Designing an Interpreter for RPAL language

Group-23

CS 3513 - Programming Languages

Author Note

A translator is an algorithm that converts source programs into equivalent target programs. An interpreter is an algorithm that simulates the execution of programs written in a given source language. RPAL, the Right-reference Pedagogic Algorithmic Language, is a functional subset of PAL with an implementation on SourceForge.

In this project, we are implementing RPAL-Interpreter by using C++ Programming language. Key methods included here are, building a Lexical Analyzer, and Parser, generating an Abstract Syntax Tree, and Standardizing Tree. Finally, a CSE-Machine to evaluate the input program.

Group Members:
NIMASHA K.L.A.A. 200425K
PRABASHWARA M.A.R. 200472B
WICKRAMARATHNA H.K.G.V.L. 200710F

Designing an Interpreter for RPAL language

Table of Content

First paragraph: Complete departmental and institutional affiliation

Second paragraph: Changes in affiliation (if any)

Third paragraph: Acknowledgments, funding sources, special circumstances

Fourth paragraph: Contact information (mailing address and e-mail)

Designing an Interpreter for RPAL language

Introduction

The objective of this project is to implement a lexical Analyzer and a Parser for RPAL-Language. Lexical Analyzer converts the given Source program into a Sequence of Tokens. Then remove all unwanted tokens including white spaces, comments, etc. The sequence of recognized keywords is the input for the Parser. Parser is a Syntax Analyzer used to determine whether the sequence of tokens is syntactically correct or not. To do that, Parser should generate a Syntax tree, mainly bottom-up is the rule used to generate this tree. Contextual Constraint Analysis is the method we used to convert the syntax tree to target code, and is done by using CSE-Machine. Before, executing CSE-Machine, generated Abstract Syntax Tree is converted into a Standardized tree. This is done in the standardizer. Evaluation, building Environment, and lookup variables are correlated with each other and are done in the CSE-Machine, Environment_generator (Environment abstract class), and CSElement..

Keywords: RPAL-Language, Scanner, Screener, Lexical Analyzer, Parser, CSE-Machine, Syntax tree

The implementation uses C++ version 13.1.0 and VSCode IDE for development.

Designing an Interpreter for RPAL language

Executing The Program.

GetFiles.cpp, generates the directory that we have generated in the **tests folder.** “tests folder” is the folder in which we include all the programs, which are written in the rpal-functional language.

Testing the program

Change the filename variable to the appropriate program file name, and execute main.cpp will generate “ `ABSTRACT SYNTAX TREE` ”, “ `STANDARDIZED TREE` ”, and “ `CSE Machine Structure` ” and terminate the program successfully.

Build, execute, and test with a single command

To build, execute, and test programs in a single command run, main.cpp

Project Structure

The following are the Hierarchy of the project.

- ☐ The **RPAL-Intepreter** is the directory that contains the nested namespace of the classes of the project.

Here seven tasks are defined. The Cpp classes are decomposed into seven abstract classes and eight inheritance classes.

- ☐ LexicalAnalyzer: Convert Source language into a Sequence of tokens. Remove unwanted tokens (spaces, comments), Recognize keywords, Merge/simplify tokens, Prepare token list for next phase (parser)
- ☐ Parser: Group the tokens into the correct syntactic structures such as Expressions, statements, procedures, functions, modules, etc. allied Tree class to Build a “Abstract Syntax Tree” in a Bottom-up manner. Use a stack of trees to generate tree hierarchy.

Designing an Interpreter for RPAL language

- ☐ Tree: include class to create AST
- ☐ Standardizer: Include class to create ST
- ☐ CSEMachine: Include class to generate & build control structures, initialize, run, and process those generated control structures with the help of Environment and CSElement
- ☐ Environment: Identified variables and their parents, children, siblings, and the corresponding values.
- ☐ CSElement: Include class to identify, bound variables, an array of elements, type of a variable

Classes and Functions

➔ main.cpp

Contains the main function of the program. Accept the filename from the terminal with the help of sending arguments to a program and use LexicalAnalyzer and Parser classes to build and standardize the tree. Then, use CSEMachine class to evaluate the standardized tree.

The provided code performs the following tasks:

1. Enums for Node Types and Tokens: It defines two enums - NodeType and Token. The NodeType enum represents different types of nodes in the Abstract Syntax Tree (AST). The Token enum represents different types of tokens generated by the Lexical Analyzer.

2. File Input and Lexical Analysis: The program reads the contents of a file named "add" from the "tests" directory. It then passes the content to the Lexical Analyzer (LexicalAnalyzer) to generate tokens based on the RPAL language's lexical rules.

3. Parsing and Building AST: The program creates a parser (Parser) and invokes its E() method to build the Abstract Syntax Tree (AST) from the tokens generated by

the Lexical Analyzer. It uses recursive descent parsing to parse the RPAL language expressions.

4. Printing AST and Standardized Tree: It then prints the generated AST and the standardized tree by calling the Tree::prettyPrint() method. The standardized tree ensures consistent representation of the AST.

5. CSE Machine Initialization: The program creates an instance of CSEMachine and passes the top-level tree of the AST to it. The CSEMachine is designed to interpret and execute RPAL language programs.

6. Termination: The program outputs a message indicating successful termination and waits for user input (commented out) before exiting.

   In summary, this program reads an RPAL language program from a file, performs lexical analysis, constructs the Abstract Syntax Tree (AST), prints the AST, standardizes the tree, initializes the CSEMachine, and finally executes the RPAL program represented by the AST.

➔ **LexicalAnalyzer.h/cpp**

This is the class that is responsible for scanning and screening the RPAL source code to output a sequence of tokens for the parser. The tokens are created according to the **RPAL's LEXICON [1].**

Methods:

public:

- ➢ LexicalAnalyzer(string Source);
  - The constructor takes the source code as a string input and initializes the member variables.

- ➢ ~LexicalAnalyzer();

- The destructor is used to autogenerate the token types.

➢ getChar();

- This function updates the charNext and charClass members by extracting the next character from the source code and determining its class using getcharClass().

➢ static int getcharClass(char symbol);

- This function returns the class code of the given character based on the rules specified in the code. The class codes are used to categorize characters into different token types.

➢ void addCharLexeme();

- This function appends the current character (charNext) to the lexeme member variable.

➢ int processNext();

- This function performs the main tokenization process. It reads characters from the source code, identifies tokens, and returns the corresponding token code.

➢ string getString(int code);

- This function returns a string representation of the token code provided as input.

Designing an Interpreter for RPAL language

➔ **Parser.h/cpp**

This is the class that is responsible for checking whether the token sequence is syntactically correct or not. Group the tokens into the correct syntactic structures. Such as Expressions, statements, procedures, functions, and modules. Build a “syntax tree”, bottom-up, as the. **RPAL's Phrase Structure Grammar [2]** rules are used. It utilizes a recursive descent parsing algorithm, an efficient method for processing the code's grammar. The class takes a pointer to a LexicalAnalyzer object as input, which is instrumental in tokenizing the input source code.

Method:

public:

- Parser(LexicalAnalyzer *mLexNew) ;
    - The Parser class maintains a stack of Tree objects to build an abstract syntax tree (AST) during the parsing process. The constructor initializes the Parser object and processes the first token from the LexicalAnalyzer.

- Tree makeTree(string mVal, int popCount);
    - The makeTree function is used to create nodes in the AST and handle the parsing rules for different non-terminal symbols.

- Tree* getTopTree();
    - finds the top element of the stack.

- void readToken(string mToken);

- readToken is used to read the next token from the LexicalAnalyzer and checks if it matches the expected token. If not, it raises an error.

➢ static bool isKeyword(string mTok);

- readToken is used to read the next token from the LexicalAnalyzer and checks if it matches the expected token. If not, it raises an error.

➢ Methods for productions in RPAL grammar:

- These functions correspond to the non-terminal symbols and recursively parse different parts of the source code to build the AST.
- void E(); Ew(); T(); Ta(); Tc(); B(); Bt(); Bs(); Bp(); A(); At(); Af(); Ap(); R(); Rn(); D(); Da(); Dr(); Db(); Vb(); Vl();

➔ **Tree.h/cpp**

This class represents nodes in the Abstract Syntax Tree (AST) that is constructed during the parsing process. Each node in the AST possesses a value (e.g., token value) and a type (e.g., GRAMMAR_RULE, ID_NAME), along with pointers to its child and sibling nodes in the tree. The class provides essential functions to create new nodes, set their values and types, and manage the tree structure, ensuring the organized representation of the program's hierarchical structure. Through the Tree class, the project can efficiently construct and manage the AST, enabling a comprehensive and structured representation of the RPAL language source code for further evaluation and processing.

Method:
public;

➢ Tree(string mVal, int mType, Tree *mChild, Tree *mSib);

➢ Tree(Tree *copyTree);

- Tree class, represents nodes in the abstract syntax tree (AST).It contains two constructors, one for creating a new node with

specified values, and another for creating a copy of an existing node.

- ➢ void operator=(Tree t);
    - initializes childNodes and sibling nodes.

➔ **Standardizer.h/cpp**

standardizes the AST by employing private helper methods to handle specific transformations for different nodes. The methods ensure a uniform representation of the program structure, promoting code organization and modularity. The standardized AST is then ready for further evaluation and processing.

Method:

public;

- ➢ static Tree* standardizeTree(Tree *tree);
    - initiates the standardization process.It takes a pointer to the root of the AST (tree) as input and returns a pointer to the standardized version of the AST.The method processes the AST recursively, handling different types of nodes based on their nodeValue. It calls the appropriate private helper methods to perform the required transformations for specific node types

- ➢ static void standardizeLet(Tree *tree);
    - handles the "let" node in the AST and standardizes it into a "gamma" node with a nested "lambda" node.The "let" node is transformed into a "gamma" node, where the first child represents the variable (bound by "=") and the second child represents the expression (bound by "in").

- ➢ static void standardizeWithin(Tree *tree);
    - handles the "within" node in the AST and standardizes it into a "=" node with a nested "lambda" node.The "within" node is transformed into a "=" node, where the first child represents the expression

within the "within" clause and the second child represents the expression within the "let" clause.

- static void standardizeFncForm(Tree *tree);
    - handles the "function_form" node in the AST and standardizes it into nested "lambda" nodes.The method converts a chain of "function_form" nodes into a chain of "lambda" nodes, with each "lambda" node representing a variable binding.

- static void standardizeLambda(Tree *tree);
    - handles the "lambda" node in the AST and standardizes it by converting multiple consecutive "lambda" nodes into a single "lambda" node with multiple variables.

- static void standardizeInfix(Tree *tree);
    - handles the "lambda" node in the AST and standardizes it by converting multiple consecutive "lambda" nodes into a single "lambda" node with multiple variables.

- static void standardizeAnd(Tree *tree);
    - handles the "and" node in the AST and standardizes it into a "," (comma) node with a "tau" node.The "and" node is transformed into a "," node, and each pair of variable bindings is separated into a "tau" node.

- static void standardizeWhere(Tree *tree);
    - handles the "where" node in the AST and standardizes it into a "gamma" node with a nested "lambda" node.The "where" node is transformed into a "gamma" node, where the first child represents the expression from the "where" clause, and the second child represents the expression from the "let" clause.

- static void standardizeRec(Tree *tree);
    - handles the "rec" node in the AST and standardizes it into a "=" node with a recursive function.The "rec" node is transformed into a "=" node, where the left-hand side is a recursive function, and the right-hand side represents the recursive call (as a "<Y*>" node).

➔ **CSEMachine.h/cpp**

The CSE machine employs the 13 predefined rules to evaluate the ST and execute the RPAL language program. It interprets the standardized code and generates the desired output, reflecting the logic and computations within the program.

Method:

public;

- string getTypeString(int type);
    - takes an integer type as input and returns a string representing the type of the node in the Abstract Syntax Tree (AST).It checks the value of type and returns a corresponding string representation for "STRING", "INTEGER", "IDENT" (identifier), "RULE", or "UNKNOWN!".

- CSEMachine(Tree* tree);
    - Constructor for the CSEMachine class.It initializes the CSEMachine with the provided AST tree. It sets some internal control flags and then proceeds to build control structures, initialize the machine, and run the CSEMachine.

- void buildControlStructures(Tree* tree, int index);

- recursively builds the control structures based on the given AST tree and its corresponding index.It creates CSElement objects for the AST nodes and organizes them into control structures (nested vectors of CSElements).It distinguishes between different types of nodes, such as "lambda" (representing functions), "tau" (representing tuples), and others.

➢ void printControlStructures();

- prints the control structures built during the process of interpreting the AST.It iterates through the controls and prints the elements in each control structure, handling special cases like "lambda" and "tau" nodes.

➢ void pushDeltaOnStack(int delta);

- used to push a new environment (CSElement with the value "env") onto the left and right stacks of the CSEMachine. It copies the CSElements from the specified control structure (with index delta) to the left stack.

➢ void initializeCSEMachine();

- Initializes the CSEMachine by pushing the first delta (environment) onto the stack and setting up the initial environment.

➢ void printCSE();

- Prints the contents of the left and right stacks in the CSEMachine.

➢ CSElement* lookupVar(string name);

- used to look up a variable by its name in the current environment and its parent environments.It returns a pointer to the corresponding CSElement if the variable is found, or NULL if it's not present.

- ➢ void runCSEMachine();
    - This function is the main loop of the CSEMachine, responsible for processing the CSEMachine until the left stack is empty.

- ➢ void processCSEMachine();
    - performs one step of the CSEMachine's execution. It checks the top element of the left stack and performs operations accordingly, such as arithmetic operations or logical "not".

➔ **CSElement.h/cpp**

The CSElement class functions as an environment creator in the presence of lambdas.

Method:

public;

- ➢ CSElement(string mVal) and CSElement(string mVal, int mIndex, string mPar1, int mparEnv);
    - CSElement class offers two distinct constructors aimed at creating CSElements under different circumstances. The first constructor is designed for the initial case, where the indexing starts at 0. The second constructor facilitates the generation of subsequent CSElement environments by incorporating relevant indexes and associated information.

➔ **Environment**

The Environment class serves as a representation of the relevant environments during the evaluation of a program using the CSE machine.

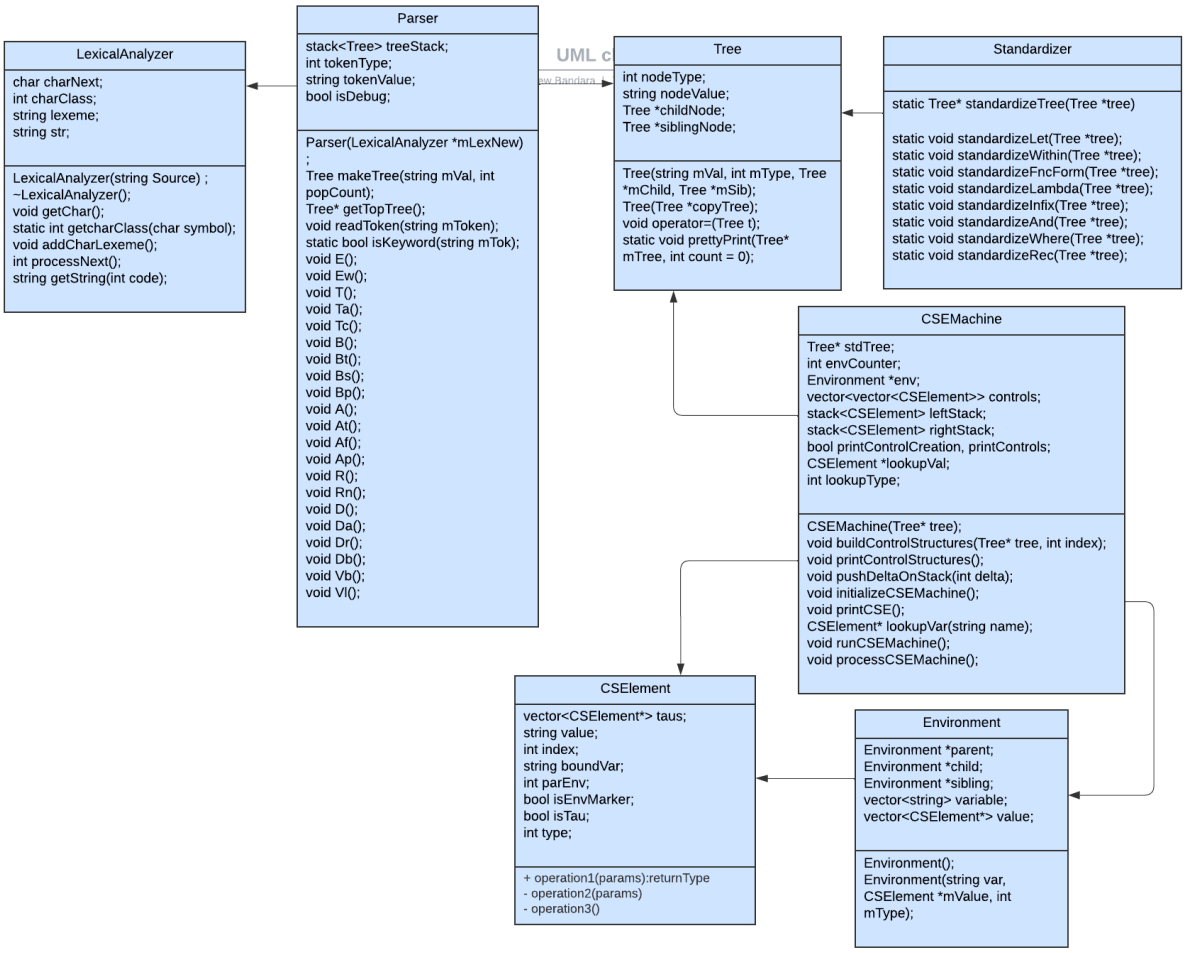
Method:

public;

- ➢ Environment(string var, CSElement *mValue, int mType):

- The constructor sets the parent, child, and sibling pointers to NULL to indicate that they are not yet linked to other environment objects. It adds the provided var to the variable vector and mValue to the value vector, creating the initial environment with a single variable-value pair.

Appendix: Class diagram for Abstract Class



Designing an Interpreter for RPAL language

References

[1] RPAL's LEXICON

[2] RPAL's Phrase Structure Grammar

[3] OOP Concepts

[4] RPAL Documentation