CS 3512 - Programming Languages
Programming Project 01

Group Number: 46

Group Members:

- Kobinath A. - 200308F
- Niles D.A. - 200421U
- Rajeevan Y. - 200501P
- Vibulan J. - 200677H

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 Standardize 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:

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

```
> make
> ./rpal20 file_name
```

The program is also tested to run with the following sequence commands:

```
> make
> ./rpal20 rpal_test_programs/rpal_01 > output.01
> diff output.01 rpal_test_programs/output01.test
> ./rpal20 rpal_test_programs/rpal_02 > output.02
> diff output.02 rpal_test_programs/output02.test
```

Structure of the Project:

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

1. main.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. Main.cpp

Introduction

The main function serves as the entry point for the program. It takes command-line arguments, reads the content of a file, and then creates a parser object to parse the file's content. The parser object is implemented in a separate file called "parser.h".

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];
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, 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 is used 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 the `parser.h` file. The parser is designed to tokenize, parse, and convert a given input code into a standardized tree (ST) representation for further execution. It follows a set of grammar rules to recognize the syntax and structure of the programming language it can parse.

Structure of the Program

1. **Tokenization:**
   The parser starts by tokenizing the input code using the `getToken()` 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.

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 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()`, `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 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_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

The `parser.h` file contains a comprehensive Recursive Descent Parser implementation that effectively tokenizes, parses, and converts input code into a standardized tree representation. The parser follows a set of grammar rules and utilizes recursive descent parsing to handle a wide range of programming language constructs. The integration of the Control Stack Environment (CSE) machine enables the execution of code based on its semantic meaning.

3. Tree.h

Introduction

The "tree" class is a C++ implementation of a syntax tree, a data structure commonly used in programming languages for representing the syntactic structure of source code. This report provides an overview of the "tree" class, its function prototypes, and the overall structure of the program.

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 the following header files:
   `#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. The class represents a node in the syntax tree.

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

```
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:**
   After the class definition, the member functions are defined outside the class 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 is a representation of a syntax tree with member functions to manipulate the nodes and print the tree structure. The function prototypes, as presented in this report, allow for the creation of nodes, setting and getting node values and types, and printing the syntax tree in a visually informative manner.

4. Token.h

Introduction

The "token" class is a C++ implementation of a basic token representation. Tokens are fundamental units in programming languages and are used to break down source code into meaningful components. This report provides an overview of the "token" class, its function prototypes, and the overall structure of the program.

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 consists of a header file named "environment.h" containing the implementation of 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 provides a way to represent and manage variable bindings within a specific scope in the CSE machine. It includes data members for the previous environment pointer, the name of the environment, and a map to associate 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. 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 ','                 => ','?;
```