CS 3512 - Programming Languages Programming Project 01

Group Number: 03

Group Members:

- Harikishna N. 210206B
- Harismnenan J. 210207E

Problem Description:

The project required the implementation of a lexical analyzer and a parser for the RPAL language. The parser's output should be an Abstract Syntax Tree (AST) for the given input program. Following this, an algorithm needed to be implemented to convert the AST into a Standardized Tree (ST), and the CSE machine should be developed. The program should be capable of reading an input file containing an RPAL program and producing output that matches the output of "rpal.exe" for the corresponding program.

Program Execution Instructions:

To compile the program and execute RPAL programs, the following sequence of commands can be used in the root of the project directory:

```
make
main.exe file_name
```

The program has been tested with the following sequence of commands:

```
make
main.exe rpal_test_programs/rpal_01
main.exe rpal_test_programs/rpal_02
main.exe rpal_test_programs/rpal_03
main.exe rpal_test_programs/rpal_04
main.exe rpal_test_programs/rpal_05
main.exe rpal_test_programs/rpal_06
```

Structure of the Project:

This project was coded entirely in C++ and consists of five main files:

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

This document outlines each file's purpose and their function prototypes.

1. Main.cpp

Introduction

The main function serves as the program's entry point, handling command-line arguments, reading a specified file's content, and creating a parser object to parse the content. The parser object is implemented in "parser.h".

Structure of the Program

The program is a simple C++ application consisting of a single main function, which acts as the starting point. 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 enable various C++ standard library functionalities and include "parser.h" for parser implementation.

2. **Main Function:**
   The main function is the entry point of the program. It 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 file specified by the command-line argument and stores it in a string.
   ```
   string filepath = argv[argv_idx]; const char *file =
   filepath.c_str(); ifstream input(filepath);

   if (!input) {
                   std::cout << "File not found!" << "\n"; return 1;
   }
   ```

```
string file_str((istreambuf_iterator<char>(input)), (istreambuf_iterator<char>()));
input.close(); 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.

Conclusion

The main function is the central component of the program. It is responsible for reading the content of a file, processing the command-line arguments, and initiating the parsing process. It utilizes the parser object to handle the parsing of the file's content. The program acts as a basic driver for the parser, allowing the parsing of files and optionally printing the Abstract Syntax Tree (AST) or Symbol Table (ST) based on the provided command-line flags.

2. Parser.h

Introduction

The Recursive Descent Parser is implemented in parser.h. It tokenizes, parses, and converts input code into a standardized tree (ST) for further execution, following grammar rules to recognize syntax and structure.

Structure of the Program

1. **Tokenization:**
   The parser begins by tokenizing the input code using `getToken()` which reads characters and categorizes them into different token types. This step is crucial for subsequent parsing.

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.

The `makeST()` 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 ST, control structures are generated using `createControlStructures()`, representing instructions for executing code in the Control Stack Environment (CSE) machine. The `cse_machine()` function executes these control structures using four stacks to manage control flow, operands, environments, and current environment access. The CSE machine supports various operations, including lambda functions, conditional expressions, tuples, built-in functions, unary and binary operators, and functional programming.

4. **Helper Functions:**
   The parser includes several helper functions, such as `isAlpha()`, `isDigit()`, `isBinaryOperator()`, and `isNumber()`, used for token classification. Additionally, there are `arrangeTuple()` and `addSpaces()` 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 grammar rules for recognizing and parsing input code, with each rule defined in recursive descent parsing functions corresponding to grammar non-terminals. The grammar rules cover constructs like let expressions, function definitions, conditional expressions, and arithmetic expressions, detailed in the appendix.

6. **Functions in the Parser:**
   Key functions include:

   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_Dr**()	void **procedure_Db**()
void **procedure_Vb**()	void **procedure_Vl**()	

   The corresponding grammar productions are provided in the appendix at the end of this report.

Conclusion

`parser.h` contains a comprehensive Recursive Descent Parser, tokenizing, parsing, and converting input code into a standardized tree. It follows grammar rules and uses recursive descent parsing for various programming constructs. The CSE machine integration enables code execution based on semantic meaning.

3. Tree.h

Introduction

The "tree" class represents a syntax tree, a data structure used in programming languages to represent source code's syntactic structure. This report provides an overview of the "tree" class, its function prototypes, and the overall program structure.

Structure of the Program

The program consists of a header file named "tree.h" containing the implementation of the "tree" class and related functions. Here is the structure of the program:

1. **Header Guards:**
   `#ifndef TREE_H_`
   `#define TREE_H_`
   The header guards prevent multiple inclusions of the "tree.h" file in the same translation unit, avoiding potential compilation errors.

2. **Include Statements:**
   The program includes:
   `#include <iostream>`
   `#include <stack>`
   These headers are necessary for standard input/output streams and the C++ stack data structure.

3. **Class Definition:**
   The "tree" class is defined with private data members and public member functions, representing a syntax tree node.

```
class tree {
        private:
                string val; // Value of node string type; //
                Type of node
        public:
                tree *left; // Left child tree *right;//
                Right child
};
```

4. **Function Prototypes:**
   The "tree" class functions include:

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

5. **Member Function Definitions:**
   Member functions are defined using the "tree::" scope resolution operator.
   `void tree::setType(string typ) void tree::setVal(string value)`

6. **Function Definitions:**
   The "createNode" function and the "print_tree" function are defined outside the class as standalone functions.
   `tree *createNode(string value, string type) tree *createNode(tree *x)`
   `void tree::print_tree(int no_of_dots)`

7. **Header Guard Closure:**
   The "tree.h" file is closed with the header guard closure:
   `#endif`

Conclusion

The "tree" class provides a structured representation of a syntax tree node, with member functions to set and retrieve node values and types, create new nodes, and print the tree structure. This class is fundamental to constructing and manipulating the Abstract Syntax Tree (AST) and Standardized Tree (ST) in the RPAL language parser.

4. Token.h

Introduction

`token.h` defines the `token` class representing tokens generated by the lexical analyzer. The lexical analyzer reads input characters and categorizes them into token types, crucial for the parsing process.

Structure of the Program

The program consists of a header file named "token.h" containing the implementation of the "token" class and related functions. Here is the structure of the program:

1. **Header Guards:**
   ```
   #ifndef TOKEN_H_
   #define TOKEN_H_
   ```
   The header guards prevent multiple inclusions of the "token.h" file in the same translation unit, avoiding potential compilation errors.

2. **Include Statements:**
   The program includes the following header file:
   `#include <iostream>`
   This header is necessary for using standard input/output streams.

3. **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;
};
```

4. **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);
```

These functions are used to set and get the type and value of a token, as well as to overload the inequality operator for token comparison.

5. **Member Function Definitions:**
   After the class definition, the member functions are defined outside the class using the "token::" scope resolution operator.
   ```
   void token::setType(const string &str) void
   token::setVal(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)
   bool token::operator!=(token t)
   ```

7. **Header Guard Closure:**
   The "token.h" file is closed with the header guard closure:
   ```
   #endif
   ```

Conclusion

The "token" class provides a simple representation of tokens used in programming languages. It allows for setting and getting the type and value of a token and overloads the inequality operator for token comparison. By using this class, we can work with individual tokens and perform various operations related to tokenization and syntax analysis.

5. Environment.h

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. This report provides an overview of the "environment" class, its function prototypes, and the overall structure of the program.

Structure of the Program

The program includes a header file, `environment.h`, containing the `environment`class and related functions.
Here is the structure of the program:

1. **Header Guards:**
   ```
   #ifndef ENVIRONMENT_H_
   #define ENVIRONMENT_H
   ```

The header guards prevent multiple inclusions of the "environment.h" file in the same translation unit, avoiding potential compilation errors.

2. **Include Statements:**
   The program includes the following header files:
   `#include <map> #include <iostream>`
   These headers are necessary for using the C++ map container and standard input/output streams.

3. **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";
        }
};
```

4. **Function Prototypes:**
   The class "environment" does not have any function prototypes declared within the class definition. However, there is a copy constructor and an assignment operator declared outside the class.

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

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:**
   The "environment.h" file is closed with the header guard closure:
   `#endif`

Conclusion

The `environment` class manages variable bindings in the RPAL program execution environment, providing functions to add and retrieve bindings. This class is crucial for maintaining the state of the program during execution.

Appendix

A. 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	

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