CS 3513 - Programming Languages

Programming Project 01

Group Number: 19

Group Members:

- Meddawitage K.C.D. - 210382H
- Pathirana D.P. - 210449V

Problem Description:

The project aimed to create tools for processing RPAL language programs, involving the development of systems to comprehend their structure and translate them into a machine readable format. This encompassed building a lexical analyzer to dissect RPAL code, a parser to comprehend the program's structure and convert it into an Abstract Syntax Tree (AST), a method to further simplify the AST into a Standardized Tree (ST), and the creation of a CSE machine capable of executing RPAL programs. Ultimately, the program should read RPAL code from a file and yield outputs akin to those produced by the "myrpal.exe" tool.

Program Execution Instructions:

The following sequence of commands can be used to run the code in cmd or vscode,

- To get the numerical output directly:
  - .\myrpal input.txt
- To get the ST along with the numerical output:
  - .\myrpal -st input.txt
- To get the AST along with the numerical output:
  - .\myrpal -ast input.txt

Also the following sequence of commands can be used in the root of the project directory to compile the program and execute rpal programs:

```
>make
>./myrpal input.txt
```

Our Makefile consists of the following codes;

```
all:
        g++ myrpal.cpp -o myrpal
```

This compiles the myrpal.cpp file using the GNU C++ compiler and produces an executable named myrpal.

```
cl:
        rm -f *.o myrpal
```

This code removes all object files (*.o) and the executable file from the directory.

Structure of the Project:

This project was coded entirely in C++. It consists of mainly 5 files.
They are,

1. myrpal.cpp
2. parser.h
3. tree.h
4. token.h
5. environment.h

This document outlines the purpose of each file and presents the function prototypes along with their uses.

1. Myrpal.cpp

a. Introduction

The main function acts as the starting point for the program's execution. It receives command-line arguments, reads the contents of a file, and instantiates a parser object to parse the file's content. The parser object's implementation resides in a distinct file named "parser.h".

b. Structure of the Program

The program is a simple C++ program consisting of a single "main" function, which is the starting point of the program. Below is the structure of the program,

1. Include Statements:

   The program includes the following header files:

```
#include <iostream>
#include <fstream>
```

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

These headers are necessary to utilize various C++ standard library functionalities and to include the "parser.h" file, which contains the implementation of the parser.

2. Main Function:

   The "main" function is the entry point of the program and is responsible for parsing the command-line arguments, reading the content of the file, and initiating the parsing process using the parser object.

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

3. Parsing Command-Line Arguments:

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

```
if (argc > 1)
    {
        // Parsing command-line arguments
    }
    else
    {
        cout << " Error : Incorrect no. of inputs " <<
endl;
    }
```

4. Reading and Processing the File:

   The main function reads the content of the file specified by the command-line argument and stores it in a string.

```
string filepath = argv[argv_idx];    // Read file name
from command line
        const char *file = filepath.c_str(); //
Convert string to char array

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

```
            return 1;
        }
        // Open file
        string
file_str((istreambuf_iterator<char>(input)),
(istreambuf_iterator<char>())); // Read file into
string
        input.close();

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

5. Creating and Initiating the Parser Object:

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

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

   The parser object is constructed by passing the char array containing the file content, the starting index (0), the size of the content, and the AST or ST flag.

2. Parser.h

a. Introduction

The parser.h file contains the implementation of a Recursive Descent Parser. This parser is responsible for tokenizing, parsing, and transforming input code into a standardized tree (ST) representation for subsequent execution. It adheres to predefined grammar rules to identify the syntax and structure of the programming language it is designed to parse.

b. Structure of the Program

1. Tokenization:

   The parser starts by tokenizing the input code using the `'getToken(char read[])'` function, which reads characters individually and categorizes them into different types of tokens.

These tokens include identifiers, keywords, operators, integers, strings, punctuation, comments, spaces, and unknown tokens. The tokenization process sets the foundation for the subsequent parsing steps.

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

   The AST is built using the `'buildTree()'` function. It constructs tree nodes based on the tokens' properties and pushes them onto the syntax tree stack (`st`). The AST represents the syntactic structure of the input code. 3 The `'makeST(tree *t)'` function is then used to convert the AST into a standardized tree (ST). The ST is created by applying transformations to the AST to standardize its representation. This standardized representation ensures consistency in the tree structure and prepares the code for further execution.

3. Control Structures Execution:

   After constructing the standardized tree, the control structures are generated using the `'createControlStructures()'` function. The control structures represent the set of instructions required to execute the code in the Control Stack Environment (CSE) machine, a theoretical model for executing high-level functional programming languages.

   The `'cse_machine()'` function is the main driver function that executes the generated control structures based on the standard 13 rules. It uses four stacks (`control`, `m_stack`, `stackOfEnvironment`, and `getCurrEnvironment`) to manage the control flow, operands, environments, and access to the current environment, respectively. The CSE machine supports 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 includes several helper functions, such as `'isAlpha(char ch)'`, `'isDigit(char ch)'`, `'isBinaryOperator(string op)'`, and `'isNumber(const std::string &s)'` used for token classification. Additionally, there are `'arrangeTuple()'` and `'addSpaces(string temp)'` functions, which are used in the context of processing and arranging tree nodes, especially for handling tuples and escape sequences in strings.

5. Grammar Rules and Recursive Descent Parsing:

   The parser follows a set of grammar rules to recognize and parse the input code. The main grammar rules are defined in the form of recursive descent parsing functions. Each function corresponds to a non-terminal in the grammar, and they recursively call each other to handle nested structures.

   The grammar rules cover various language constructs, such as let expressions, function definitions, conditional expressions, arithmetic expressions, and more. The grammar rules implemented in this project are provided in the appendix at the end of this report.

6. Functions in the Parser:

   The following are functions in the parser class,

```
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)
token getToken(char read[])
void parse()
void makeST(tree *t)
tree *makeStandardTree(tree *t)
void createControlStructures(tree *x, tree
*(*setOfControlStruct)[200])
void cse_machine(vector<vector<tree *> >
&controlStructure)
void arrangeTuple(tree *tauNode, stack<tree *> &res)
string addSpaces(string temp)
```

7. 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_Da()
void procedure_Dr()
void procedure_Db()
void procedure_Vb()
void procedure_Vl()
```

3. Tree.h

a. Introduction

The tree.h file is a C++ header file that defines a tree class which is commonly used in programming. The structure of the tree class and implementation of its functions are as follows.

b. Structure of the Program

This tree class is used to create a binary tree data structure, where each node can have a left and a right child. Each node in the tree also has a val and type attribute.

1. Header Guards

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

This pattern is used to prevent the same header file from being included multiple times, which can cause problems such as redefinition of classes, functions, or variables.

2. Include statements

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

These include directives in C++. They are used to include standard library headers into the program. These are important in C++ stack data structures and standard input/output streams.

3. Class definition

```
class tree
{
private:
    string val;  // Value of node
    string type; // Type of node

public:
    tree *left;                                    //
Left child
    tree *right;                                   //
Right child
}
```

This is a class definition for a tree in C++ with private data members and public member functions. It's a binary tree, where each node can have a left and a right child. Each node in the tree also has a val and type attribute.

4. Function prototypes

```
    void setType(string typ);                      // Set
type of node
    void setVal(string value);                     // Set
value of node
    string getType();                              // Get
type of node
    string getVal();                               // Get
value of node
    tree *createNode(string value, string typ); //
Create node
```

```
    tree *createNode(tree *x);                  //
Create node
    void print_tree(int no_of_dots);
// Print tree
```

These are member functions defined outside the class definition.

5. Member function definitions

   Tree scope resolution operator defines the member functions outside the class definition. This assigns the type and value of the tree node.

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

6. Function definitions

   These are defined outside the class as standalone functions.

```
tree *createNode(string value, string typ)
```

   This function creates a new tree node with the given value and type. It initializes the left and right pointers to NULL and returns a pointer to the newly created node.

```
tree *createNode(tree *x)
```

   This function creates a new tree node by copying the value and type from an existing tree node x. It also copies the left pointer from x but initializes the right pointer to NULL. It returns a pointer to the newly created node. This function can be used to create a copy of a node, but it does not create copies of the child nodes.

```
void tree::print_tree(int no_of_dots)
```

   This function is used to print the tree. The no_of_dots parameter is used to control the number of dots printed for indentation, to visualize the tree structure.

7. Header guard closure

   The #endif is closing a conditional block started with #ifndef TREE_H_.

4. Token.h

a. Introduction

The ‘token’ class is a C++ implementation of a basic token representation. Tokens divide the source code into meaningful components.

b. Structure of the Program

This is a token header file named ‘token.h’ including token class and related functions.

1. Header Guards

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

   This pattern is used to prevent the same header file from being included multiple times, which can cause problems such as redefinition of classes, functions, or variables.

2. Include statements

```
#include <iostream>
```

   They are used to include standard library headers into the program. These are important in standard input/output streams in C++.

3. Class definition

```
class token{
      Private:
            string type; // Type of token
            string val;
};
```

   This is the beginning of a class definition for a token in C++. A token object has two private member variables: type and val.

4. Function prototypes

```
void setType(const string &sts); // Set type of token
void setVal(const string &str);  // Set value of token
string getType();                // Get type of token
string getVal();                 // Get value of token
bool operator!=(token t);
```

These functions get and set the type and value of a token and overload the inequality operator for token comparison.

5. Member function definitions

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

This function sets the type of a token object. It takes a constant reference to a string str as an argument and assigns it to the private member variable type.

```
void token::setVal(const string &str)
```

This function sets the val of a token object. It takes a constant reference to a string str as an argument and assigns it to the private member variable val.

6. Operator overload definitions

These are defined outside the class as standalone functions. The ‘operator!=’ function is used for token comparison.

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

7. Header guard closure

The #endif is closing a conditional block started with #ifndef TOKEN_H_.

5. Environment.h

a. Introduction

The “environment” class is a C++ implementation of an environment, which is used in the CSE machine to keep track of variable bindings and their values in a specific scope.

b. Structure of the Program

This is a C++ header file defining a class named environment. This contains "environment.h" header file with the implementation of "environment" class and related functions.

1. Header Guards

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

They prevent the header file from being included more than once in the same file or in other files, which can cause problems such as redefinition of classes, functions, or variables.

2. Include Statements

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

These lines include the map and iostream standard libraries. map is a container that stores elements formed by a combination of a key value and a mapped value, following a specific order. iostream provides facilities for input/output operations.

3. Class Definition

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

```
environment *prev;
```

This is a public member variable that stores a pointer to the previous environment object. This could be used to create a linked list of environments, where each environment has a reference to the one before it.

```
map<tree *, vector<tree *> > boundVar;
```

This public member variable stores a map. The keys are pointers to tree objects, and the values are vectors of pointers to tree objects. This could be used to map variables to their values.

```
environment()
    {
        prev = NULL; // Default previous environment
        name = "env0"; // Default name
    }
```

This is the default constructor for the environment class. It initializes prev to NULL and name to "env0".

4. Function Prototypes

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

No any function prototypes declared in the class definition. There’s a copy constructor and an assignment operator outside the class.

5. Header Guard Closure

```
#endif
```

This environment.h file is closed with a header guard closure. This ends the code which is conditionally compiled.