CS 3513 - Programming Language
Programming Project 01

Group Number: 14

Group Members:

Kopimenan L. - 210296X
Ahamed M.N.Z. - 210023K

Problem Description:

The project requirement was to implement a lexical analyzer and a parser for the RPAL language. The output of the parser should be the Abstract Syntax Tree (AST) for the given input program. Then an algorithm must be implemented to convert the Abstract Syntax Tree (AST) into a Standardised Tree (ST) and the CSE machine should be implemented. The program should be able to read an input file that contains an RPAL program. The output of the program should match the output of “rpal.exe“ for the relevant program.

Program Execution Instructions:

Below is a sequence of commands that can be executed in the root directory of the project to compile the program and run RPAL programs:

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

If make does not work, try the below command

```
> g++ main.cpp -o myrpal
```

The program has also been tested to run using the following sequence of commands:

```
> make
> ./myrpal rpal_test_programs/rpal_01
```

The input file contains all the functions, including Print, Stem, Stern, Order, and Conc, with the assumption that the first letter of each function is capitalised.

Project Structure:

The project was developed exclusively in C++. It comprises five primarily files, namely:

1. main.cpp
2. interpreter.h
3. environment.h
4. tree.h
5. token.h

This document provides an overview of each file's purpose and presents the function prototypes along with their respective uses.

1. main.cpp

Introduction

The main.cpp file serves as the entry point for the program execution. It handles command-line arguments, reads the content of a file, and initialises the parsing process through a parser object. The interpretation/parsing logic is encapsulated within a separate header file named "interpreter.h".

Structure of the Program

main.cpp constitutes a simple C++ program comprising a single main function, which acts as the starting point of the application.

Include Statements:

The file includes the following header files:

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

#include "interpreter.h"
```

These headers enable access to various functionalities of the C++ standard library and include the "interpreter.h" file for parser implementation.

Main Function:

The main function is the program's entry point and is responsible for parsing command-line arguments, reading file content, and initiating the parsing process.

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

Parsing Command-Line Arguments:

The main function checks for the presence of command-line arguments to determine the filename and the presence of AST or ST flags.

```
if (argc > 1)
       {//Parsing command line arguments]
else
   { // Error message for incorrect number of inputs
       cout << " Error : Incorrect no. of inputs " << endl;
       cout << " This is the correct argument format: " << endl;
       cout << " .\\myrpal.exe <ast/st switch>(optional) <rpal file path> " <<endl; }
```

Reading and Processing the File:

main.cpp reads the specified file's content provided via command-line argument and stores it in a string.

```
// Reading file name from command line
string filepath = argv[argv_indx];


// Convert string to char array
const char *file = filepath.c_str();

// Create input stream object and read file into string
ifstream input(filepath);
if (!input)
{
       // Error message for file not found
       std::cout << "File " << "\"" << filepath << "\"" << " not found!" << "\n";
       return 1;
}

// Read file into string
string file_str((istreambuf_iterator<char>(input)),(istreambuf_iterator<char>()));
input.close();

// Convert string to char array to pass to parser
char file_array[file_str.size()];
for (int i = 0; i < file_str.size(); i++)
      file_array[i] = file_str[i];
```

Creating and Initiating the Parser Object:

main.cpp creates a parser object and initialises the parsing process by calling the "parse" method on the parser object.

```
// Create parser object and parse the file to the interpreter
parser rpal_parser(file_array, 0, file_str.size(), ast_flag);
rpal_parser.parse();
```

The parser object is instantiated with parameters such as the char array containing the file content, the starting index (0), the content's size, and the AST or ST flag.

Conclusion:

The main function plays a pivotal role in the program, responsible for reading file content, processing command-line arguments, and initiating the parsing process. It leverages the parser object for parsing file content. This file acts as the primary driver for the parser, facilitating file parsing and optionally printing the Abstract Syntax Tree (AST) or Symbol Table (ST) based on command-line flags.

2. interpreter.h

Introduction:

The parser.h file encompasses the implementation of a Recursive Descent Parser. This parser is engineered to tokenize, parse, and transform input code into a standardised tree (ST) format, facilitating further execution. It adheres to a predefined set of grammar rules to identify the syntax and structure of the programming language it interprets.

Program Structure:

1. Tokenization:

The parsing process begins with tokenization, executed by the `‘getToken()’` function. This function processes characters individually, categorising them into various token types such as identifiers, keywords, operators, integers, strings, punctuation, comments, spaces, and unknown tokens. This initial tokenization phase lays the foundation for subsequent parsing steps.

2. Abstract Syntax Tree (AST) and Standardised Tree (ST):

The AST construction is facilitated by the `‘buildTree()’` function. It constructs tree nodes based on token properties and appends them to the syntax tree stack, representing the syntactic structure of the input code.

The conversion of the AST into a standardised tree (ST) is performed by the `‘makeST()’` function. This process involves applying transformations to standardise the AST representation, ensuring consistency in tree structure and preparing the code for further execution.

3. Control Structures Execution:

Following the construction of the standardised tree, control structures are generated using the `'createControlStructures()'` function. These structures encapsulate the instructions necessary for executing the code within the Control Stack Environment (CSE) machine, 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. It adheres to 13 standard rules and utilises four stacks `('control', 'm_stack', 'stackOfEnvironment', and 'getCurrEnvironment')` to manage control flow, operands, environments, and current environment access, respectively. The CSE machine boasts support for lambda functions, conditional expressions, tuple creation and augmentation, built-in functions, unary and binary operators, environment management, and functional programming.

4. Helper Functions:

The parser incorporates several helper functions such as `'isAlpha()', 'isDigit()', 'isBinaryOperator()', and 'isNumber()'` for token classification. Additionally, functions like `'arrangeTuple()' and 'addSpaces()'` aid in processing and arranging tree nodes, particularly for handling tuples and escape sequences in strings.

5. Grammar Rules and Recursive Descent Parsing:

The parser adheres to a set of grammar rules to recognize and parse input code. These rules are implemented as recursive descent parsing functions, with each function corresponding to a non-terminal in the grammar. They recursively call each other to handle nested structures. Grammar rules encompass various language constructs such as let expressions, function definitions, conditional expressions, and arithmetic expressions. The grammar rules implemented in this project will be provided in the appendix.

6. Functions in the Parser:

The parser class contains numerous functions responsible for tokenization, parsing, AST construction, ST transformation, control structure generation, and CSE machine execution. Additionally, procedures corresponding to RPAL grammar rules are defined to handle specific language constructs.

```
parser(char read_array[], int i, int size, int af)
bool isReservedKey(string str)
bool isOperator(char ch)
bool isAlpha(char ch)
bool isDigit(char ch)
bool isBinaryOperator(string op)
bool isNumber(const std::string &s)
void read(string val, string type)
void buildTree(string val, string type, int child)
void parse()
void makeST(tree *t)
void createControlStructures(tree *x, tree *(*setOfControlStruct)[200])
void cse_machine(vector<vector<tree *>> &controlStructure)
void arrangeTuple(tree *tauNode, stack<tree *> &res)
token getToken(char read[])
tree *makeStandardTree(tree *t)
tree *(*setOfControlStruct)
string addSpaces(string temp_str)
```

7. Grammar Rule Procedures:

The parser includes procedures for grammar rules of RPAL, coded as functions. Each function corresponds to a specific grammar rule, facilitating the parsing process. The corresponding grammar productions are provided in the appendix.

```
void E()                        void At()

void Ew()                       void Af()

void T()                        void Ap()

void Ta()                       void R()

void Tc()                       void Rn()

void B()                        void D()

void Bt()                       void Dr()

void Bs()                       void Db()

void Bp()                       void Vb()

void A()                        void Vl()
```

Conclusion:

`'interpreter.h'` encompasses a comprehensive Recursive Descent Parser implementation capable of tokenizing, parsing, and converting input code into a standardised tree representation. The parser adheres to grammar rules and employs recursive descent parsing to handle a wide range of programming language constructs. Integration with the Control Stack Environment (CSE) machine facilitates code execution based on semantic meaning.

3. tree.h

Introduction:

The "tree" class serves as a C++ implementation of a syntax tree, a fundamental data structure used in programming languages to represent the syntactic structure of source code. This report offers an overview of the "tree" class, detailing its function prototypes and outlining the program's overall structure.

Structure of the Program:

1. Header Guards:

To prevent multiple inclusions of the "tree.h" file in the same translation unit and avoid potential compilation errors, header guards are employed:

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

2. Include Statements:

The program includes essential header files for standard input/output streams and the C++ stack data structure:

```
#include <iostream>
#include <stack>
```

3. Class Definition:

The "tree" class is defined with private data members representing the value and type of a node, along with left and right child pointers:

```
class tree // Tree class to store the AST nodes
{
private:
   string val;  // Value of node
   string type; // Type of node

public:
   tree *left;
   tree *right;
   void setType(string typ);
   void setValue(string value);
   string getType();
   string getValue();
   tree *createNode(string value, string typ);
   tree *createNode(tree *x);
   void print_tree(int dots_count);
};
```

4. Function Prototypes:

The "tree" class declares several member functions that are defined outside the class. The function prototypes include methods for setting and getting node values and types, creating nodes, and printing the tree structure:

```
   void setType(string typ);
   void setValue(string value);
   string getType();
   string getValue();
   tree *createNode(string value, string typ);
   tree *createNode(tree *x);
   void print_tree(int dots_count);
```

5. Member Function Definitions:

Member functions are defined outside the class using the "tree::" scope resolution operator:

```
void tree::setType(string typ)
void tree::setValue(string value)
```

6. Function Definitions:

Additional functions such as "createNode" and "print_tree" are defined outside the class as standalone functions:

```
tree *createNode(string value, string typ)
tree *createNode(tree *x)
void tree::print_tree(int dots_count) // Function to print the AST/ST
```

7. Header Guard Closure:

Finally, the "tree.h" file is closed with the header guard closure:

```
#endif
```

Conclusion:

The "tree" class provides a representation of a syntax tree, offering member functions for node manipulation and tree structure printing. The function prototypes outlined in this report enable node creation, value and type manipulation, and informative visualisation of the syntax tree.

4. token.h

Introduction:

The "token" class serves as a C++ implementation of a basic token representation, which are fundamental units in programming languages used to break down source code into meaningful components. This report offers an overview of the "token" class, detailing its function prototypes and outlining the program's overall structure.

Structure of the Program:

1. Header Guards:

To prevent multiple inclusions of the "token.h" file in the same translation unit and avoid potential compilation errors, header guards are employed:

```
#ifndef TOKEN_H_
#define TOKEN_H_
```

2. Include Statements:

The program includes an essential header file for standard input/output streams:

```
#include <iostream>
```

3. Class Definition:

The "token" class is defined with private data members representing the type and value of a token:

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

public:
   void setType(const string &sts);
   void setValue(const string &str);
   string getType();
   string getValue();
   bool operator!=(token t); // Overloaded operator to compare two tokens
};
```

4. Function Prototypes:

The "token" class declares several member functions that are defined outside the class. The function prototypes include methods for setting and getting the type and value of a token, along with overloading the inequality operator for token comparison:

```
    void setType(const string &sts);
    void setValue(const string &str);
    string getType();
    string getValue();
    bool operator!=(token t); // Overloaded operator to compare two tokens
```

5. Member Function Definitions:

Member functions are defined outside the class using the "token::" scope resolution operator:

```
void token::setType(const string &str
void token::setValue(const string &str)
```

6. Operator Overload Definition:

The "operator!=" function, used for token comparison, is defined outside the class as a standalone function:

```
    bool token::operator!=(token t);
```

7. Header Guard Closure:

Finally, the "token.h" file is closed with the header guard closure:

```
#endif
```

Conclusion:

The "token" class offers a straightforward representation of tokens used in programming languages. It facilitates setting and getting the type and value of a token and overloads the inequality operator for token comparison. By utilising this class, operations related to tokenization and syntax analysis can be performed effectively.

5. token.h

Introduction:

The "environment" class constitutes a C++ implementation of an environment, which serves a crucial role in the Control Stack Environment (CSE) machine by tracking variable bindings and their values within a particular scope. This report offers an overview of the "environment" class, outlining its function prototypes and providing insights into the program's overall structure.

Structure of the Program:

1. Header Guards:

To prevent multiple inclusions of the `"environment.h"` file in the same translation unit and avoid potential compilation errors, header guards are utilised:

```
#ifndef ENVIRONMENT_H_
#define ENVIRONMENT_H_
```

2. Include Statements:
The program includes necessary header files for utilising the C++ map container and standard input/output streams:

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

3. Class Definition:

The "environment" class is defined with public data members representing the previous environment pointer, the name of the environment, and a map to associate bound variables with their values.

Additionally, a default constructor is provided:

```
class environment // Environment class to store the environment objects and their bindings
{
public:
   environment *prev;
   string name;
    map<tree *, vector<tree *> > boundVar; // Map to store the bindings of variables to
environment objects

   environment()
   { // Constructor to initialize the environment object
       prev = NULL;
       name = "env0";
   }
};
```

4. Function Prototypes:

Although the "environment" class does not declare any function prototypes within its definition, a copy constructor and an assignment operator are declared outside the class:

```
environment(const environment &); // Copy constructor to copy the environment object

environment &operator=(const environment &env); // Assignment operator to assign the
environment object
```

5. Member Function Definitions:

The member functions of the "environment" class are not declared within the class definition, and thus, their definitions are not provided in the header file.

6. Header Guard Closure:

Finally, the `"environment.h"` file is closed with the header guard closure:

```
#endif
```

Conclusion:

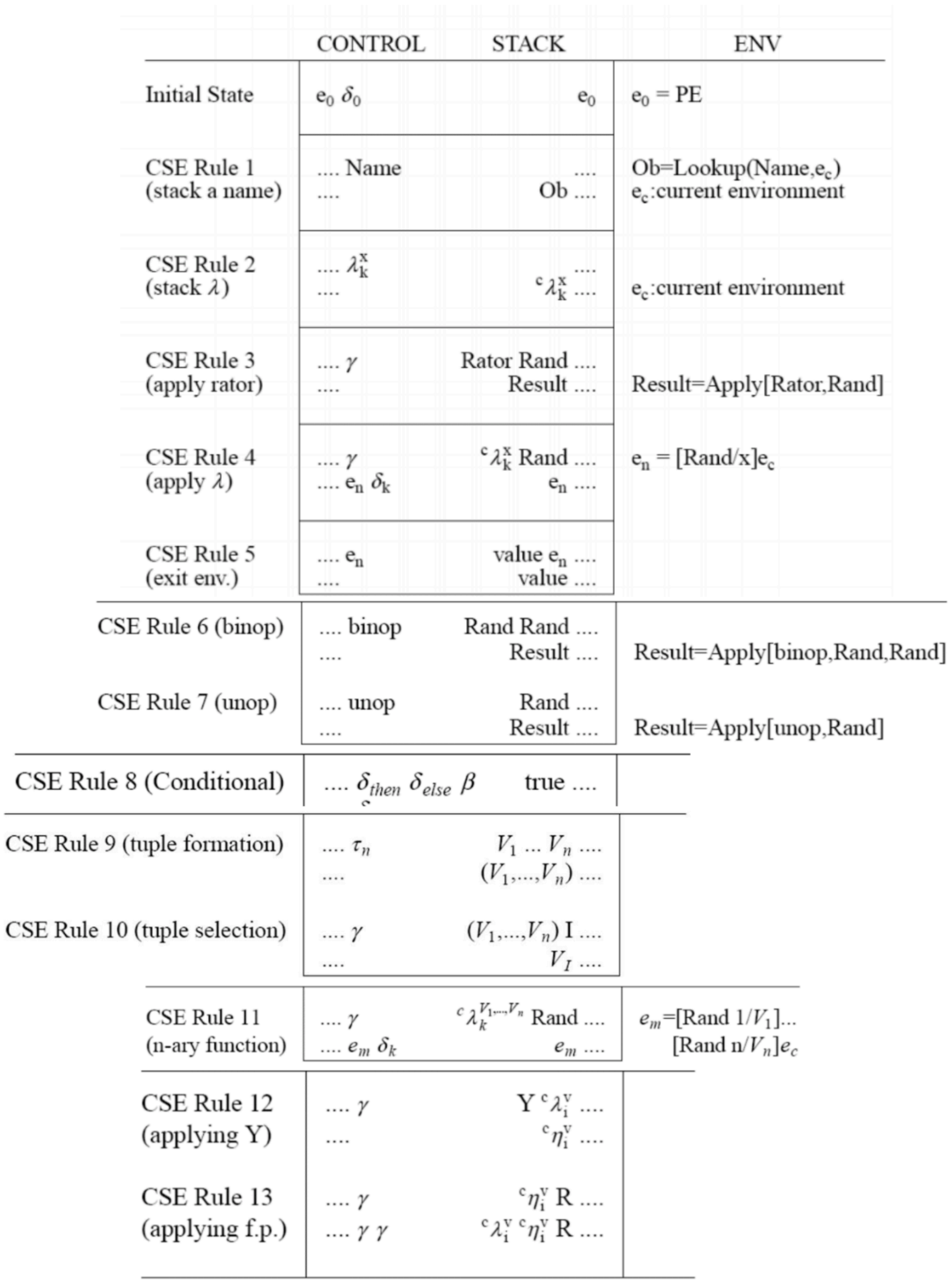
The "environment" class offers a means to represent and manage variable bindings within a specific scope in the CSE machine. It encompasses data members for maintaining the previous environment pointer, environment name, and a map for associating bound variables with their values. Although the function prototypes for the copy constructor and assignment operator are declared outside the class, their definitions are not provided in the header file.

APPENDIX

A. 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 ','                 => ','?;
```

B. CSE Machine Rules



	CONTROL	STACK	ENV
Initial State	$e_0\ \delta_0$	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 .... $e_n\ \delta_k$	${}^c\lambda_k^x$ Rand .... e_n	e_n = [Rand/x]e_c
CSE Rule 5 (exit env.)	.... e_n	value e_n .... value	
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)	.... $\delta_{then}\ \delta_{else}\ \beta$	true	
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 function)	.... γ .... $e_m\ \delta_k$	${}^c\lambda_k^{V_1,...,V_n}$ Rand .... e_m	e_m=[Rand 1/V_1]... [Rand n/V_n]e_c
CSE Rule 12 (applying Y)	.... γ	Y ${}^c\lambda_i^v$ .... ${}^c\eta_i^v$	
CSE Rule 13 (applying f.p.)	.... γ .... $\gamma\ \gamma$	${}^c\eta_i^v$ R .... ${}^c\lambda_i^v\ {}^c\eta_i^v$ R	