CS 3513

Programming Languages

Project Report

Group 28

Members:

Sandeepa H.N.A. - 210571L

Manamperi L.R. - 210369B

Table of Contents

Problem Description:

The task was to develop a lexical analyzer and parser for the RPAL language, generating an Abstract Syntax Tree (AST) from the input program. Additionally, an algorithm was required to convert the AST into a Standardize Tree (ST), followed by implementing the CSE machine. The program should read an input file containing an RPAL program and produce output consistent with that of "rpal.exe" for the corresponding program.

How to execute the program:

Step 01: Create an executable file

1. Open your preferred IDE or terminal.
2. Navigate to the directory.
3. Type the following command and enter:

   ```
   make
   ```

Step 02: Run the executable file

To execute the program correctly, follow these steps:

4. Type the following command and enter:

   ```
   ./myrpal file_name
   ```

Replace file_name with the name of the file you want to process.

The sample input file input.txt is included in the directory.

For example, the given input.txt that you want to use as input, you would type:

```
./myrpal input.txt
```

This command will execute the myrpal program with input.txt as the input file. The sample input given in the project description question is also included as input.txt in the directory.

```
PS D:\Academic\Programming Languages\Assignments\Project\RPAL-Interpreter> make
g++ main.cpp -o myrpal
PS D:\Academic\Programming Languages\Assignments\Project\RPAL-Interpreter> ./myrpal input.txt
15
```

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

1. Main.cpp

Introduction

The main function acts as the program's entry point, accepting command-line arguments, reading the contents of a file, and instantiating a parser object to parse the file's content. The parser object is implemented in a distinct file named "parser.h".

Structure of the Program

The program is a straightforward C++ application featuring a single "main" function, serving as the program's entry point. Below outlines the structure of the program:

1. Include Statements:

```
#include <iostream>

#include <fstream>

#include <cstdlib>

#include <string.h>

#include "parser.h"
```

These headers are essential for accessing various functionalities from the C++ standard library and for including the "parser.h" file, which includes the parser's implementation.

2. Main Function:

The "main" function serves as the program's entry point, handling tasks such as parsing command-line arguments, reading file contents, and initializing the parsing process through the parser object.

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

3. Parsing Command-Line Arguments:

The main function verifies the existence of command-line arguments to identify the filename and determine 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 variable.

```
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 instantiates a parser object and initiates the parsing process by invoking 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 character array containing the file content, the starting index (0), the size of the content, and the AST or ST flag as parameters.

Conclusion

The "main" function serves as the core component of the program, handling tasks such as reading file content, processing command-line arguments, and triggering the parsing process. It relies on the "parser" object to manage the parsing of the file's content. This program acts as a simple driver for the parser, enabling file parsing and optionally displaying the Abstract Syntax Tree (AST) or Symbol Table (ST) based on the provided command-line flags.

1. Parser.h

Introduction

The Recursive Descent Parser is implemented within the `parser.h` file. This parser is engineered to tokenize, parse, and transform a provided input code into a standardized tree (ST) representation, facilitating subsequent execution. It adheres to a predefined set of grammar rules to identify the syntax and structure of the programming language it's capable of parsing.

Structure of the Program

1. Tokenization:

The parser initiates the tokenization process of the input code by employing the getToken() function. This function reads characters one by one, classifying them into various types of tokens such as 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 construction is facilitated by the `buildTree()` function. This function creates tree nodes according to the properties of the tokens and adds them to the syntax tree stack (`st`). The AST encapsulates the syntactic arrangement of the input code.

The `makeST()` function is subsequently employed to transform the AST into a standardized tree (ST). This process involves applying transformations to the AST to standardize its representation. By doing so, the resulting ST ensures uniformity in the tree structure, thus priming the code for subsequent execution.

3. Control Structures Execution:

Following the construction of the standardized tree, control structures are generated using the `createControlStructures()` function. These control structures encompass the series of instructions necessary for executing the code within the Control Stack Environment (CSE) machine. The CSE machine serves as a theoretical framework for executing high-level functional programming languages.

The `cse_machine()` function serves as the primary driver function responsible for executing the generated control structures in accordance with the standard 13 rules. It utilizes four stacks (`control`, `m_stack`, `stackOfEnvironment`, and `getCurrEnvironment`) to oversee the control flow, operands, environments, and access to the current environment, respectively. The CSE machine offers 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 various auxiliary functions like `isAlpha()`, `isDigit()`, `isBinaryOperator()`, and `isNumber()` to aid in token classification. Furthermore, there exist `arrangeTuple()` and `addSpaces()` functions, primarily utilized in the context of organizing and manipulating tree nodes, particularly for managing tuples and escape sequences within strings.

5. Grammar Rules and Recursive Descent Parsing:

The parser adheres to a predefined set of grammar rules to identify and parse the input code. These rules are articulated through recursive descent parsing functions, with each function representing a non-terminal in the grammar. These functions recursively invoke one another to manage nested structures effectively.

These grammar rules encompass a wide array of language constructs, including let expressions, function definitions, conditional expressions, arithmetic expressions, and more. For detailed reference, the grammar rules implemented in this project are provided in the appendix at the conclusion of this report.

6. Functions in the Parser:

Here are the functions in the parser class:

`parser(char read_array[], int i, int size, int af)` // Constructor for the parser class

`bool isReservedKey(std::string str)` // Checks if a given string is a reserved keyword

`bool isOperator(char ch)` // Checks if a given character is an operator

`bool isAlpha(char ch)` // Checks if a given character is an alphabetic character

`bool isDigit(char ch)` // Checks if a given character is a digit

`bool isBinaryOperator(std::string op)` // Checks if a given string is a binary operator

`bool isNumber(const std::string &s)` // Checks if a given string represents a number

`void read(std::string val, std::string type)` // Reads a value with its type

`void buildTree(std::string val, std::string type, int child)` // Builds a tree node with a value, type, and number of children

`token getToken(char read[])` // Gets the next token from the input

`void parse()` // Parses the input code

`void makeST(tree *t)` // Converts the Abstract Syntax Tree (AST) into a Standardize Tree (ST)

`tree *makeStandardTree(tree *t)` // Constructs a standardized tree from the AST

`void createControlStructures(tree *x, tree *(*setOfControlStruct)[200])` // Creates control structures for execution

`void cse_machine(std::vector<std::vector<tree *>> &controlStructure)` // Executes control structures using the CSE machine

`void arrangeTuple(tree *tauNode, std::stack<tree *> &res)` // Arranges tuples in the tree node

`std::string addSpaces(std::string temp)` // Adds spaces to a string

These functions collectively contribute to tokenization, parsing, building the Abstract Syntax Tree (AST), converting it to a Standardize Tree (ST), and executing control structures in the parser class.

7. Grammar Rule Procedures:

Here are the functions representing 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()
```

These procedures align with the grammar rules of RPAL, which are detailed in the appendix at the end of this report.

Conclusion

The parser.h file includes the Recursive Descent Parser implementation, proficient in tokenizing, parsing, and converting input code into a standardized tree structure. Adhering to a predefined set of grammar rules, it employs recursive descent parsing to manage diverse programming language constructs adeptly. The integration of the Control Stack Environment (CSE) machine further empowers the parser to execute code based on its semantic interpretation.

2. Tree.h

Introduction

The "tree" class in C++ serves as an implementation of a syntax tree, a fundamental data structure utilized in programming languages to represent the syntactic structure of source code. This report offers an insight into the "tree" class, including its function prototypes and the overarching structure of the program.

Structure of the Program

The program comprises a header file named "tree.h," encompassing the implementation of the "tree" class alongside associated functions. Below outlines the structure of the program:

1. Header Guards:

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

The header guards serve to prevent multiple inclusions of the "tree.h" file within the same translation unit, mitigating the risk of compilation errors.

2. Include Statements:

The inclusion of the following header files is essential for the program's functionality:

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

These headers facilitate standard input/output operations and provide access to the stack data structure within the program.

3. Class Definition:

The "tree" class is structured with private data members and public member functions, representing a node within the syntax tree.

```
class tree {

private:

    std::string val; // Value of the node

    std::string type; // Type of the node

public:
```

```
    tree *left; // Left child

   tree *right; // Right child

};
```

This class encapsulates the structure of a syntax tree node, with "val" representing the node's value, "type" denoting its type, and "left" and "right" pointers referencing its left and right children, respectively.

4. Function Prototypes:

The function prototypes for the member functions of the "tree" class are as follows:

`void setType(std::string typ);` // Sets the type of the tree node

`void setVal(std::string value);` // Sets the value of the tree node

`std::string getType();` // Retrieves the type of the tree node

`std::string getVal();` // Retrieves the value of the tree node

`tree *createNode(std::string value, std::string typ);` // Creates a new tree node with the specified value and type

`tree *createNode(tree *x);` // Creates a new tree node by copying an existing tree node

`void print_tree(int no_of_dots);` // Prints the tree structure with indentation specified by the number of dots

These member functions contribute to the manipulation and inspection of tree nodes within the "tree" class.

5. Member Function Definitions:

After the class definition, the member functions are defined outside the class using the "tree::" scope resolution operator as shown below:

`void tree::setType(string typ)` // Sets the type of the tree node

`void tree::setVal(string value)` // Sets the value of the tree node

These member functions provide functionality to set the type and value of a tree node, respectively.

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)` // Creates a new tree node with the specified value and type

`tree *createNode(tree *x)` // Creates a new tree node by copying an existing tree node

`void tree::print_tree(int no_of_dots)` // Prints the tree structure with indentation specified by the number of dots

These functions are responsible for creating new tree nodes and printing the tree structure, respectively.

7. Header Guard Closure:

The "tree.h" file is concluded with the closure of the header guard:

```
#endif
```

This ensures that the contents of the file are included only once during compilation, preventing any potential issues with redefinitions.

Conclusion

The "tree" class serves as a representation of a syntax tree, equipped with member functions for node manipulation and tree structure printing. The function prototypes outlined in this report facilitate node creation, setting and retrieval of node values and types, as well as printing the syntax tree in a visually informative manner.

3. Token.h

Introduction

The "token" class in C++ serves as a basic representation of a token, which is a fundamental unit in programming languages used to break down source code into meaningful components. This report offers an overview of the "token" class, including its function prototypes and the overarching structure of the program.

Structure of the Program

The program comprises a header file named "token.h," encompassing the implementation of the "token" class alongside associated functions. Below outlines the structure of the program:

1. Header Guards:

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

The header guards within the "token.h" file prevent multiple inclusions of the file within the same translation unit, thus averting potential compilation errors.

2. Include Statements:

The inclusion of the <iostream> header file in the program is essential for utilizing standard input/output streams.

```
#include <iostream>
```

3. Class Definition:

The "token" class is defined with private data members and public member functions, serving as a representation of a single token in the programming language.

```
class token {

private:

string type;

string val;

};
```

This class encapsulates the structure of a token, with "type" representing the type of the token and "val" denoting its value.

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 std::string &sts);` // Sets the type of the token

`void setVal(const std::string &str);` // Sets the value of the token

`std::string getType();` // Retrieves the type of the token

`std::string getVal();` // Retrieves the value of the token

`bool operator!=(token t);` // Overloads the inequality operator for token comparison

These functions are utilized to manipulate the type and value of a token, and also to overload the inequality operator for comparing tokens.

5. Member Function Definitions:

Following the class definition, the member functions are defined outside the class scope using the "token::" scope resolution operator. Here are the definitions:

`void token::setType(const string &str)` // Sets the type of the token

`void token::setVal(const string &str)` // Sets the value of the token

6. Operator Overload Definition:

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

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

7. Header Guard Closure:

The "token.h" file is concluded with the closure of the header guard:

`#endif`

Conclusion

The "token" class is a basic way to show tokens used in programming languages. It lets you set and get the type and value of a token. It also changes how tokens are compared using the inequality operator. With this class, we can handle single tokens and do different tasks like tokenizing and analyzing syntax.

4. Environment.h

Introduction

The "environment" class in C++ serves as an implementation of an environment within the CSE machine. Its purpose is to manage variable bindings and their corresponding values within a particular scope. This document offers a summary of the "environment" class, including its function prototypes and the general program structure.

Structure of the Program

The program comprises a header file called "environment.h," which encompasses the implementation of the "environment" class along with associated functions.

Here is the structure of the program:

1. Header Guards:

```
#ifndef ENVIRONMENT_H_

#define ENVIRONMENT_H
```

Header guards serve to prevent multiple inclusions of the "environment.h" file within the same translation unit, thus mitigating the risk of compilation errors.

2. Include Statements:

The program incorporates the subsequent header files:

`#include <map>` // map header for using map containers

`#include <iostream>` // iostream header for input and output operations

These headers are essential for utilizing the C++ map container and standard input/output streams.

3. Class Definition:

The "environment" class is delineated with public data members and a default constructor. This class embodies an environment within the CSE machine.

```
class environment {

public:
```

```
environment *prev;

string name;

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

environment() {

prev = NULL; name = "env0";

}

};
```

4. Function Prototypes:

Within the class definition of "environment," there are no function prototypes declared. Nevertheless, a copy constructor and an assignment operator are declared outside the class.

`environment(const environment &);` // Copy constructor declaration

`environment &operator=(const environment &env);` // Assignment operator declaration

5. Member Function Definitions:

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

6. Header Guard Closure:

The "environment.h" file concludes with the closure of the header guard:

```
#endif
```

Conclusion

The "environment" class offers a way for representing and handling variable bindings within a designated scope in the CSE machine. It encompasses data members for the previous environment pointer, the environment's name, and a map to correlate bound variables with their values. While the function prototypes for the copy constructor and assignment operator are declared outside the class, their definitions are absent from the header file.

Sample Input and Output

Sample 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) )
```

Sample Output

```
PS D:\Academic\Programming Languages\
Assignments\Project\RPAL-Interpreter>
 make
g++ main.cpp -o myrpal
PS D:\Academic\Programming Languages\
Assignments\Project\RPAL-Interpreter>
 ./myrpal input.txt
15
```

Output of Abstract Syntax Tree (AST)

```
PS D:\Academic\Programming Languages\Assignments\Project
\RPAL-Interpreter> ./myrpal -ast input.txt
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>
..gamma
...<ID:Sum>
...tau
....<INT:1>
....<INT:2>
....<INT:3>
....<INT:4>
....<INT:5>
15
```

Output of Standardized Tree (ST)

```
PS D:\Academic\Programming Languages\Assignments\Project\RPAL-Interpreter>
            ./myrpal -st input.txt
gamma
.lambda
..<ID:Sum>
..gamma
...<ID:Print>
...gamma
....<ID:Sum>
....tau
.....<INT:1>
.....<INT:2>
.....<INT:3>
.....<INT:4>
.....<INT:5>
.lambda
..<ID:A>
..gamma
...lambda
....<ID:Psum>
....gamma
.....<ID:Psum>
.....tau
......<ID:A>
......gamma
.......<ID:Order>
.......<ID:A>
...gamma
....YSTAR
....lambda
.....<ID:Psum>
.....lambda
......,
.......<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>
15
```

References

[1] Dr. Adeesha Wijayasiri, "Lecture Slides for CS3513 Programming Languages," University of Moratuwa, Sri Lanka.

[2] “RPAL - Right-reference Pedagogic Algorithmic Language.” https://rpal.sourceforge.net/

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 ','                 => ','?;
```