
  ```

*Note: In the code implementation, the file named **input.txt** is used as the file_name*

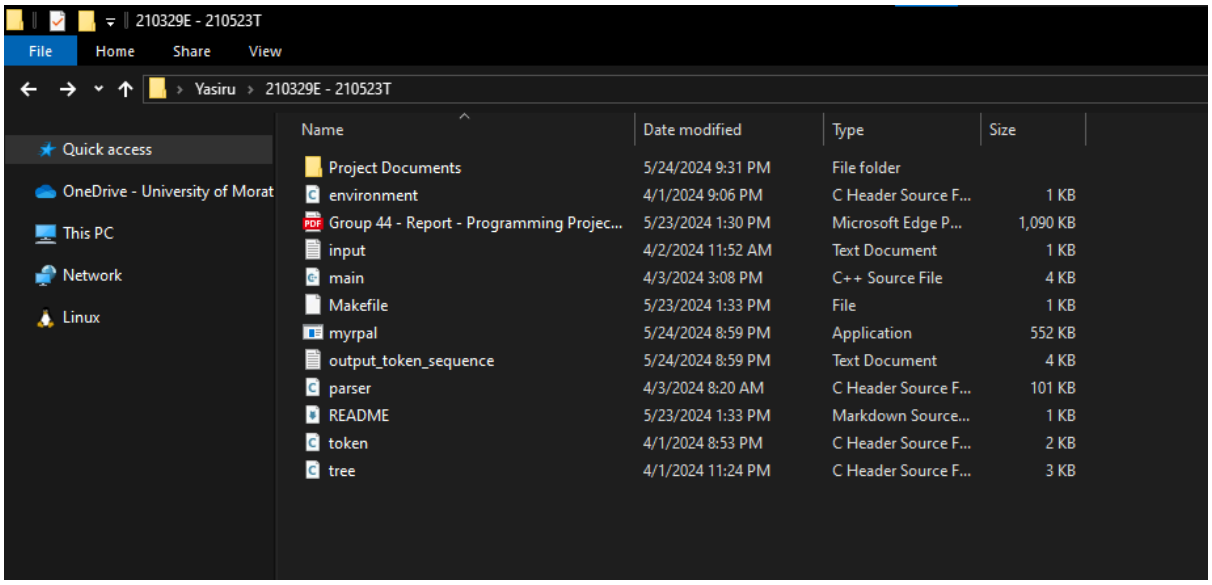
4. Running the Program in a Windows Environment

This section provides step-by-step instructions for running the RPAL program in the **Windows environment**.

Steps to Run the Program

1) Open the Root Folder

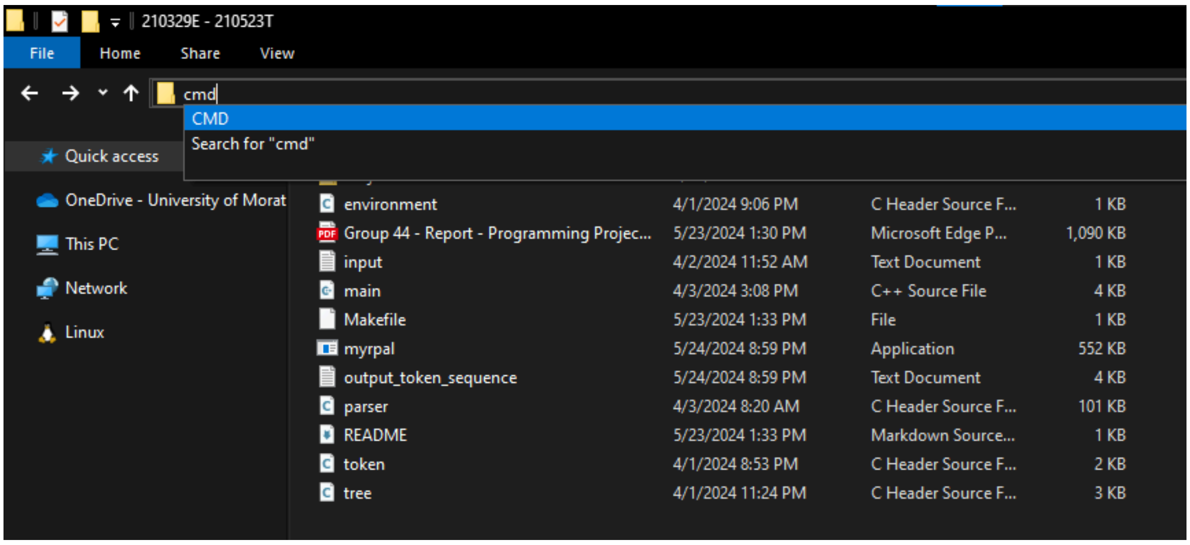
Navigate to the root folder of the project where the Makefile and main.cpp are located.



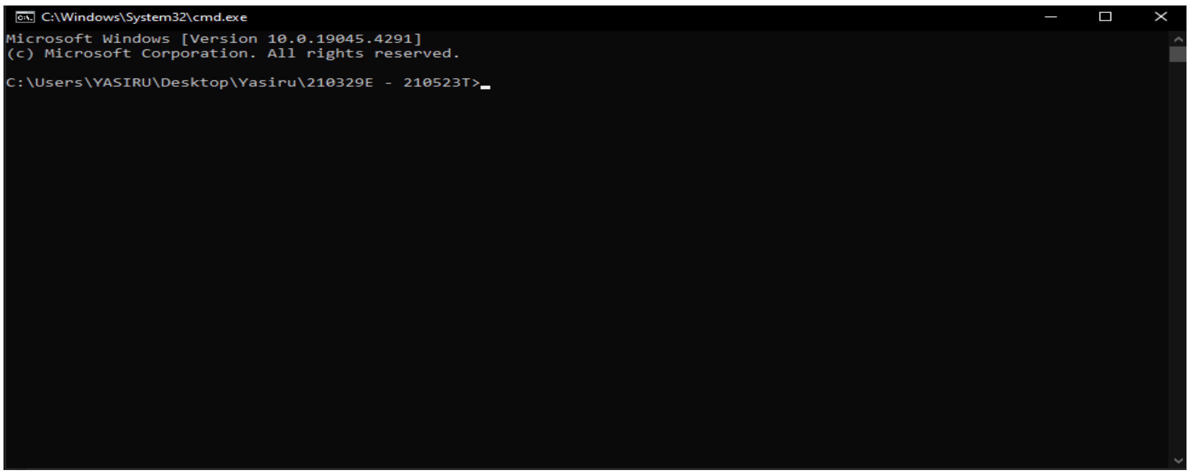
Name	Date modified	Type	Size
Project Documents	5/24/2024 9:31 PM	File folder	
environment	4/1/2024 9:06 PM	C Header Source F...	1 KB
Group 44 - Report - Programming Projec...	5/23/2024 1:30 PM	Microsoft Edge P...	1,090 KB
input	4/2/2024 11:52 AM	Text Document	1 KB
main	4/3/2024 3:08 PM	C++ Source File	4 KB
Makefile	5/23/2024 1:33 PM	File	1 KB
myrpal	5/24/2024 8:59 PM	Application	552 KB
output_token_sequence	5/24/2024 8:59 PM	Text Document	4 KB
parser	4/3/2024 8:20 AM	C Header Source F...	101 KB
README	5/23/2024 1:33 PM	Markdown Source...	1 KB
token	4/1/2024 8:53 PM	C Header Source F...	2 KB
tree	4/1/2024 11:24 PM	C Header Source F...	3 KB

2) Open Command Prompt

Open the Command Prompt and navigate to the root folder of your project. You can do this by typing cmd in the address bar of the File Explorer and pressing Enter.



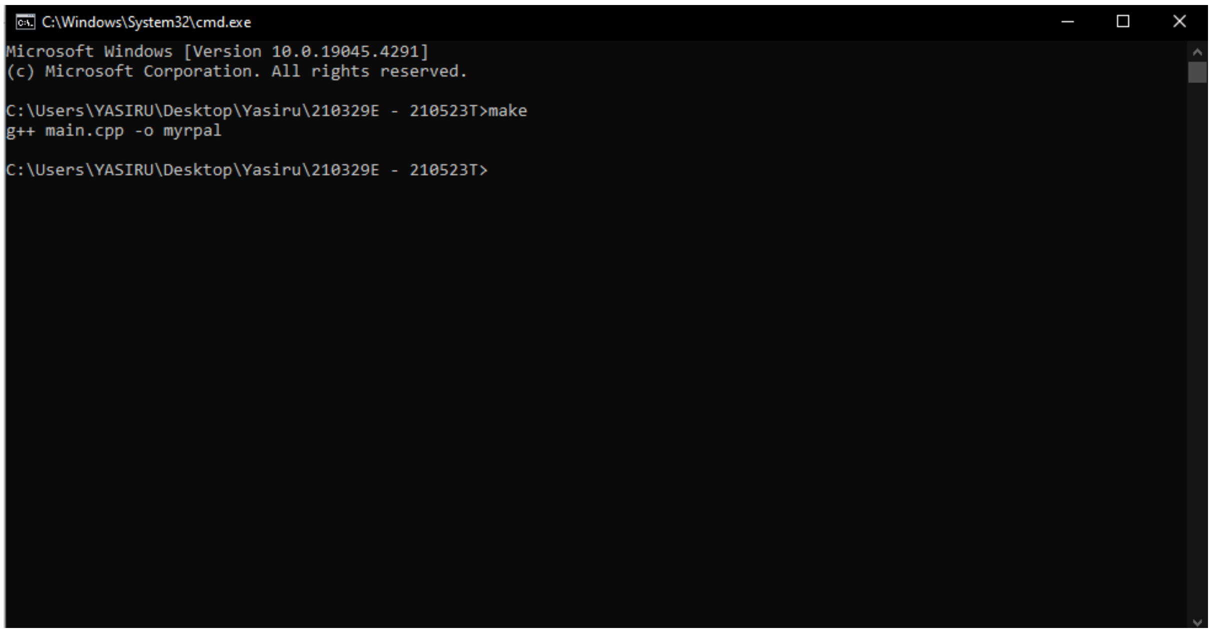
environment	4/1/2024 9:06 PM	C Header Source F...	1 KB
Group 44 - Report - Programming Projec...	5/23/2024 1:30 PM	Microsoft Edge P...	1,090 KB
input	4/2/2024 11:52 AM	Text Document	1 KB
main	4/3/2024 3:08 PM	C++ Source File	4 KB
Makefile	5/23/2024 1:33 PM	File	1 KB
myrpal	5/24/2024 8:59 PM	Application	552 KB
output_token_sequence	5/24/2024 8:59 PM	Text Document	4 KB
parser	4/3/2024 8:20 AM	C Header Source F...	101 KB
README	5/23/2024 1:33 PM	Markdown Source...	1 KB
token	4/1/2024 8:53 PM	C Header Source F...	2 KB
tree	4/1/2024 11:24 PM	C Header Source F...	3 KB



```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\YASIRU\Desktop\Yasiru\210329E - 210523T>
```

3) **Compile the Program**

   In the Command Prompt, type the **make** command to compile the program.

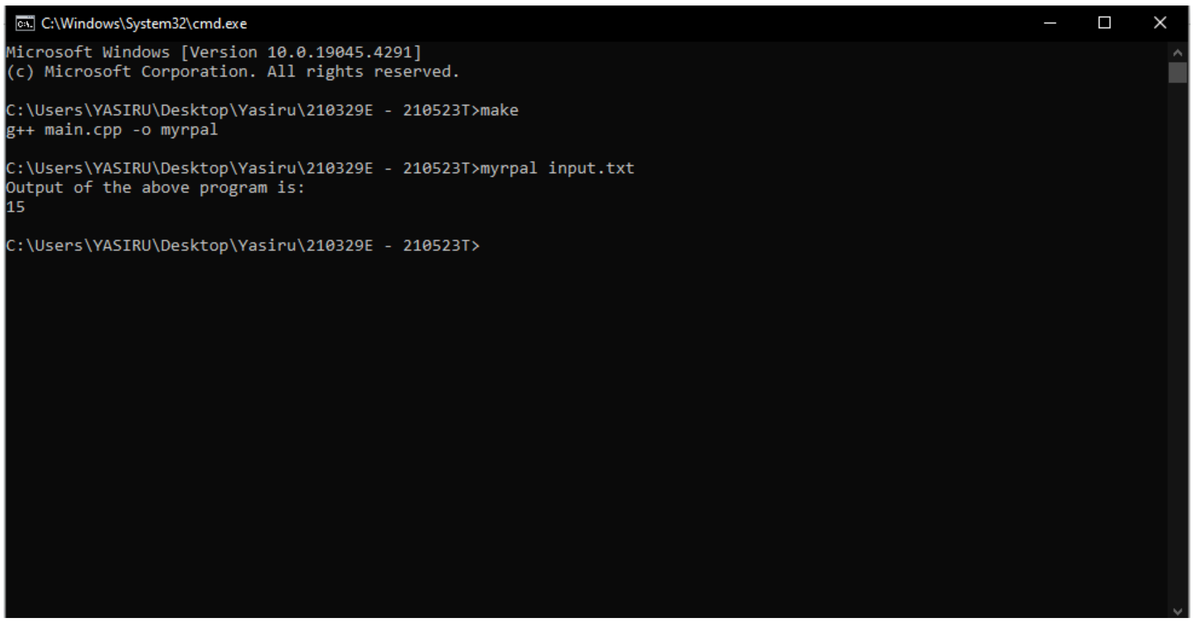


```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\YASIRU\Desktop\Yasiru\210329E - 210523T>make
g++ main.cpp -o myrpal

C:\Users\YASIRU\Desktop\Yasiru\210329E - 210523T>
```

4) **Run the Program**

   After compiling, run the program with the input file by typing the **myrpal input.txt** command.

   Note:- Here, we used input.txt as the file name. Replace input.txt with the actual file you wish to process.



```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\YASIRU\Desktop\Yasiru\210329E - 210523T>make
g++ main.cpp -o myrpal

C:\Users\YASIRU\Desktop\Yasiru\210329E - 210523T>myrpal input.txt
Output of the above program is:
15

C:\Users\YASIRU\Desktop\Yasiru\210329E - 210523T>
```

By following above instructions, we should be able to compile and run the RPAL programs successfully in the Windows environment.

5. Program Structure

The program is structured into several components.

Components	**Description**
Lexical Analyzer	Responsible for tokenizing input RPAL programs based on the lexical rules defined in RPAL_Lex.pdf.
Parser	Implements the parsing logic according to the grammar rules specified in RPAL_Grammar.pdf.
AST Generator	Constructs the Abstract Syntax Tree (AST) from the parsed tokens.
ST Converter	Converts the AST into Standardized Tree (ST) format.
CSE Machine	Implements the CSE (Control, Stack, and Environment) Machine to execute RPAL programs.

6. Implementation Details

6.1 Lexical Analyzer

Note: The implementation of the Lexical Analyzer resides within the **`get_token(char read[])`** *function, which is encapsulated within the `parser` class declared in the `parser.h` file..*

6.1.1 Functionality

- ➔ Reads the input file line by line.
- ➔ Matches tokens using regular expressions defined for each token type.
- ➔ Handles errors for unrecognized characters.
- ➔ Generates tokens and stores them in a list.

6.1.2 Implementation

- ➔ Defines token types and their corresponding regular expressions.
- ➔ Implements a **`get_token`** function to tokenize the input file.

Functionality of the **`get_token`** function

- Initialization and Token Declaration

  At the start of the function, several variables are declared to store different types of tokens (`id`, `num`, `isop`, `isString`, `isPunctuation`, `isComment`, `isSpace`) and an object of the `token` class (`t`) is created.

- End of File Check

  The function first checks if the end of the file has been reached. If so, it creates a token with type and value set to "EOF" and returns it.

- Tokenization Loop

  The function enters a loop to read characters from the input string (`read`) and tokenize them.

Token Types	**Description**
IDENTIFIER	Represents identifiers in the RPAL program.
INTEGER	Represents integer literals.
OPERATOR	Represents operators such as `+`, `-`, `*`, etc.
STRING	Represents string literals enclosed in triple quotes (`'''`).
PUNCTUATION	Represents punctuation characters like `()`, `,`, etc.
WHITESPACE	Represents whitespace characters.
COMMENT	Represents comments starting with `//`.
ERROR	Represents unrecognized characters causing errors.

- Token Assignment and Output

  After identifying the token type and value, the function assigns them to the `t` object and writes the token details to an output file (`output_token_sequence.txt`) in a specified format.

6.2 parser.h

The `parser.h` file houses the Recursive Descent Parser, which undertakes the crucial tasks of tokenizing, parsing, and transforming input code into a Standardized Tree (ST) representation for subsequent execution. Here's an overview of the program's structure and functionalities:

6.2.1 Implementation

1. Tokenization

   The parser begins by tokenizing input code through the `getToken()` function. This process involves categorizing individual characters into distinct token types like identifiers, keywords, operators, integers, strings, punctuation, comments, spaces, and unknown tokens. This initial tokenization lays the groundwork for subsequent parsing operations.

2. Abstract Syntax Tree (AST) and Standardized Tree (ST)

    - The **`buildTree()`** function constructs an Abstract Syntax Tree (AST) by creating tree nodes based on token properties, which are then added to the syntax tree stack (`st`). The AST captures the syntactic structure of the input code.
    - The **`makeST()`** function transforms the AST into a Standardized Tree (ST) by standardizing its representation. This standardized form ensures uniformity in the tree structure, preparing the code for further execution.

3. Control Structures Execution
    - Following ST construction, control structures are generated using **`createControlStructures().`** These structures outline the instructions necessary for executing the code within a Control Stack Environment (CSE) machine, which is a theoretical model for executing high-level functional programming languages.
    - The **`cse_machine()`** function serves as the primary driver for executing the generated control structures according to the standard 13 rules. It manages control flow, operands, environments, and current environment access via four stacks (**`control`**, **`machine_stack`**, **`stackOfEnvironment`**, and **`getCurrEnvironment`**).
    - The CSE machine supports various language features such as lambda functions, conditional expressions, tuple operations, built-in functions, operators, and environment handling.

4. Helper Functions
    - Several helper functions(**`isAlpha()`**, **`isDigit()`**, **`isBinaryOperator()`**, **`isNumber()`**, etc.) aid in token classification and processing.
    - Additional functions like **`arrangeTuple()`** and **`addSpaces()`** are utilized for managing tree nodes, particularly when dealing with tuples and escape sequences within strings.

5. Grammar Rules and Recursive Descent Parsing
    - The parser adheres to a set of grammar rules for recognizing and parsing input code. These rules are implemented using recursive descent parsing functions, where each function corresponds to a non-terminal in the grammar. They recursively call each other to handle nested language constructs such as let expressions, function definitions, conditional expressions, arithmetic operations, and more.

6.2.2 Functions in the Parser

Function	Description
parser(char read_array[], int i, int size, int af)	Constructor for the parser object that takes an array of characters (file content), starting index, size of content, and AST flag as parameters.
bool isReservedKey(string str)	Checks if a given string is a reserved keyword in the programming language.
bool isOperator(char ch)	Determines if a given character is an operator.
bool isAlpha(char ch)	Checks if a given character is an alphabetic character.
bool isDigit(char ch)	Checks if a given character is a digit.
bool isBinaryOperator(string op)	Checks if a given string is a binary operator.
bool isNumber(const std::string &s)	Determines if a given string represents a number.
void read(string val, string type)	Reads a token value and its type and writes them to an output file.
void buildTree(string val, string type, int child)	Builds a parse tree node with the specified value, type, and number of children.
token getToken(char read[])	Retrieves the next token from the input character array.
void parse()	Initiates the parsing process using the parser object.
void MST(tree *t)	Constructs a Symbol Table (ST) based on the parse tree.
tree *makeStandardTree(tree *t)	Constructs a standardized parse tree.
void buildControlStructures(tree *x*, *tree*(*controlNodeArray)[200])	Builds control structures for the parse tree.
void cse_machine(vector<vector<tree *> > &controlStructure)	Implements a CSE (Control Structure Environment) machine based on control structures.
void arrangeTuple(tree *tauNode, stack<tree *> &res)	Arranges tuples in the parse tree.
string addSpaces(string temp)	Adds spaces to a string for formatting purposes.

6.2.3 Grammar Rule procedures

Procedure	**Grammar Rule**
procedure_E()	E -> 'let' D 'in' E => 'let' -> 'fn' Vb+ '.' E => 'lambda' -> Ew;
procedure_Ew()	Ew -> T 'where' Dr => 'where' -> T;
procedure_T()	T -> Ta (',' Ta)+ => 'tau' -> Ta ;
procedure_Ta()	Ta -> Ta 'aug' Tc => 'aug' -> Tc ;
procedure_Tc()	Tc -> B '->' Tc '|' Tc => '->' -> B ;
procedure_B()	B -> B 'or' Bt => 'or' -> Bt ;
procedure_Bt()	Bt -> Bt '&' Bs => '&' -> Bs ;
procedure_Bs()	Bs -> 'not' Bp => 'not' -> Bp ;
procedure_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 ;
procedure_A()	A -> A '+' At => '+' -> A '-' At => '-' -> '+' At -> '-'At =>'neg' -> At ;
procedure_At()	At -> At '*' Af => '*' -> At '/' Af => '/' -> Af ;
procedure_Af()	Af -> Ap '**' Af => '**' -> Ap ;

procedure_Ap()	Ap -> Ap ’@’ ’<IDENTIFIER>’ R => ’@’ -> R ;
procedure_R()	R -> R Rn => ’gamma’ -> Rn ;
procedure_Rn()	Rn -> ’<IDENTIFIER>’ -> ’<INTEGER>’ -> ’<STRING>’ -> ’true’ => ’true’ -> ’false’ => ’false’ -> ’nil’ => ’nil’ -> ’(’ E ’)’ -> ’dummy’
procedure_D()	D -> Da ’within’ D => ’within’ -> Da ;
procedure_Da()	Da -> Dr (’and’ Dr)+ => ’and’ -> Dr ;
procedure_Dr()	Dr -> ’rec’ Db => ’rec’ -> Db ;
procedure_Db()	Db -> Vl ’=’ E => ’=’ -> ’<IDENTIFIER>’ Vb+ ’=’ E => ’fcn_form’ -> ’(’ D ’)’ ;
procedure_Vb()	Vb -> ’<IDENTIFIER>’ -> ’(’ Vl ’)’ -> ’(’ ’)’
procedure_Vl()	Vl -> ’<IDENTIFIER>’ list ’,’

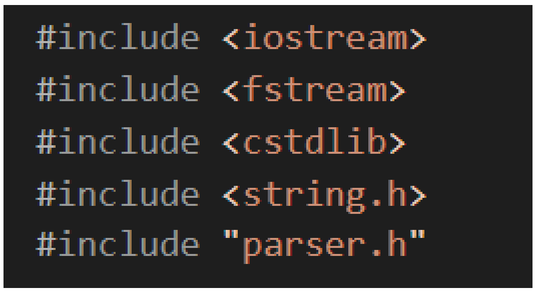
6.3 Main.cpp

6.3.1 Functionality

The main.cpp file contains the main function, which acts as the starting point of the program. It is responsible for handling command-line arguments, reading file content, and initializing the parser object defined in the **`parser.h`** file.

6.3.2 Program Structure

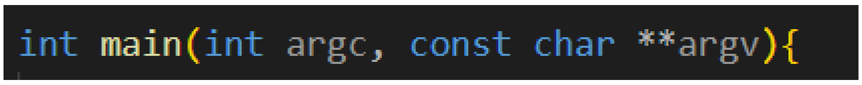
1. Include Statements



```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include "parser.h"
```

The program includes necessary header files such as iostream, fstream, cstdlib, string, and **`parser.h`** to access standard C++ functionalities and the parser implementation.

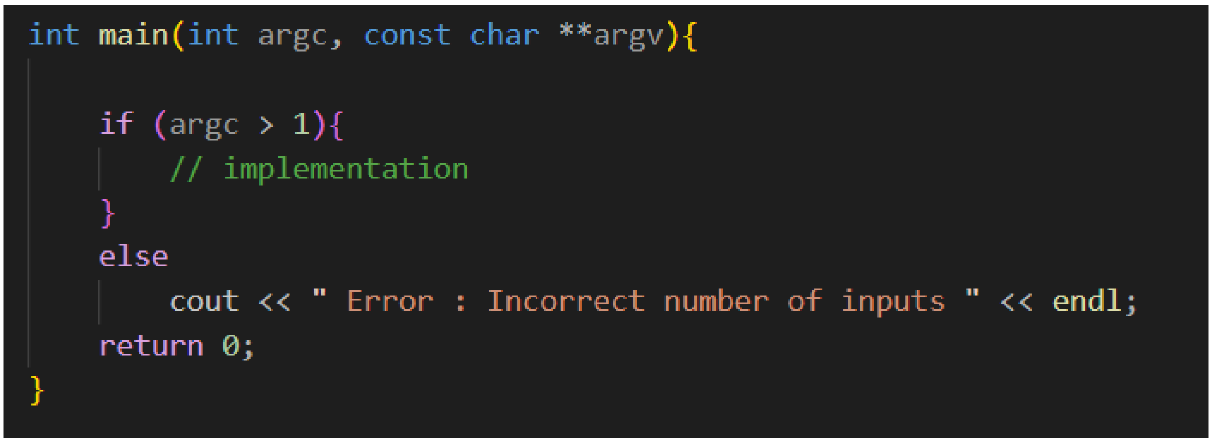
2. Main Function



```
int main(int argc, const char **argv){
```

The **`main`** function is where the program execution begins. It processes command-line arguments, reads file content, and starts the parsing process using the parser object.

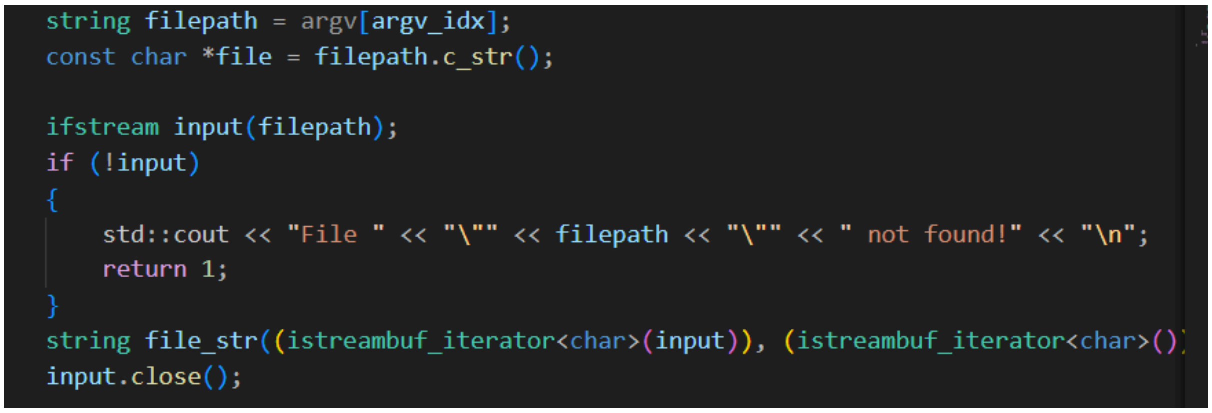
3. Parsing Command-Line Arguments



```
int main(int argc, const char **argv){

    if (argc > 1){
        // implementation
    }
    else
        cout << " Error : Incorrect number of inputs " << endl;
    return 0;
}
```

The main function checks if command-line arguments are provided to determine the filename and any additional flags like AST or ST.

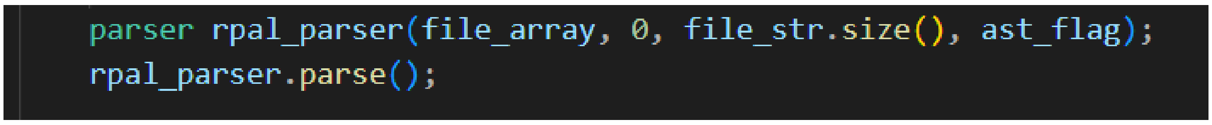
4. Reading and Processing the File



```
string filepath = argv[argv_idx];
const char *file = filepath.c_str();

ifstream input(filepath);
if (!input)
{
    std::cout << "File " << "\"" << filepath << "\"" << " not found!" << "\n";
    return 1;
}
string file_str((istreambuf_iterator<char>(input)), (istreambuf_iterator<char>()
input.close();
```

It reads the specified file's content into a string, stores it in a char array, and prepares it for parsing.

5. Creating and Initiating the Parser Object



```
parser rpal_parser(file_array, 0, file_str.size(), ast_flag);
rpal_parser.parse();
```

The main function creates an instance of the parser object and begins parsing the file content by calling the parse method on the parser object.

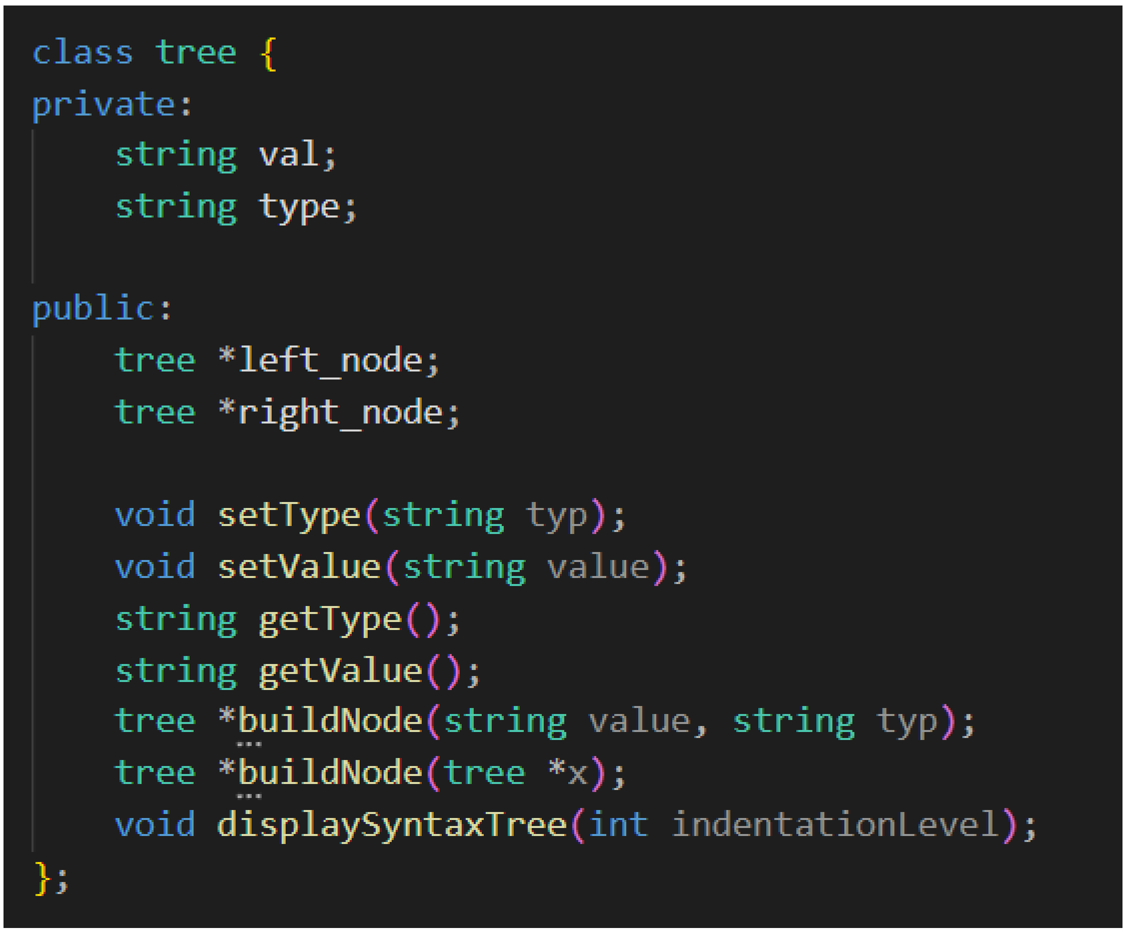
Note : The main function is crucial for driving the program, managing file input, handling command-line arguments, and initiating the parsing process using the parser object. It enables parsing files and optionally generating outputs like Abstract Syntax Trees (AST) or Symbol Tables (ST) based on user-specified flags.

6.4 tree.h

6.4.1 Introduction

The **`tree.h`** header file contains the implementation of a syntax tree node class, used as part of implementing the RPAL language compiler. This class represents individual nodes within the syntax tree, storing information such as value and type, and providing methods for creating new nodes and displaying the syntax tree.

6.4.2 Implementation



```
class tree {
private:
    string val;
    string type;

public:
    tree *left_node;
    tree *right_node;

    void setType(string typ);
    void setValue(string value);
    string getType();
    string getValue();
    tree *buildNode(string value, string typ);
    tree *buildNode(tree *x);
    void displaySyntaxTree(int indentationLevel);
};
```

The tree.h header file encapsulates the implementation of several functions as below.

1. **`setType(string typ)`**
    - Description:- Sets the type of the node to the specified value.
    - Parameters:- `typ` (string)
        - Type of the node
    - .Return Type:- Void

2. **`setType(string typ)`**
    - Description:- Sets the value of the node to the specified string.

- Parameters:- `value` (string)
    - Value of the node.
- Return Type:- Void

3. **`getType()`**
    - Description:- Retrieves the type of the node.
    - Return Type:- string

4. **`getValue()`**
    - Description:- Gets the value of the node.
    - Return Type:- string

5. **`buildNode(string value, string typ)`**
    - Description:- Creates a new node with the provided value and type, returning a pointer to the new node.
    - Parameters:- `value` (string) , `typ` (string)
        - Value of the new node , Type of the new node
    - Return Type:- tree*

6. **`buildNode(tree *x)`**
    - Description:- Creates a new node with the same value and type as the provided node `x`, returning a pointer to the new node.
    - Parameters:- `x` (tree*)
        - Pointer to the node to copy values from.
    - Return Type:- tree*

7. **`displaySyntaxTree(int indentationLevel)`**
    - Description:- Recursively displays the syntax tree starting from the current node, with the specified indentation level for each node.
    - Parameters:- `indentationLevel` (int)
        - Level of indentation for the current node.
    - Return Type: Void

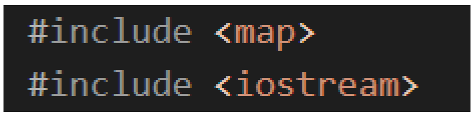
6.5 Environment.h

6.5.1 Introduction

The **`environment.h`** file encapsulates the "environment" class, implemented in C++, which serves as a representation of an environment within the context of the CSE (Control Structure Environment) machine. Below is an overview of the program structure outlined in the file.

6.5.2 Implementation

1. Header Guards

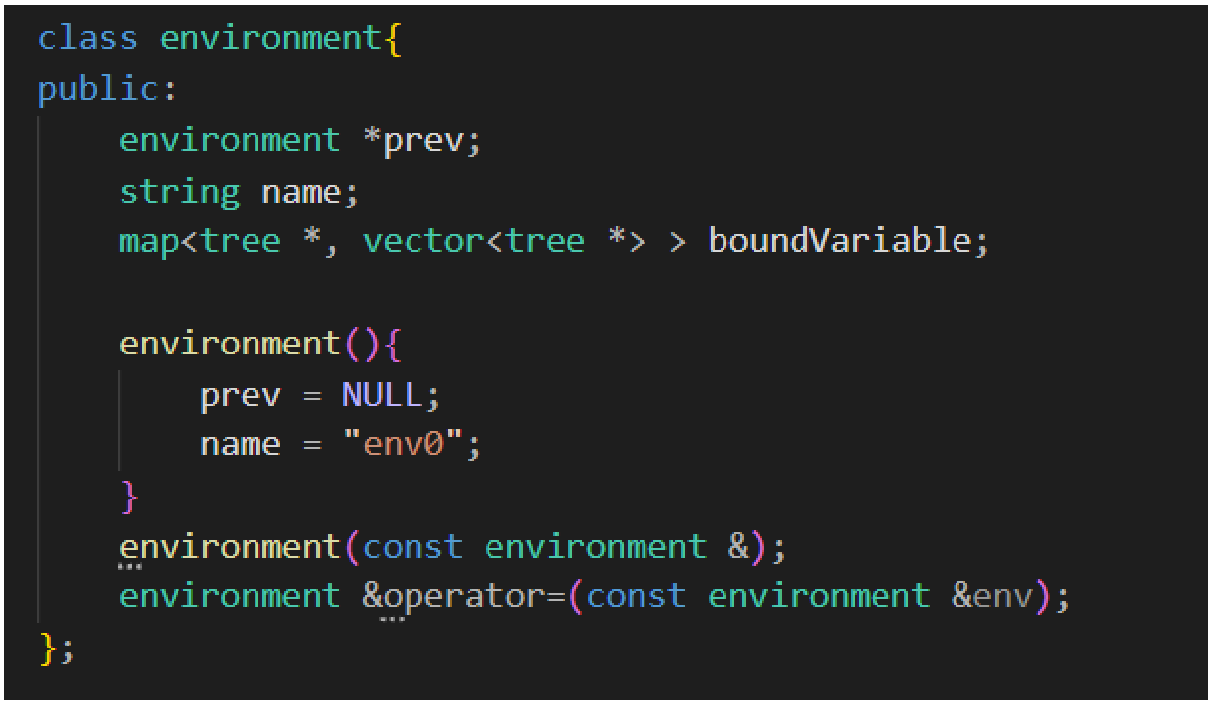


```
#include <map>
#include <iostream>
```

   - The file begins with standard header guards (`#ifndef ENVIRONMENT_H_` and `#define ENVIRONMENT_H`) to prevent multiple inclusions, ensuring there are no compilation issues due to repeated inclusion of the header file.

2. Include Statements
   - Necessary header files, such as `<map>` for using the map container and `<iostream>` for standard input/output streams, are included to enable the functionality required by the "environment" class.

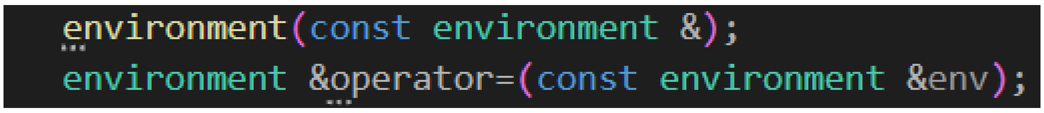
3. Class Definition



```
class environment{
public:
    environment *prev;
    string name;
    map<tree *, vector<tree *> > boundVariable;

    environment(){
        prev = NULL;
        name = "env0";
    }
    environment(const environment &);
    environment &operator=(const environment &env);
};
```

- The "environment" class is defined with public data members and a default constructor. It represents an environment in the CSE machine and includes:
- A pointer to the previous environment (`prev`).
- A string indicating the name of the environment (`name`).
- A map (`boundVar`) associating tree nodes with vectors of tree nodes, used to track variable bindings and their values within the environment.

4. Function Prototypes



```
environment(const environment &);
environment &operator=(const environment &env);
```

   - The "environment" class declares a copy constructor (**`environment(const environment &)`** and an assignment operator **`environment &operator=(const environment &env)`** outside the class definition. However, their implementations are not provided within the header file.

5. Member Function Definitions
   - The definitions of member functions for the "environment" class are not included in the header file, suggesting that they are implemented elsewhere, possibly in a corresponding "environment.cpp" file.

6. Header Guard Closure
   - The header file is closed with the standard header guard closure (`#endif`), ensuring proper encapsulation of its contents.

Note: The "environment" class encapsulates variable bindings and their values within a specific scope in the CSE machine. While the file declares function prototypes for essential operations like copying and assignment, their actual implementations are expected to be provided elsewhere in the program.

6.6 Abstract Syntax Tree (AST) Generator

6.6.1 Introduction

Abstract Syntax Tree (AST) generation is a fundamental step in the process of parsing and analyzing code. It involves creating a hierarchical representation of the syntactic structure of a program without including all the details of the original code.

6.6.2 Implementation

1. Tokenization and Parsing

   The AST generation process involves parsing the input RPAL program and constructing a hierarchical tree structure. Each node in the tree represents a syntactic element of the program, such as integers, operators, keywords, identifiers, strings, spaces, comments, punctuation and literals.

   - The process starts with tokenization, where the input code is broken down into individual tokens (e.g., identifiers, keywords, operators) using the **`get_token`** function.
   - After tokenization, the code enters the parsing phase, where it is analyzed based on predefined grammar rules and syntax.
   - In the provided code, the parsing process begins with the **`procedure_E()`** function, which likely represents the starting point of parsing based on the grammar rules.

2. Ignoring DELETE Tokens
   - The code includes logic to ignore `DELETE` tokens, which are typically used to mark elements for removal or deletion but are not relevant to the syntax analysis or AST construction.

3. AST Construction
   - During parsing, as different parts of the code are recognized and processed, tree nodes are created to represent the syntactic elements and their relationships.
   - The **`buildNode`** function is used to create tree nodes with specific values and types, such as keywords or punctuation marks.
   - Conditions and logic statements in the code determine how these tree nodes are structured and connected to form the AST.

- For example, when encountering certain keywords like "let," "and," "where," "within," "rec," "function_form," "lambda," or "@," specific actions are taken to construct corresponding nodes in the AST.

4. Handling Different Constructs
    - The code snippet includes logic to handle various language constructs such as variable assignments, function definitions, conditional expressions, and more.
    - For instance, when encountering a "let" statement followed by an assignment ("="), the AST node is transformed into a "gamma" node with a lambda function and associated variables and expressions.
    - Similarly, other constructs like "and," "where," "within," "rec," "function_form," "lambda," or "@" are processed based on their syntax and semantics to build appropriate AST nodes.

5. Standardization of the AST
    - After constructing the initial AST, the **`MST`** function (Make Standard Tree) is called to standardize the tree structure further.
    - Standardization involves transforming specific nodes or rearranging the tree to adhere to a standardized representation or specific rules.
    - The AST standardization process ensures consistency and simplifies subsequent analysis or execution steps.

6. Recursive Traversal
    - The function recursively traverses the syntax tree, examining each node and applying specific transformations based on predefined rules.

7. End of Parsing and AST Utilization
    - Once the parsing and AST generation are complete, further actions can be taken based on the AST's structure and content.
    - The code snippet includes additional steps like printing the AST, creating standardized trees (ST), building control structures, and executing them in a Control Stack Environment (CSE) machine.
    - These actions demonstrate the practical use of the AST for various purposes such as code analysis, transformation, and execution.

AST generation involves tokenizing input code, parsing it based on grammar rules, constructing a hierarchical tree representation (AST), standardizing the tree structure if needed, and utilizing the AST for further processing or execution within a programming language or compiler context.

6.7 Standardized Tree (ST)

1. Purpose of ST Converter

- The ST Converter is responsible for transforming the AST into a standardized tree format.
- Its primary purpose is to simplify subsequent processing steps like optimization and execution on a Control Stack Environment (CSE) machine.

2. Normalization and Consistency

- The ST format represents the program in a normalized manner, ensuring consistency and ease of interpretation.
- It adheres to predefined conventions and rules to achieve a standardized structure.

3. Role of makeStandardTree Function

- The **`makeStandardTree`** function in the provided code snippet serves as the ST Converter.
- It performs transformations on the syntax tree to convert it into a standardized form suitable for further processing.

4. Transformation Rules

- The ST Converter applies specific rules and transformations to restructure nodes, modify node types, and rearrange expressions within the syntax tree.
- These transformations ensure that the resulting tree follows the predefined ST conventions.

5. Contribution to Compilation Process

- The ST Converter ensures that the resulting tree is compatible with subsequent stages of the compilation process, such as generating control structures for the CSE machine.
- It simplifies the implementation of further compilation and execution phases by providing a standardized representation.

6. Efficiency and Correctness

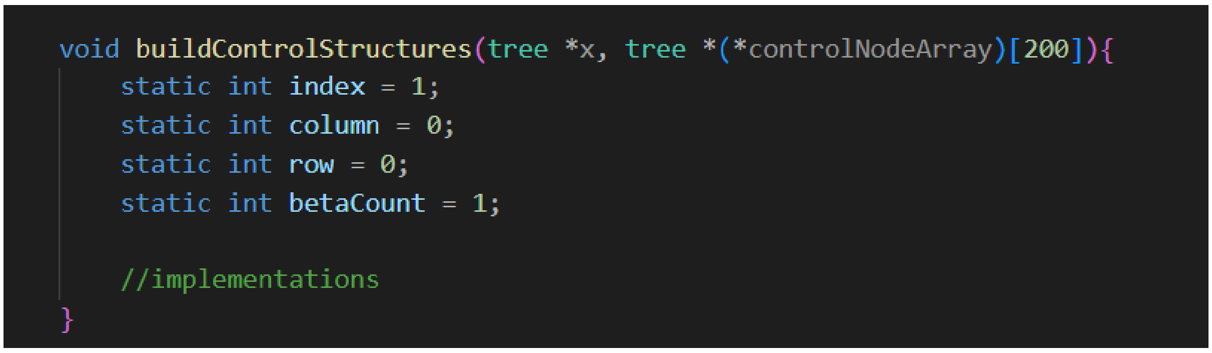
- By standardizing the tree representation, the ST Converter contributes to the overall efficiency and correctness of the RPAL compiler.
- It facilitates better optimization strategies and streamlined execution on the CSE machine.

6.8 CSE Machine

The Control Stack Environment (CSE) Machine is the execution engine for RPAL programs. It interprets RPAL code based on control structures derived from the program's abstract syntax tree (AST). These structures manage program flow, variable bindings, and function calls efficiently. In this report section, we explore the CSE Machine's implementation, starting with the construction of control structures from the syntax tree. These structures are crucial for guiding RPAL program execution.

6.8.1 Implementations

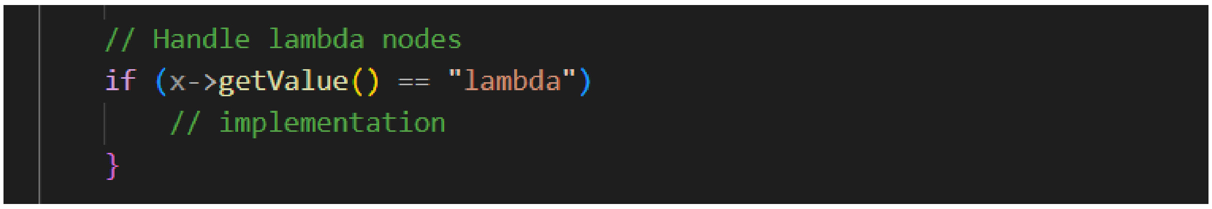
1. **buildControlStructures** Function



```
void buildControlStructures(tree *x, tree *(*controlNodeArray)[200]){
    static int index = 1;
    static int column = 0;
    static int row = 0;
    static int betaCount = 1;

    //implementations
}
```

- The **`buildControlStructures`** function plays a crucial role in constructing the control structures necessary for the CSE machine to execute RPAL programs.
- It handles different types of nodes in the syntax tree, extracting relevant information and populating the **`controlNodeArray`** accordingly.
- These control structures are vital for managing program execution flow and maintaining the state of the machine during evaluation.

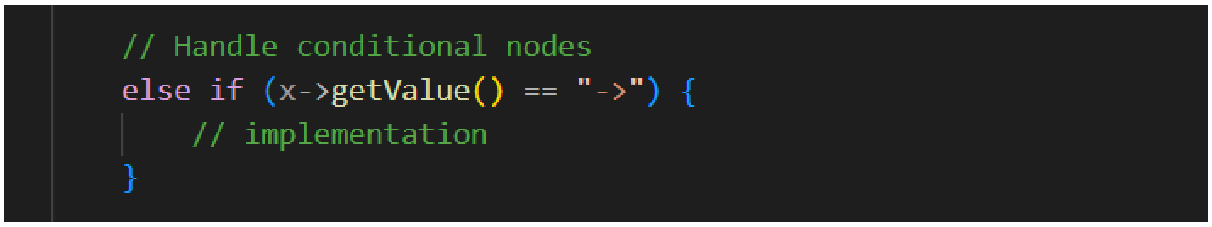
➔ Handling Lambda Nodes



```
// Handle lambda nodes
if (x->getValue() == "lambda")
    // implementation
}
```

- When encountering a lambda node in the syntax tree, the function assigns necessary information related to the lambda node to the control structure array.
- It recursively builds control structures for the lambda node's children.

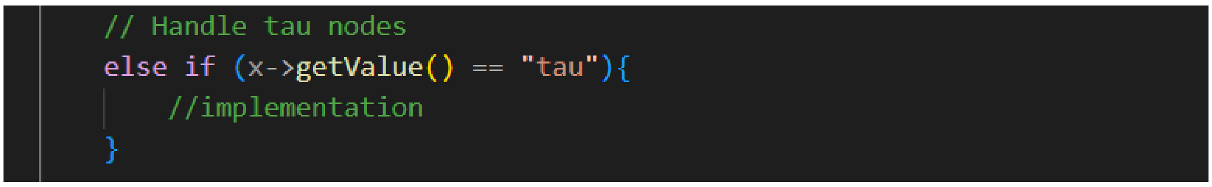
➔ Handling Conditional Nodes



```
// Handle conditional nodes
else if (x->getValue() == "->") {
    // implementation
}
```

- For nodes representing conditional expressions (denoted by '->'), the function prepares control structures.
- It populates the `controlNodeArray` with essential data such as delta numbers and beta nodes.
- The function recursively constructs control structures for the conditional node's children.
- Each step is executed to ensure proper management of conditional execution within the CSE Machine environment.

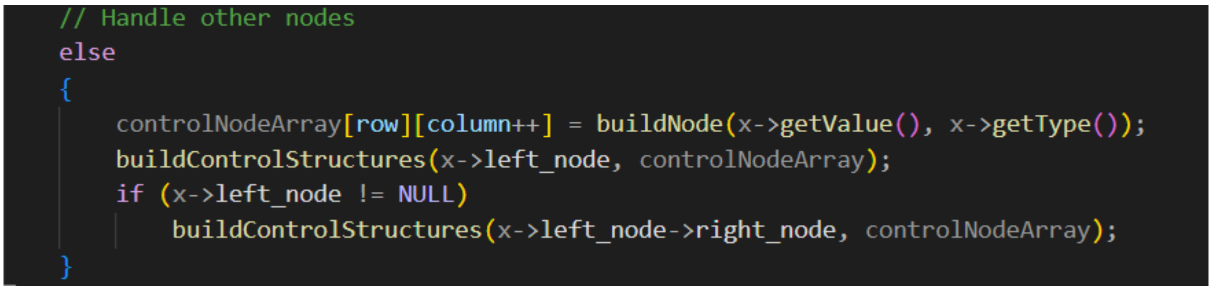
➔ Handling Tau Nodes



```
// Handle tau nodes
else if (x->getValue() == "tau"){
    //implementation
}
```

- For tau nodes, which represent function applications, the function determines the number of children and populates the control structure array accordingly.
- It then recursively builds control structures for the tau node's children.

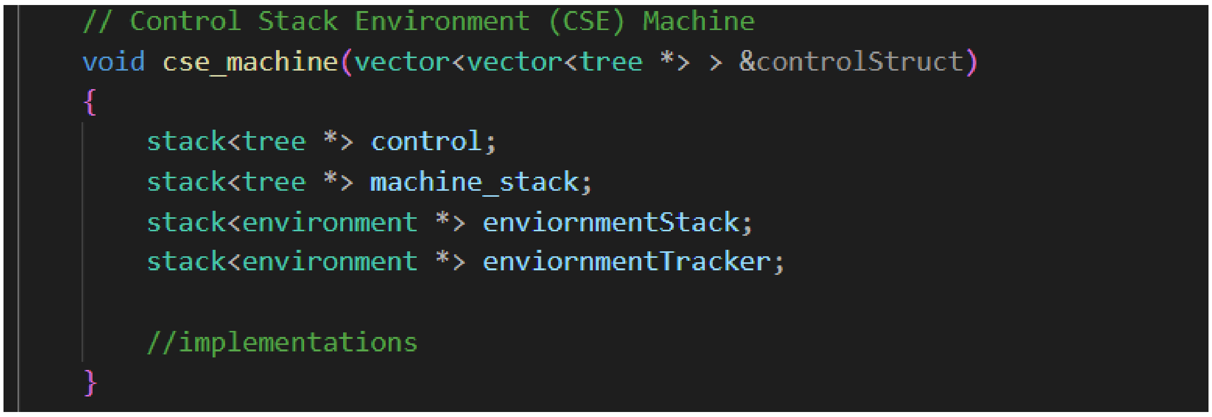
➔ Handling Other Nodes



```
// Handle other nodes
else
{
    controlNodeArray[row][column++] = buildNode(x->getValue(), x->getType());
    buildControlStructures(x->left_node, controlNodeArray);
    if (x->left_node != NULL)
        buildControlStructures(x->left_node->right_node, controlNodeArray);
}
```

- For nodes other than lambda, conditional, and tau nodes, the function populates the control structure array with the node's value and type.
- It recursively builds control structures for the node's children and their siblings.

2. **cse_machine** Function



```
// Control Stack Environment (CSE) Machine
void cse_machine(vector<vector<tree *> > &controlStruct)
{
    stack<tree *> control;
    stack<tree *> machine_stack;
    stack<environment *> enviornmentStack;
    stack<environment *> enviornmentTracker;

    //implementations
}
```

The **`cse_machine`** function serves as the backbone of the Control Stack Environment (CSE) Machine, facilitating the execution of RPAL (Recursive Porgramming: A Language) programs. Operating on a set of control structures derived from the syntax tree of the RPAL language, this function orchestrates the evaluation of expressions, manages variable scoping, and controls program flow. By utilizing stacks to maintain control structures, operands, and environments, it implements the semantics of RPAL, ensuring accurate execution of programs.

The **`cse_machine`** function aims to execute RPAL programs accurately and efficiently within the context of the CSE Machine. By coordinating control structures, managing environments, and evaluating operands, it enables the interpretation and execution of RPAL code, thereby facilitating the realization of the RPAL language's computational model.

1. Token Handling
   - Nil Token Handling

     When encountering a token with the value "nil," the function sets its type to "tau." This step aligns with RPAL semantics, where "nil" tokens represent the end of a list or tuple.

2. Operand and Special Instruction Recognition
   - ❖ The function examines each token to determine if it's an operand or a special instruction based on its value and type.
   - ❖ Operands include integers (INT), strings (STR), booleans (BOOL), and nil (NIL).
   - ❖ Special instructions encompass specific RPAL constructs like lambdas, YSTAR, Print, and various type-checking functions (e.g., Isinteger, Istruthvalue).

The **`cse_machine`** function implements several operations on tokens in the CSE (Control Stack Environment) machine. Here's a paraphrased explanation of how each function is handled within the **`cse_machine`** function:

`Isinteger`
- Checks if the token on top of the machine stack is an integer.
- If true, it pushes a boolean value (true) onto the machine stack; otherwise, it pushes false.

`Istruthvalue`
- Checks if the token on top of the machine stack is a boolean value.
- If true, it pushes true onto the machine stack; otherwise, it pushes false.

`Isstring`
- Checks if the token on top of the machine stack is a string.
- If true, it pushes true onto the machine stack; otherwise, it pushes false.

`Istuple`
- Checks if the token on top of the machine stack represents a tuple.
- If true, it pushes true onto the machine stack; otherwise, it pushes false.

`Isfunction`

- Checks if the token on top of the machine stack represents a function.
- If true, it pushes true onto the machine stack; otherwise, it pushes false.

`Isdummy`

- Checks if the token on top of the machine stack is a dummy value.
- If true, it pushes true onto the machine stack; otherwise, it pushes false.

`Lambda`

- If the token value corresponds to "lambda," the function treats it as a lambda expression.

`Stem`

- Retrieves the first element of a tuple from the machine stack and pushes it back.

`Order`

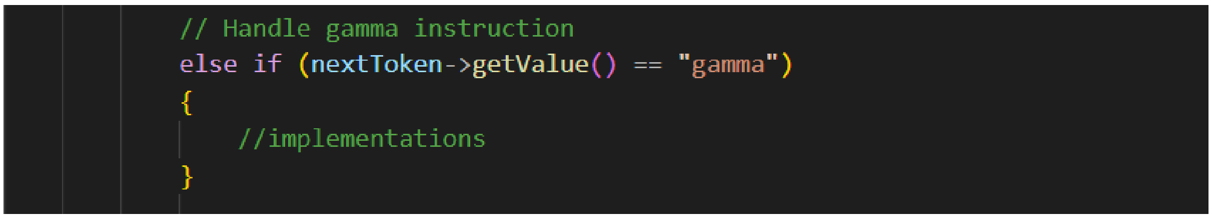
- Checks the order of a tuple on the machine stack and pushes the result (an integer) back.

`Conc (Concatenation)`

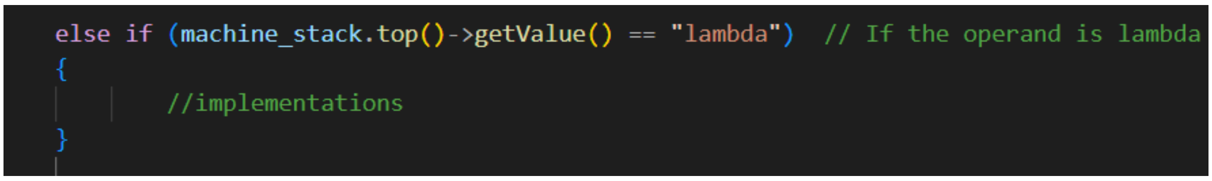
- Concatenates two tuples or strings from the machine stack and pushes the result back.

Note : The code contains the actual implementations of these functions.

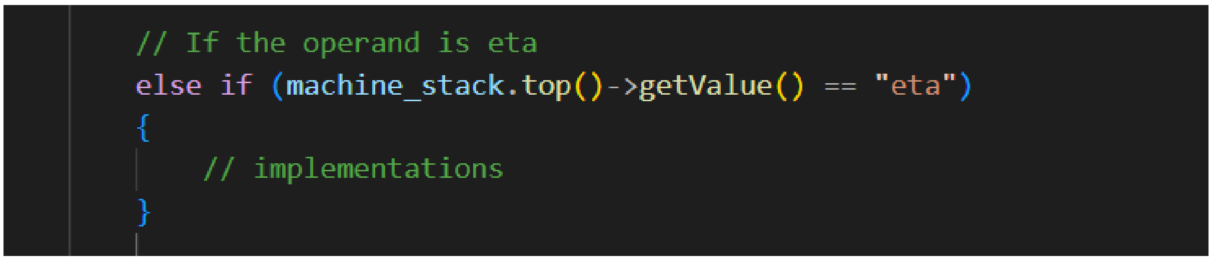
The **`cse_machine`** function aims to execute RPAL programs accurately and efficiently within the context of the CSE Machine. By coordinating control structures, managing environments, and evaluating operands, it enables the interpretation and execution of RPAL code, thereby facilitating the realization of the RPAL language's computational model.



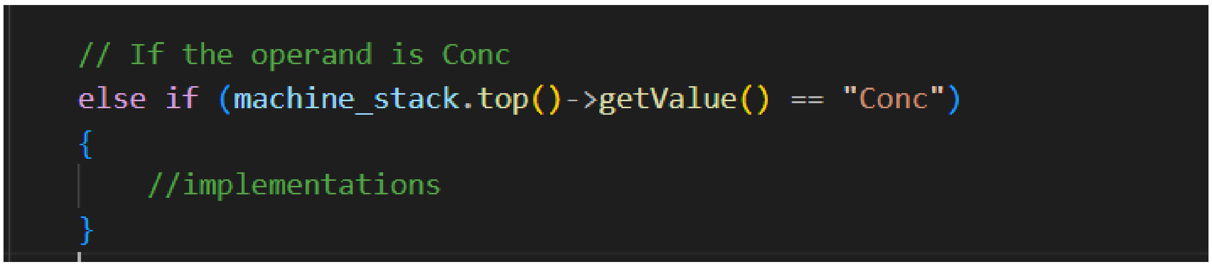
```
        // Handle gamma instruction
        else if (nextToken->getValue() == "gamma")
        {
            //implementations
        }
```



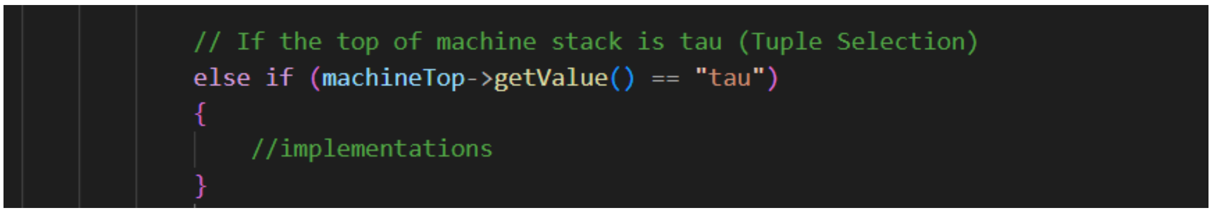
```
    else if (machine_stack.top()->getValue() == "lambda")  // If the operand is lambda
    {
            //implementations
    }
```



```
// If the operand is eta
else if (machine_stack.top()->getValue() == "eta")
{
    // implementations
}
```



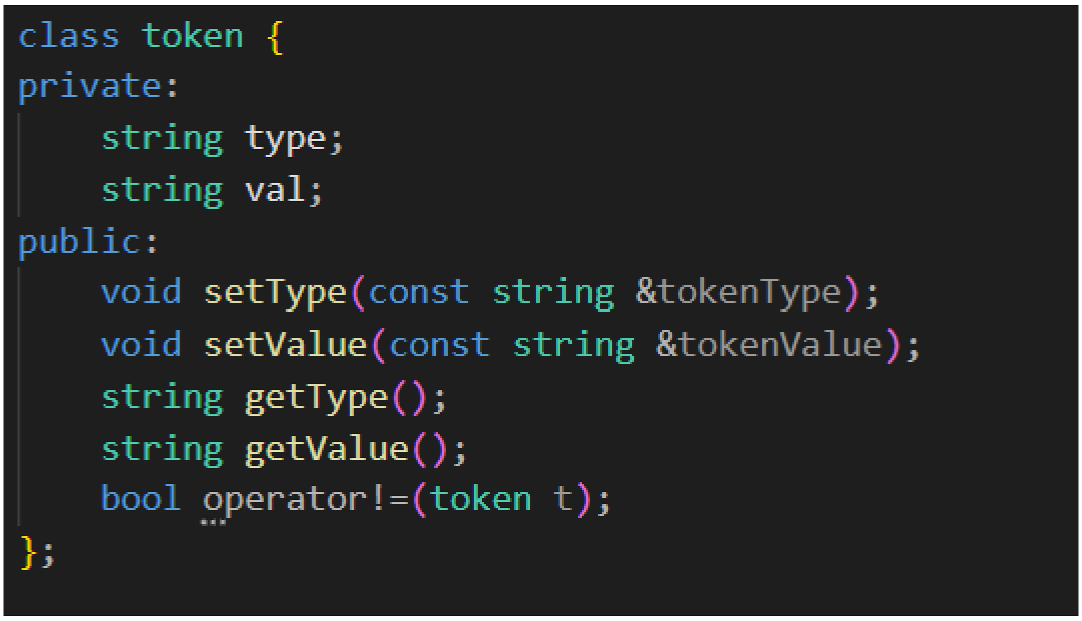
```
// If the operand is Conc
else if (machine_stack.top()->getValue() == "Conc")
{
    //implementations
}
```



```
// If the top of machine stack is tau (Tuple Selection)
else if (machineTop->getValue() == "tau")
{
    //implementations
}
```

6.9 token.h

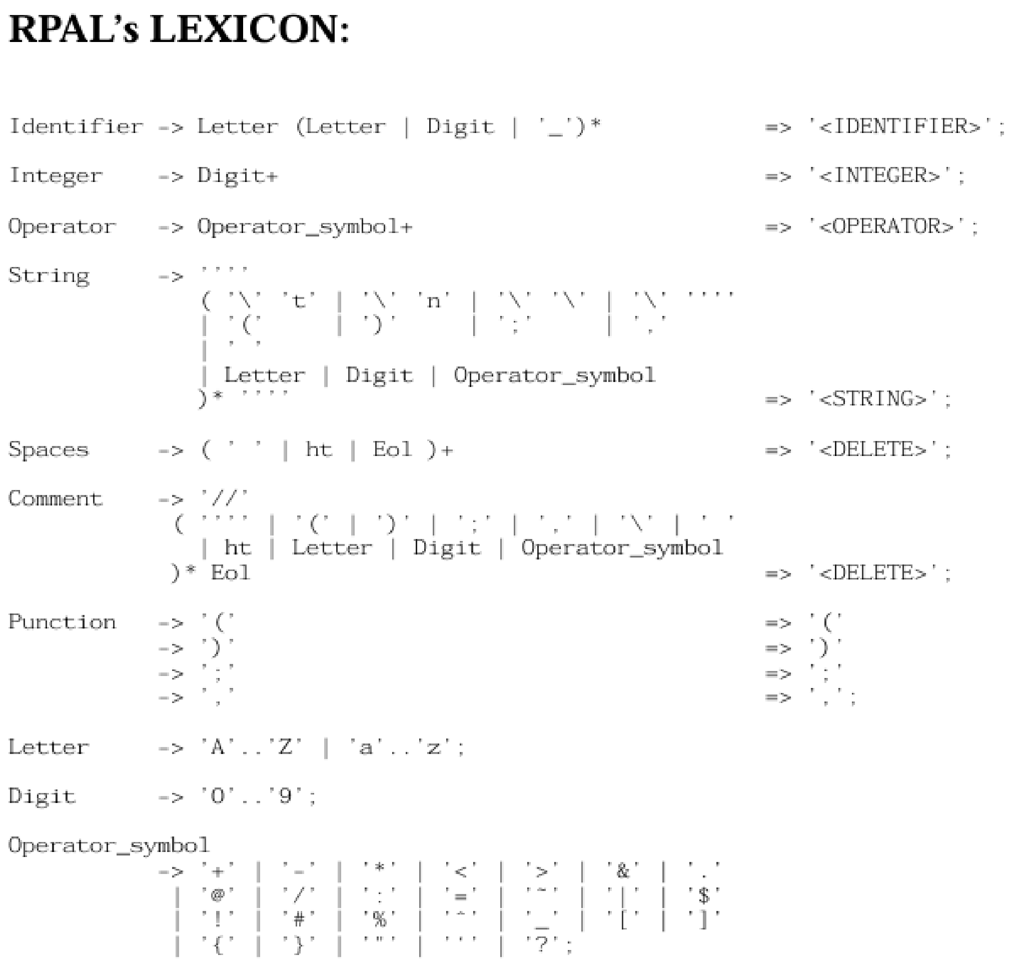
The **`token.h`** header file provides the implementation of a class called token, which represents tokens in a lexer component of the RPAL Language Compiler.



```
class token {
private:
    string type;
    string val;
public:
    void setType(const string &tokenType);
    void setValue(const string &tokenValue);
    string getType();
    string getValue();
    bool operator!=(token t);
};
```

Function	**Description**
void setType(const string &tokenType)	Sets the type of the token to the specified string value.
void setValue(const string &tokenValue)	Sets the value of the token to the specified string value.
string getType()	Retrieves the type of the token.
string getValue()	Retrieves the value of the token.
bool operator!=(token t)	Overloaded inequality operator used for comparing tokens.

References



RPAL's LEXICON:

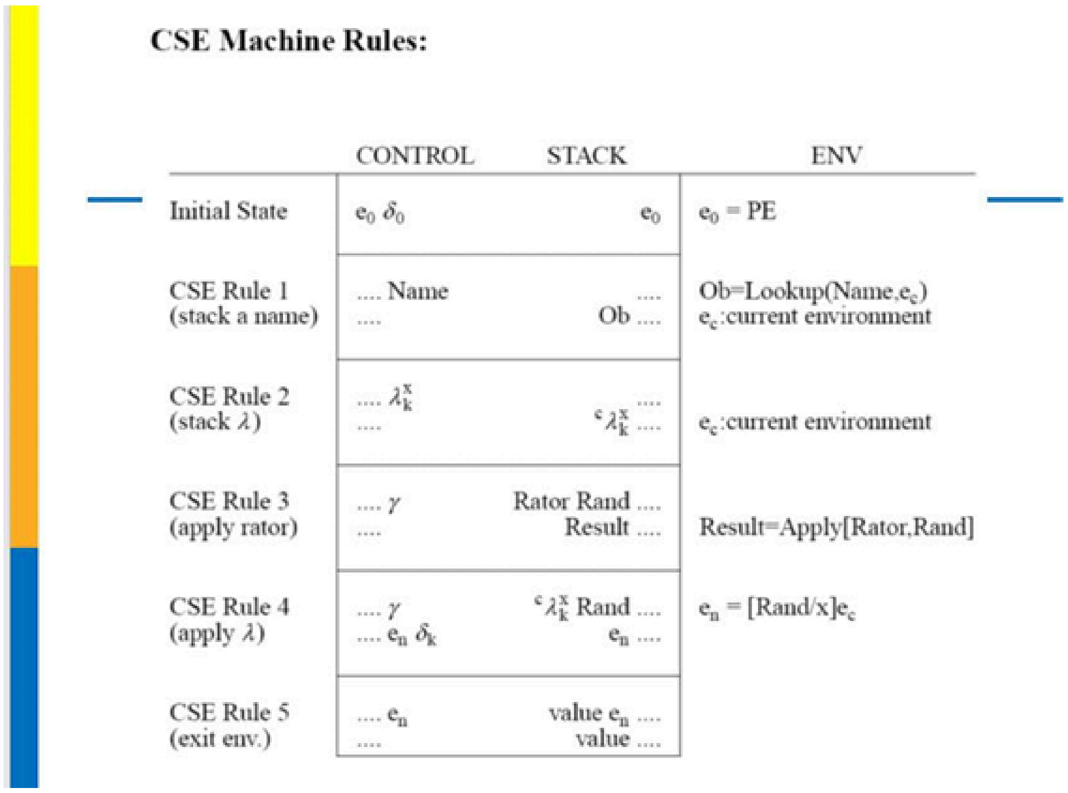
```
Identifier -> Letter (Letter | Digit | '_')*            => '<IDENTIFIER>';

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' ;
```

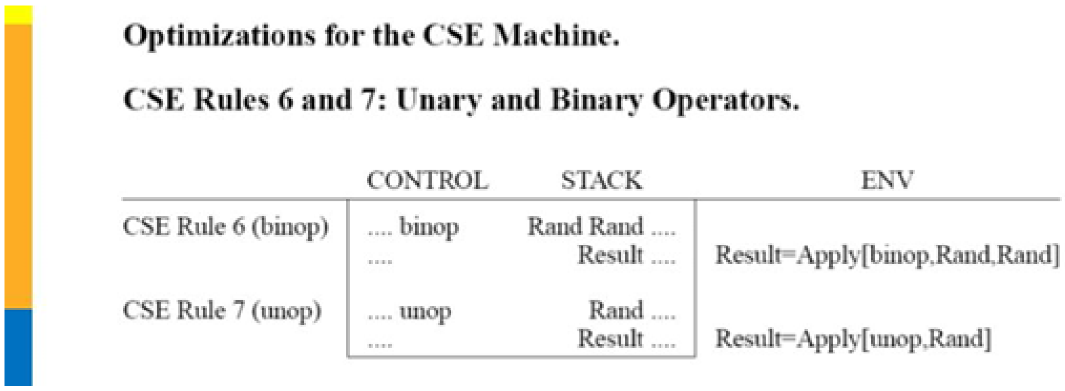
```
# Definitions ##############################################

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 ','                 => ','?;
```

CSE Rules



CSE Machine Rules:

	CONTROL	STACK	ENV
Initial State	e_0 δ_0	e_0	e_0 = PE
CSE Rule 1 (stack a name)	.... Name	.... Ob	Ob=Lookup(Name,e_c) e_c:current environment
CSE Rule 2 (stack λ)	.... λ_k^x	.... ${}^c\lambda_k^x$	e_c:current environment
CSE Rule 3 (apply rator)	.... γ	Rator Rand .... Result	Result=Apply[Rator,Rand]
CSE Rule 4 (apply λ)	.... γ .... e_n δ_k	${}^c\lambda_k^x$ Rand .... e_n	e_n = [Rand/x]e_c
CSE Rule 5 (exit env.)	.... e_n	value e_n .... value	



Optimizations for the CSE Machine.

CSE Rules 6 and 7: Unary and Binary Operators.

	CONTROL	STACK	ENV
CSE Rule 6 (binop)	.... binop	Rand Rand .... Result	Result=Apply[binop,Rand,Rand]
CSE Rule 7 (unop)	.... unop	Rand .... Result	Result=Apply[unop,Rand]

CSE Rule 8: Conditional.

	CONTROL	STACK	ENV
CSE Rule 8 (Conditional)	.... δ_{then} δ_{else} β	true	
	.... δ_{then}	....	
	.... δ_{then} δ_{else} β	false	
	.... δ_{else}	....	

CSE Rules 9 and 10: Tuples.

	CONTROL	STACK	ENV
CSE Rule 9 (tuple formation)	.... τ_n	V_1 ... V_n	
	....	$(V_1,...,V_n)$	
CSE Rule 10 (tuple selection)	.... γ	$(V_1,...,V_n)$ I	
	....	V_I	

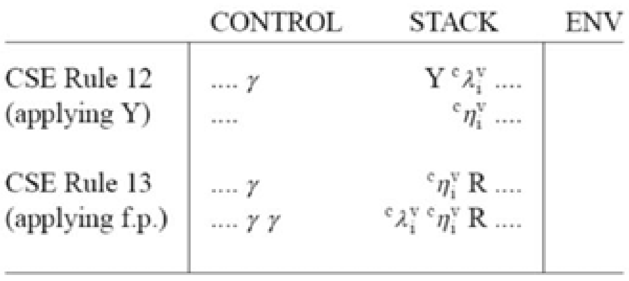
CSE Rule 11: n-ary functions.

	CONTROL	STACK	ENV
CSE Rule 11	.... γ	${}^{c}\lambda_k^{V_1,...,V_n}$ Rand	e_m=[Rand 1/V_1]...
(n-ary function)	.... e_m δ_k	e_m	[Rand n/V_n]e_c

Example:

Applicative expression	Control Structures
$(\lambda(x,y).x+y)\ (5,6)$	$\delta_0 = \gamma\ \lambda_1^{x,y}\ \tau_2\ 5\ 6$
	$\delta_1 = +\ x\ y$

RULE	CONTROL	STACK	ENV
1,1	$e_0\ \gamma\ \lambda_1^{x,y}\ \tau_2\ 5\ 6$	e_0	e_0=PE
9	$e_0\ \gamma\ \lambda_1^{x,y}\ \tau_2$	5 6 e_0	
2	$e_0\ \gamma\ \lambda_1^{x,y}$	(5,6) e_0	
11	$e_0\ \gamma$	${}^{0}\lambda_1^{x,y}$ (5,6) e_0	
1,1	$e_0\ e_1$ + x y	$e_1\ e_0$	e_1=[5/x][6/y]e_0
6	$e_0\ e_1$ +	5 6 $e_1\ e_0$	
5,5	$e_0\ e_1$	11 $e_1\ e_0$	
		11	



	CONTROL	STACK	ENV
CSE Rule 12	.... γ	Y ${}^{c}\lambda_{i}^{v}$	
(applying Y)	....	${}^{c}\eta_{i}^{v}$	
CSE Rule 13	.... γ	${}^{c}\eta_{i}^{v}$ R	
(applying f.p.)	.... γ γ	${}^{c}\lambda_{i}^{v}$ ${}^{c}\eta_{i}^{v}$ R	