

# CS 3512- Programming Languages

## Programming Project 01

Group Number – 13

210343P - Liyanage I.V.S

210536K – Rathnayaka W.T

### **Problem**

The project entailed the development of a software system capable of analyzing and parsing the RPAL programming language. This involved constructing a lexical analyzer to break down RPAL code into its fundamental components, followed by the creation of an Abstract Syntax Tree (AST) to represent the program's structure and logic. Subsequently, a parsing algorithm was implemented to transform this AST into a Standardize Tree (ST). Additionally, the system was required to execute RPAL programs using a CSE (Compiled Stack Environment) machine. The ultimate objective was to ensure that the output of our system matched that of an established RPAL interpreter, specifically "rpal.exe", when provided with identical input programs.

### **Program Execution**

To compile the program in the root directory print

**g++ -o myrpal main.cpp**

To execute the program

**./myrpal rpal\_test**

To print the AST, use the *-ast* switch.

**./myrpal -ast rpal\_test**

## **Structure of the project**

### **Main.cpp**

#### **Introduction:**

The main.cpp file serves as the entry point for the program. It handles command-line arguments, reads the content of a file, and initializes a parser object to process the file's content. The parser object's implementation resides in a separate file named "parser.h".

#### **Main Function:**

The main function is the starting point of the program. It orchestrates the parsing process and file handling.

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

#### **Creating and Initiating the Parser Object:**

Main function creates a parser object and initiates the parsing process by calling the "parse" method.

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

### **Parser.h**

#### **Introduction:**

The parser.h file houses the implementation of a Recursive Descent Parser. This parser is engineered to tokenize, parse, and transform input code into a standardized tree (ST) format, priming it for subsequent execution. It adheres to a predefined set of grammar rules to recognize and delineate the syntax and structure of the supported programming language.

- Abstract Syntax Tree (AST) and Standardized Tree (ST):

The AST is constructed utilizing the `buildTree()` function, which generates tree nodes based on token properties and adds them to the syntax tree stack (`st`). The AST serves to represent the syntactic structure of the input code.

Subsequently, the `makeST()` function is employed to convert the AST into a standardized tree (ST). This transformation involves applying operations to the AST to standardize its representation, ensuring uniformity in tree structure and preparing the code for further processing.

- Tokenization:

The parser commences by tokenizing the input code via the `getToken()` function. This function reads individual characters and classifies them into various token types, including identifiers, keywords, operators, integers, strings, punctuation, comments, spaces, and unidentified tokens. Tokenization lays the groundwork for subsequent parsing stages.

- Helper Functions:

The parser incorporates several auxiliary functions such as `isAlpha()`, `isDigit()`, `isBinaryOperator()`, and `isNumber()`, dedicated to token classification.

Additionally, `arrangeTuple()` and `addSpaces()` functions are utilized for processing and organizing tree nodes, particularly in the context of handling tuples and escape sequences in strings.

- Grammar Rules and Recursive Descent Parsing:

The parser operates in accordance with a set of grammar rules to parse the input code. These rules are implemented using recursive descent parsing functions, with each function corresponding to a non-terminal in the grammar. Recursive calls among these functions facilitate the handling of nested structures.

The grammar rules span diverse language constructs such as `let` expressions, function definitions, conditional expressions, arithmetic expressions, and more. Detailed grammar rules pertinent to this project are provided in the appendix.

- Grammar Rule Procedures: The following are the functions for grammar rules of RPAL coded as procedures,

```
void procedure_E()    void procedure_Ew()    void procedure_T()
void procedure-Ta()   void procedure_Tc()    void procedure_B()
void procedure_Bt()   void procedure_Bs()    void procedure_Bp()
void procedure_A()    void procedure_At()    void procedure_Af()
void procedure_Ap()   void procedure_R()      void procedure_Rn()
void procedure_D()    void procedure_Dr()    void procedure_Db()
void procedure_Vb()   void procedure_Vl()
```

## Tree.h

### Introduction:

The "tree" class presented in this report is a C++ implementation of a syntax tree, a fundamental data structure widely employed in programming languages to represent the syntactic structure of source code. This overview delineates the "tree" class, elucidating its function prototypes and the overarching structure of the program.

- Header Guards:

```
#ifndef TREE_H_
#define TREE_H_
```

Header guards, as exemplified above, serve to prevent multiple inclusions of the "tree.h" file within the same translation unit. By circumventing redundant inclusions, potential compilation errors are averted.

- Class Definition:

```
class tree {
private:
    std::string val;
    std::string type;
public:
    tree *left;
    tree *right;
};
```

The "tree" class, encapsulated within the provided class definition, comprises private data members and public member functions. Each instance of the "tree" class represents a node within the syntax tree. The private data members include val for storing the node's value and type for denoting its type. Publicly accessible member variables left and right are pointers to the left and right child nodes, respectively.

- Function Prototypes

```
void setType(string typ);
string getType();
void setVal(string value);
string getVal();
tree *createNode(string value, string typ);
void print_tree(int no_of_dots);
tree *createNode(tree *x);|
```

## Token.h

### Introduction:

The "token" class presented herein is a C++ implementation aimed at representing basic tokens. Tokens serve as fundamental units in programming languages, facilitating the breakdown of source code into meaningful components. This overview elucidates the "token" class, encompassing its function prototypes and the overall structure of the program.

- Class Definition:

The "token" class is defined with private data members and public member functions. The class represents a single token in the programming language.

```
class token {
private:
    string type;
    string val;
};|
```

- Function Prototypes:

The class "token" has several member functions that are defined outside the class definition. The function prototypes present in the class are:

```
void setType(const string &sts);
void setVal(const string &str);
string getType();
string getVal();
bool operator!=(token t);|
```

## Environment.h

### Introduction:

The "environment" class constitutes a C++ implementation designed to represent an environment within the context of the Control Stack Environment (CSE) machine. Environments serve the crucial role of managing variable bindings and their corresponding values within a defined scope. This overview aims to illuminate the "environment" class, encompassing its function prototypes and delineating the program's overarching structure.

- Class Definition:

The "environment" class is defined with public data members and a default constructor. The class represents an environment in the CSE machine.

```
class environment {
public:
    environment *prev;
    string name;
    map<tree *, vector<tree *> > boundVar;
    environment() {
        prev = NULL; name = "env0";
    }
};
```

- Function Prototypes:

The "environment" class lacks any function prototypes declared within its class definition. However, there exists a copy constructor and an assignment operator declared outside the class.

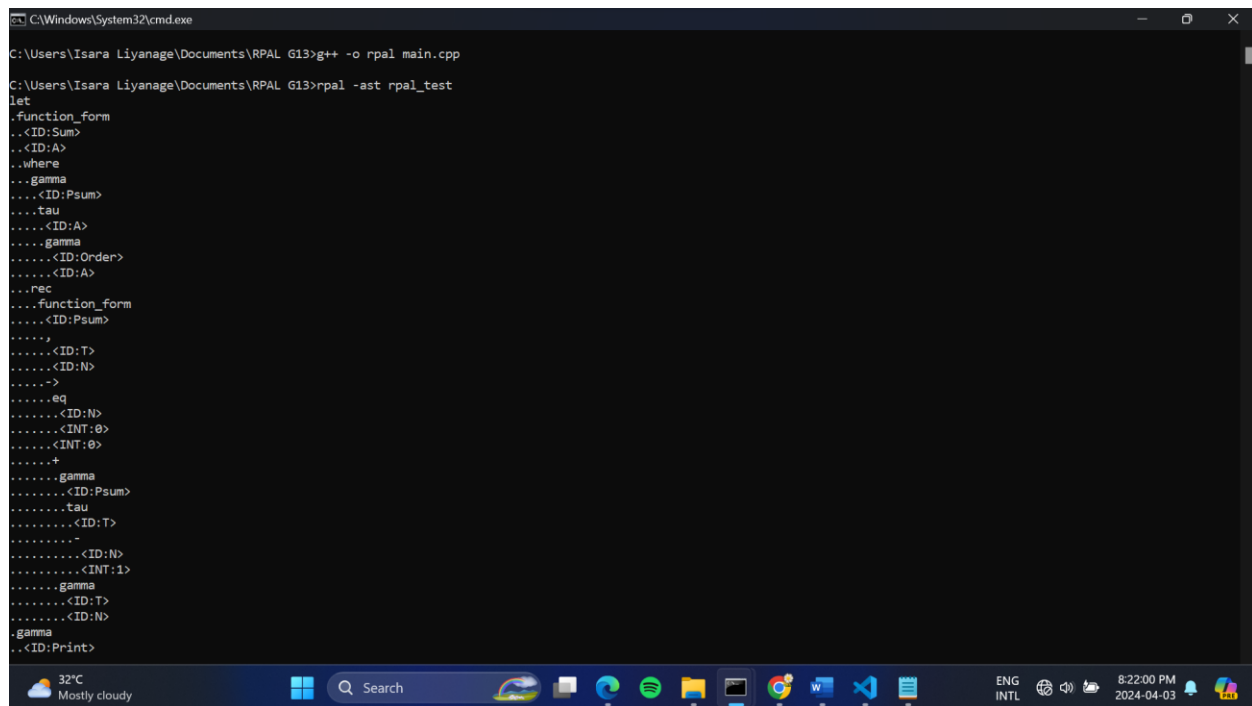
```
environment(const environment &);
environment &operator=(const environment &env);|
```

## Example run of the program.

### Input

```
let Sum(A) = Psum (A,Order A )
              where rec Psum (T,N) = N eq 0 -> 0
                  | Psum(T,N-1)+T N
in Print ( Sum (1,2,3,4,5) )
```

### Output



```
C:\Windows\System32\cmd.exe
C:\Users\Isara Liyanage\Documents\RPAL G13>g++ -o rpal main.cpp
C:\Users\Isara Liyanage\Documents\RPAL G13>rpal -ast rpal_test
let
.function_form
..<ID:Sum>
..<ID:A>
..where
...gamma
....<ID:Psum>
....tau
.....<ID:A>
.....gamma
.....<ID:Order>
.....<ID:A>
...rec
....function_form
.....<ID:Psum>
.....
.....<ID:T>
.....<ID:N>
.....>
.....eq
.....<ID:N>
.....<INT:0>
.....<INT:0>
.....+
.....gamma
.....<ID:Psum>
.....tau
.....<ID:T>
.....
.....<ID:N>
.....<INT:1>
.....gamma
.....<ID:T>
.....<ID:N>
.gamma
..<ID:Print>
```