CS 3513 – Programming Languages

Programming Project 1

Group 130

Lexical Analyzer, Parser, Standardize Tree Conversion and CSE Machine Implementation for the RPAL Language

Group Members :

Vithurshan.P - 210676M

Lithushan.S - 210340E

Introduction

Our project involves constructing a lexical analyzer and a parser for the RPAL language. We are required to create these components without relying on external tools such as 'lex' or 'yacc'. The lexical analyzer's task is to read the input RPAL program and tokenize it according to the lexical rules outlined in the RPAL_Lex.pdf document. Subsequently, the parser will utilize the grammar rules specified in the RPAL_Grammar.pdf document to produce an Abstract Syntax Tree (AST) representation of the input program.

The objective of the project is to establish a comprehensive pipeline capable of handling an input RPAL program. This pipeline encompasses lexical analysis and parsing stages to construct the Abstract Syntax Tree (AST), followed by the transformation of the AST into Semantic Tree (ST). Ultimately, the program executes the transformed code using the CSE machine. It is imperative that the output produced by our implementation aligns with the output generated by the provided "rpal.exe" for the corresponding RPAL program, thus validating the accuracy of our solution. Java was employed as the programming language for this project.

In summary, this project was completed in four stages as follows:

1. Develop a lexical analyzer to tokenize the input RPAL program, breaking it down into meaningful units.
2. Construct a parser to check the program is syntactically correct by parsing and simultaneously create the Abstract Syntax Tree (AST) from the sequence of tokens obtained in the previous step.
3. Translate the AST into a Standardized Tree (ST) using the RPAL Subtree Transformational Grammar table, ensuring a consistent representation.
4. Implement the CSE machine, which processes the ST to evaluate expressions and produce the final output value.

Lexical Analyzer

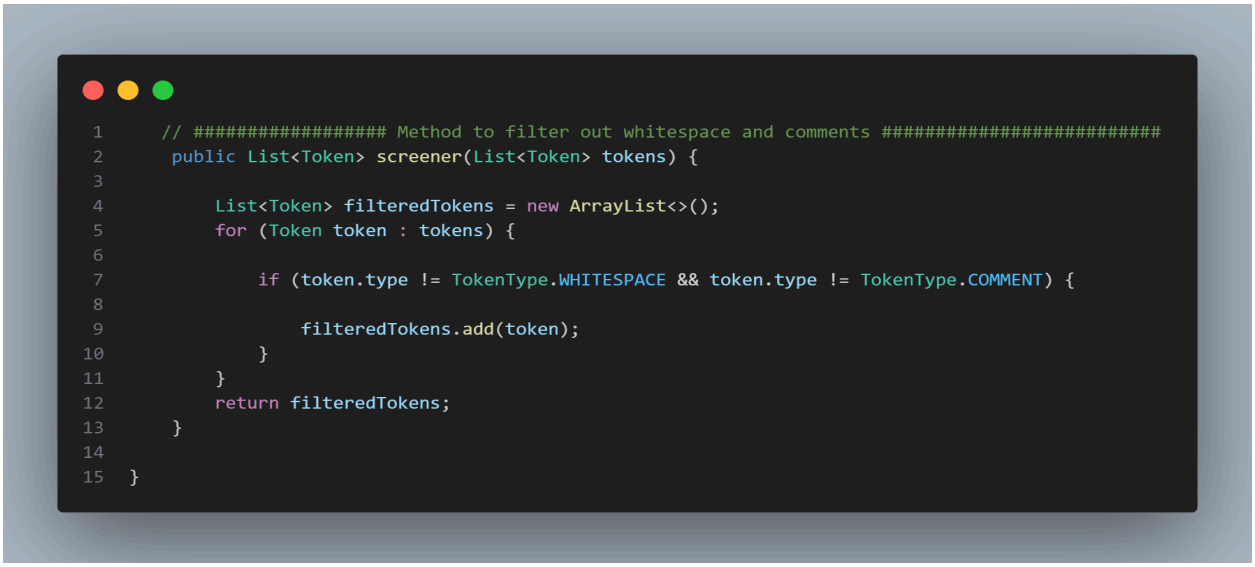
- The ‘LexAnalyser’ class represents the lexical analyzer. It has a constructor that takes the name of the input file as an argument.
- The ‘scanner’ method is responsible for scanning the input file line by line and tokenizing each line by invoking the ‘tokenize_Line’ method.
    - Possible errors while executing the ‘scanner’ method include files not found in that directory and errors while reading that file line by line. These are handled in a way that it will provide warning to the user what the issue occurred.

```java
// Method to scan the input file and tokenize it
public List<Token> scanner () {

    try (FileReader fileReader = new FileReader(fileName);
            BufferedReader reader = new BufferedReader(fileReader)) {

        String line;
        int lineCount = 0;

        while ((line = reader.readLine()) != null) {

            lineCount++;
            try {
                tokenize_Line(line, lineCount);
            } catch (Lexical_Exception e) {

                System.out.println("Error while reading line " + lineCount + ": " + e.getMessage());
            }
        }
    } catch (FileNotFoundException e) {

        System.out.println("File not found in the current directory");

    }catch(IOException e){

        System.out.println("IOException !!!");
    }

    return screener(tokens);
}
```

- The ‘tokenize_Line’ method tokenizes each line of input based on certain rules: It identifies identifiers, keywords, integers, operators, string literals, and punctuation marks. It throws lexical exceptions for unrecognized tokens or undetermined string literals.

- The ‘isKeyword’, ‘isOperator’, and ‘isPunctuation’ methods are used as a helper methods to check if a given token or character is a keyword, operator, or punctuation mark, respectively.

- The ‘screener’ method filters out whitespace and comments from the list of tokens and adds the filtered tokens into a new array list and returns it.



```
    // ################## Method to filter out whitespace and comments ##########################
     public List<Token> screener(List<Token> tokens) {

         List<Token> filteredTokens = new ArrayList<>();
         for (Token token : tokens) {

             if (token.type != TokenType.WHITESPACE && token.type != TokenType.COMMENT) {

                 filteredTokens.add(token);
             }
         }
         return filteredTokens;
     }

}
```

- The ‘Lexical_Exception’ class is a custom exception for handling lexical errors.
- The ‘Tester’ class allows you to test the functionality of the **LexAnalyser** by running it on a sample input file and observing the tokens generated from the input file. This can be helpful for debugging and verifying that the lexical analyzer is working correctly.
    - It creates an instance of the ‘LexAnalyser’ class, passing the name of the input file ("<File name>.txt") to the constructor.
    - It calls the scanner method of the ‘LexAnalyser’ instance to tokenize the contents of the input file.
    - In the end it iterates over the list of tokens returned by the ‘scanner’ method and prints each token along with its type and line number.

- The ‘Token’ class provides a convenient way to encapsulate information about individual tokens produced by the lexical analyzer, making it easier to work with them in subsequent stages of the compilation process. It has three fields:
    - type: Indicates the type of the token, represented by the ‘TokenType’ enum.
    - value: Stores the actual value of the token, which could be a keyword, identifier, integer, etc.
    - line: Indicates the line number in the input file where the token was found.

- The ‘TokenType’ enum provides a standardized way to categorize and identify tokens, enabling easier processing and manipulation of token streams during lexical analysis.

Parser

- The ‘RPAL_parser’ class represents a parser for the RPAL programming language. It takes a list of tokens generated by a lexical analyzer and builds an Abstract Syntax Tree (AST) based on the syntax rules of RPAL. The class maintains a list of tokens (‘tokens’) and a stack of AST nodes (‘stack’).It provides methods to build the AST according to the RPAL grammar rules.
- The ‘get_AST()’ method initiates the parsing process and returns the AST built by the parser. Various methods (E(), Ew(), T(), Ta(), etc.) correspond to different non-terminal symbols in the grammar and handle the parsing logic accordingly. The ‘addStrings()’ method is a utility function to format the AST nodes for printing.
- The ‘AST_Node’ class represents a node in the Abstract Syntax Tree (AST) used by the RPAL parser. Its structure is as follows:
    - parent: A reference to the parent node in the AST. Initially set to null.
    - value: A string representing the value associated with the node. For example, an identifier, integer, string, etc.
    - type: An enum representing the type of node (e.g., identifier, integer, operator, etc.).
    - isStandartized: A boolean flag indicating whether the node has been standardized (it is a redundant one for RPLA_parser but it will be used in Standardizer to check whether a node is standardized or not already).
    - children: An ArrayList to store references to the child nodes of the current node. This allows for a variable number of children, making it suitable for representing the hierarchical structure of AST.

  There's a constructor that initializes the node with a token and its corresponding type.

  ‘AddChildren’ is a helper method to add child nodes to the current node. It appends the child node to the list of children and sets the parent of the child node to the current node.

- The ‘Parser_Tester’ class serves as the entry point for testing the RPAL parser.
    - It first creates a ‘LexAnalyser’ object and uses it to tokenize the input file, resulting in a list of tokens.

- Then, it creates an instance of the ‘RPAL_parser’ class, passing the token list to its constructor.
- It calls the ‘get_AST’ method of the parser to generate the Abstract Syntax Tree (AST) from the tokens.
- Finally, it prints the AST using the ‘printTree’ method of the parser, starting from the root node.

This class facilitates testing the RPAL parser by providing a way to tokenize input files, parse them, and visualize the resulting AST.

Standardizer

- The ‘Standardizer’ class contains methods to standardize the Abstract Syntax Tree (AST) generated by the RPAL parser. Here's an overview of each method:
  - ‘standardizeAST’: This method recursively traverses the AST and standardizes each node based on its type. It delegates the standardization process to specific methods for each node type.
  - Methods like ‘Let_Standardizer’, ‘Within_Standardizer’, ‘FcnForm_Standardizer’, ‘And_Standardizer’, ‘Where_Standardizer’, ‘Rec_Standardizer’, ‘At_Standardizer’, and ‘multi_param_function_Standardizer’ handle the standardization logic for each respective node type. Overall, the Standardizer class plays a crucial role in transforming the AST into a standardized form, which can simplify further processing or analysis of RPAL programs.
  - Here we have omitted the standardization for Binary and Unary operations, as well as some other operations like multi parameter function,etc. while keeping the AST structure of them same for ST also. These things are specifically handled in the CSE machine.
  - The ‘printTree’ method is used to print the standardized AST in a readable format.

  Overall, this class serves as a crucial component for transforming the AST into a standardized form, which can then be used for further processing or analysis.

- This ‘Standardizer_Tester’ class demonstrates how to use the ‘Standardizer’ class to standardize an Abstract Syntax Tree (AST) generated by an RPAL parser. Here's a breakdown of its functionality:
  - It initializes a ‘LexAnalyser’ object to tokenize the input RPAL code from a testing text file for example ‘file1’.
  - It uses the scanner method to tokenize the input code and obtains a list of tokens.

- It initializes an ‘RPAL_parser’ object and passes the list of tokens to generate the AST.
- It retrieves the AST using the ‘get_AST’ method.
- It creates an instance of the ‘Standardizer’ class.
- It calls the ‘standardizeAST’ method of the ‘Standardizer’ instance, passing the root node of the AST obtained in step 4.
- If the standardization process is successful, it prints the standardized AST using the ‘printTree’ method.

This tester class provides a convenient way to verify the functionality of the ‘Standardizer’ class by applying it to actual RPAL code. If any errors occur during the standardization process, it catches the exceptions and displays an error message, providing information about what went wrong.

CSE Machine

This class represents a machine for executing computations using Control Stack Environment (CSE) semantics. Here's a summary of what the class does:

Fields :

- ‘control’: An ArrayList of Symbol objects representing the control stack.
- ‘stack’: An ArrayList of Symbol objects representing the data stack.
- ‘environment’: An ArrayList of E objects representing the environments.

Constructor:

- Initializes the control, stack, and environment with the provided ArrayLists.

Methods:

- ‘execute()’: Executes the CSE machine according to the rules specified in the method.
- ‘apply_Unary_OP()’: Applies unary operations like negation and logical NOT.
- ‘apply_BinOp()’: Applies binary operations like addition, subtraction, multiplication, etc.
- ‘get_Tup_Value()’: Retrieves the values stored in a Tup object as a formatted string.
- ‘compute_Answer()’: Executes the CSE machine and returns the resulting value as a string.

Printing Methods:

- ‘printControl()’, ‘printStack()’, and ‘printEnvironment()’: These methods print the contents of the control stack, data stack, and environments, respectively.

This class provides functionality for executing computations using the CSE semantics, handling various types of symbols, and applying unary and binary operations.

The 'CSEMachineFactory' class is responsible for creating instances of the 'CSEMachine' class and generating the necessary symbols based on the standardized Abstract Syntax Tree that means Standardized Tree(ST) nodes. Here's a breakdown of what each method does:

- 'Constructor': Initializes the environment e0 and i,j variables to their respective values
- 'get_Symbol' Method: Returns a symbol corresponding to the given AST node. It handles unary and binary operators, gamma, tau, Ystar, and different types of operands such as identifiers, integers, strings, tuples, boolean literals, and dummy symbols.These symbols are used in our control stack,stack,environments for the easy handling and proper representation instead of AST nodes.So this is important conversion (Tree nodes → Symbol)
- 'get_B' Method: Creates and returns a B symbol containing symbols obtained from the pre-order traversal of the AST node.(this one used for conditional expressions representation)
- 'get_Lambda' Method: Creates and returns a Lambda symbol based on the given AST node. It sets the delta and identifiers for the lambda function.
- ‘preOrder_Traversal’ Method: Performs a pre-order traversal of the AST and constructs symbols accordingly. It handles lambda functions, delta functions, and other symbols.
- 'getDelta' Method: Creates and returns a Delta symbol based on the given AST node. It sets the symbols using the pre-order traversal.
- 'initialize_Control_Struct', 'initialize_Stack', and 'getEnvironment' Methods: Generate initial control, stack, and environment structures containing the root environment e0.
- 'initialize_CSEMachine' Method: Constructs and returns a 'CSEMachine' instance using the AST. It initializes the control, stack, and environment based on the AST structure.

This factory class encapsulates the logic for generating symbols and initializing a 'CSEMachine' instance from the given ST.

Symbols

These are the basic blocks in the CSE Machine implementation. Symbols are defined in a separate Package. Among those things B,Lambda,Delta ,Environment,Eta,Tup are some important ones.

B:

This one is used while computing the conditional expressions like B -> A|C .Inside B symbol object there is a array list structure to store the **condition which should evaluate first to take decision,** as a symbols

CSE Rule 8: Conditional

- Do not standardize → node. Instead, for B → E1 | E 2, generate

  $\delta_{then}\ \delta_{else}\ \beta\ B$, where

  δ_{then} = control structure for E1

  δ_{else} = control structure for E2

- B evaluated first, then β pops the stack, keeps one δ and discards the other.

B

Lambda:

Applicative expression	Control Structures
$(\lambda x.\lambda w.x+w)\ 5\ 6$	$\delta_0 = \gamma\ \gamma\ \lambda_1^x\ 5\ 6$ $\delta_1 = \lambda_2^w$ $\delta_2 = \gamma\ \gamma + x\ w$

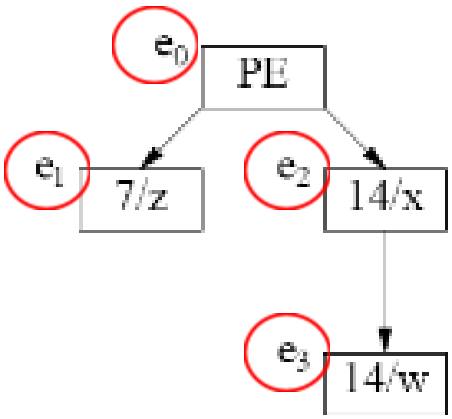
Lambda

Delta:

Applicative expression	Control Structures
$(\lambda x.\lambda w.x+w)\ 5\ 6$	$\delta_0 = \gamma\ \gamma\ \lambda_1^x\ 5\ 6$ $\delta_1 = \lambda_2^w$ $\delta_2 = \gamma\ \gamma + x\ w$

Delta

Environment:



Here we have 4 environments.each environment has structure to bind variables and their values and look up mechanism to search if the variable exists or not and if exists,the value of that. Each environment has a reference/pointer to the parent environment for back tracing the value of the variable if needed.

Exception

The 'CustomException' class provides several constructors to allow different ways of initializing the exception object. This flexibility is useful because it provides different levels of detail when throwing and handling exceptions.

- Default Constructor ('CustomException()'): Creates an instance of 'CustomException' without any message or cause.
- Message Constructor ('CustomException(String message)'): Creates an instance of 'CustomException' with the specified error message.
- Message and Cause Constructor ('CustomException(String message, Throwable cause)'): Creates an instance of 'CustomException' with the specified error message and the cause of the exception.
- Cause Constructor ('CustomException(Throwable cause)'): Creates an instance of 'CustomException' with the specified cause of the exception.

This structure allows us to handle exceptions with different levels of detail depending on the context in which they occur. Additionally, the '@SuppressWarnings("serial")' annotation suppresses compiler warnings about missing serialVersionUID fields, which is appropriate for custom exceptions that don't need to be serialized.

myrpal.java

'myrpal' class is the main entry point for the RPAL interpreter. It reads command-line arguments to determine the input file and any optional flags.

If there is at least one argument:

- It reads tokens from the input file using a lexical analyzer ('LexAnalyser').
- It parses the tokens into an abstract syntax tree (AST) using an RPAL parser ('RPAL_parser').
- If the '-ast' flag is provided, it prints the AST.
- If the '-st' flag is provided, it standardizes the AST using a 'Standardizer' and then prints the standardized AST.
- If no flags are provided, it constructs a CSE machine using a 'CSEMachineFactory' and executes it to get the final output, printing the result.

If there are no arguments or an invalid command is provided, it prints an error message.

This structure allows the interpreter to handle different scenarios based on the command-line arguments, such as printing ASTs, standardizing ASTs, or executing RPAL programs.

How to Run:

1. The whole package will be inside the zipped file.So first you need to extract the file.
2. Then in the terminal go to the extracted file directory
3. use '**make**' command if you are using Ubuntu or Linux terminal. So the make file will execute the compile command. or if you are using windows cmd use '**javac myrpal.java**' command to compile the interpreter(CSE machine).

   !!!!!!!!!!! Sometimes the '**make**' command may not work because of indentation issues in the makefile. Sometimes, when the makefile is created in Windows and then run in Ubuntu, there can be problems with spaces and tabs. If this issue occurs, try correcting the indentation in the makefile. Make sure that all indentation is done using tabs instead of spaces. This should help resolve the problem and allow the make command to work properly !!!!!!!!!!!!!!!!!

4. Now you can use commands like below ones to run the program
    a. **java myrpal test1**
    b. **java myrpal test1 -ast**
    c. **java myrpal test1 -st**

```
C:\Users\Owner>cd "C:\Users\Owner\Desktop\Semester 4\Programming Languages\Final Project"

C:\Users\Owner\Desktop\Semester 4\Programming Languages\Final Project>javac myrpal.java

C:\Users\Owner\Desktop\Semester 4\Programming Languages\Final Project>java myrpal test1
15

C:\Users\Owner\Desktop\Semester 4\Programming Languages\Final Project>java myrpal test1 -ast
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>
```