CS 3513 - Programming Languages

Project Report

Group 129

DE SILVA D.A.V	*210099V*
KULATHUNGA K.M.L.I	*210307K*

01. Problem Description

It is required to implement a lexical analyzer and a parser for the RPAL language referring to the lexical rules and grammar rules. The output of the parser should be the Abstract Syntax Tree for the given program. Then implement an algorithm to convert the Abstract Syntax Tree to a Standardized Tree (ST) and implement the CSE machine. 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.

02. Solution

1. Tokenization:

   Begin with a robust lexical analysis phase using a hardcoded scanner. Implement regular expressions to recognize and categorize tokens such as keywords, identifiers, literals, and operators. This process involves cleaning the input by removing comments and unnecessary whitespace, producing an array of tokens encapsulating each token's type and value.

2. Parsing:

   Implement a recursive descent parser to systematically traverse the token sequence and construct an Abstract Syntax Tree (AST). Focus on a subset of RPAL's grammar, handling simple expressions and basic function definitions. Ensure the parser effectively detects and reports syntax errors, providing clear feedback if the input does not conform to the RPAL language syntax.

3. AST Evaluation:

   Develop an interpreter capable of evaluating expressions by traversing the AST. Initially, supports basic arithmetic operations and function calls. This stage aims to execute RPAL programs by interpreting the structured AST representation generated during parsing.

4. Standardize Tree (ST):

   Extend functionality to transform the AST into a Standardized Tree (ST). Implement transformation rules, through a pre-order traversal of the AST, to simplify constructs such as let-bindings and lambda functions. This step enhances the efficiency and clarity of subsequent program transformations and optimizations.

5. Exception Handling:

   Integrate comprehensive error-handling mechanisms throughout the lexer and parser components. Ensure robust detection and reporting of syntax errors, providing informative messages to aid users in correcting input that deviates from the expected RPAL language syntax.

6. Control Structure Generation:

   Utilize a pre-order traversal of the AST to generate control structures systematically. Implement mechanisms, involving a First-In-First-Out (FIFO) queue, to manage and construct control flow constructs as specified by the transformed AST. This ensures the correct execution flow of RPAL programs based on their structured representation.

03. General functionalities of modules

1. **Lexer.cpp**: Responsible for converting the source code into a sequence of tokens that can be easily processed by subsequent stages of the compilation or interpretation process.
2. **Parser.cpp**: Takes the sequence of tokens produced by the lexer.cpp and constructs an Abstract Syntax Tree, which represents the hierarchical structure of the source code.
3. **Standardized_Tree.cpp**: Responsible for standardizing a tree representation of a programming language's abstract syntax tree.
4. **CSE_Machine.cpp**: Handles the execution of operations and the management of the control structure environment machine.
5. **Environment.cpp**: Managing variable scope ensuring proper variable visibility.
6. **Controller.cpp**: Handles the control structures.

4. Detailed functionalities

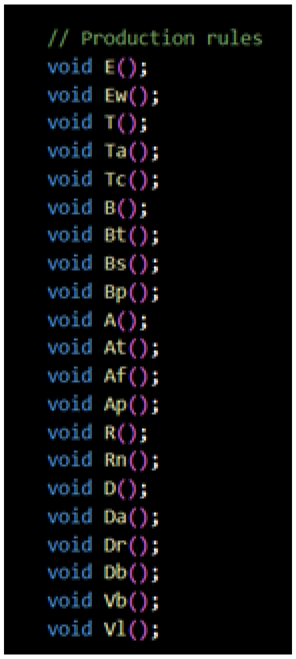
1) Tokenization and Lexical Analyzer

- The **lexer.cpp** file provides a comprehensive implementation of the lexical analyzer and tokenizer for the RPAL language.
- It defines methods to reset the lexer, classify characters, extract different types of tokens, and return the next token from the source file.

- This lexer will be used to tokenize the input source code, facilitating further parsing and analysis.
- These methods extract various types of tokens from the source file:
    - **tokenIdentifier**: Extracts identifier tokens.
    - **tokenInteger**: Extracts integer tokens.
    - **tokenStrings**: Extracts string tokens, handling escape sequences.
    - **tokenSpaces**: Extracts whitespace characters, updating line and character counts.
    - **tokenComment**: Extracts comment tokens.
    - **tokenOperator**: Extracts operator tokens.
- **getNextToken**: The main function that returns the next token from the source file. It identifies the type of the next token based on the next character and calls the appropriate extraction method. The token is then returned with its type and value.

2) Parsing and AST generation

- The **parser.cpp** defines various methods for parsing a sequence of tokens provided by a lexer.
- The parser follows a top-down parsing recursive approach.
- The goal is to create a parse tree that represents the structure of the input program.
- These methods are used for parsing and evaluation.
    - **parse()**: The main parsing function that reads tokens and constructs the AST.
    - **evaluateProg()**: Parses the input, converts it to a standardized tree, and evaluates it using the CSE machine.
- These methods are used for Tree Printing:
    - **printAST()**: Parses the input and prints the Abstract Syntax Tree (AST) if parsing is successful.
    - **printST()**: Parses the input and prints the Standardized Tree.
    - **printTree(treeNode* topNode, int numDots)**: Helper function to print the tree structure recursively.
- The code contains a series of functions that correspond to the different non-terminals in the RPAL grammar. Each parsing function is responsible for handling the grammar rules associated with a specific non-terminal.

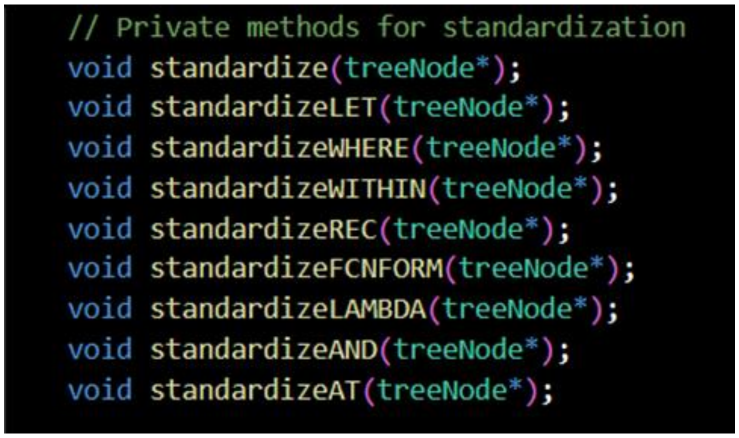


```
// Production rules
void E();
void Ew();
void T();
void Ta();
void Tc();
void B();
void Bt();
void Bs();
void Bp();
void A();
void At();
void Af();
void Ap();
void R();
void Rn();
void D();
void Da();
void Dr();
void Db();
void Vb();
void Vl();
```

- These are some additional helper functions that are used in the parser.
    - ❖ **isKeyword(string val)**: Checks if a string is a keyword.
    - ❖ **read(string tokStr)**: Reads the next token and checks if it matches the expected token string.
    - ❖ **buildTree()**: Builds a tree node and manages the tree stack.
    - ❖ **to_s(treeNode* node)**: Converts a tree node to its string representation.

3) Abstract Syntax Tree (AST) to Standardized Tree (ST)

- The **Standardized_Tree.cpp** contains methods to transform various types of nodes into a standardized form.
- Each standardization function is designed to handle a specific type of node, adjusting the node's type and rearranging pointers to fit a predefined structure.

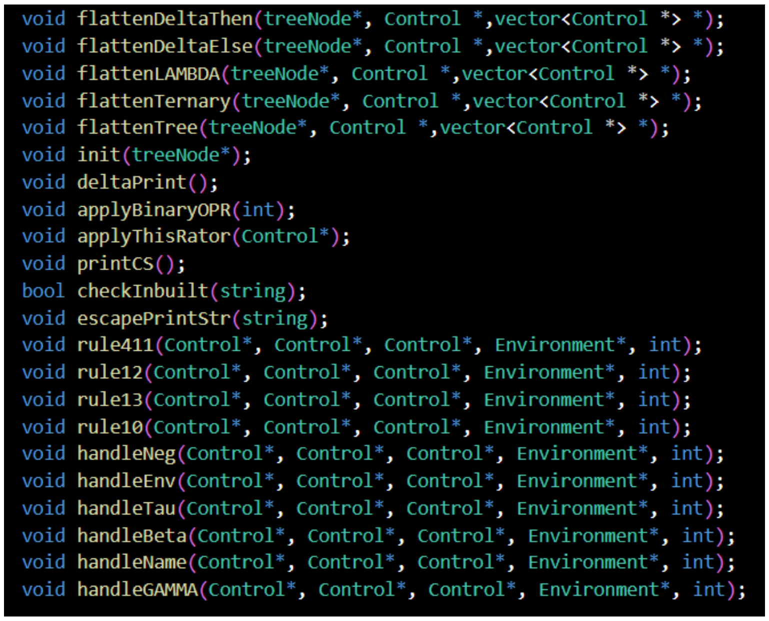


```
// Private methods for standardization
void standardize(treeNode*);
void standardizeLET(treeNode*);
void standardizeWHERE(treeNode*);
void standardizeWITHIN(treeNode*);
void standardizeREC(treeNode*);
void standardizeFCNFORM(treeNode*);
void standardizeLAMBDA(treeNode*);
void standardizeAND(treeNode*);
void standardizeAT(treeNode*);
```

- The recursive **standardize(treeNode* topNode)** function ensures that the entire tree is processed, applying the appropriate transformations as needed. Here's how it works.
  - ❖ Recursively standardizes the entire tree starting from the given top node.
  - ❖ Calls the appropriate standardization function based on the node type.
  - ❖ Recursively processes child and sibling nodes before standardizing the current node.

4)CSE Machine

- The CSE machine's primary functions include initializing the environment, handling various operations, and managing the control flow and data within the environment.
- Some methods used in CSE machine:
  - ❖ **CSE_Machine(treeNode* topNode)**: Initializes the machine with a specified top node of a tree, which represents the root of the program structure. This node is stored for further processing.
  - ❖ **CSE_Machine()**: Sets up a default instance of the machine by initializing environment variables, creating a new environment, and setting up necessary control structures such as environment stack and execution stack.
  - ❖ **applyBinaryOPR()**: Executes binary operations such as addition, subtraction, multiplication, division, and logical comparisons.
  - ❖ **handleName()**: Manages the lookup and binding of names within the environment. It checks if a name corresponds to an inbuilt function or a variable stored in the current environment. If the name is not found, it handles the error accordingly.
- And many more methods are used to implement CSE machine rules.



```
void flattenDeltaThen(treeNode*, Control *,vector<Control *> *);
void flattenDeltaElse(treeNode*, Control *,vector<Control *> *);
void flattenLAMBDA(treeNode*, Control *,vector<Control *> *);
void flattenTernary(treeNode*, Control *,vector<Control *> *);
void flattenTree(treeNode*, Control *,vector<Control *> *);
void init(treeNode*);
void deltaPrint();
void applyBinaryOPR(int);
void applyThisRator(Control*);
void printCS();
bool checkInbuilt(string);
void escapePrintStr(string);
void rule411(Control*, Control*, Control*, Environment*, int);
void rule12(Control*, Control*, Control*, Environment*, int);
void rule13(Control*, Control*, Control*, Environment*, int);
void rule10(Control*, Control*, Control*, Environment*, int);
void handleNeg(Control*, Control*, Control*, Environment*, int);
void handleEnv(Control*, Control*, Control*, Environment*, int);
void handleTau(Control*, Control*, Control*, Environment*, int);
void handleBeta(Control*, Control*, Control*, Environment*, int);
void handleName(Control*, Control*, Control*, Environment*, int);
void handleGAMMA(Control*, Control*, Control*, Environment*, int);
```

5) Environment

- The **Environment.cpp** provides an implementation of an environment for symbol lookup and management within the given context.
- It defines methods to initialize environment instances, manage parent-child relationships between environments, look for symbols in the environment's symbol table, and print the contents of the environment.

6) Controller

- The **Controller.cpp** defines various methods for managing and manipulating nodes of different types within a control structure.
- It includes constructors for initializing different types of control nodes, a method for converting control nodes to string representations, and functions for adding new control nodes to the structure.
- Here are some important functions used in this module.
  - ❖ **toStr()**: Converts the current control node into its corresponding string representation based on its type.
  - ❖ **std_print()**: Prints the string representation of a DELTA-type Control object followed by its nested control structures.
  - ❖ **addCtrl()**: Adds a new control node to the control structure based on the type of the input tree node. Handles various node types by creating corresponding Control objects.

5. Running and Testing

- To compile the program, use the command **$ make**. This will generate an executable named **"myrpal"**. Once compiled, you can run the program with different options:

❖ .\myrpal <filename>	: generate the output
❖ .\myrpal <filename> -ast	: generates the abstract syntax tree
❖ .\myrpal <filename> -st	: generates the standardized tree

- Replace <filename> with the actual name of the file you want to process.
- Some test cases are available in the same directory. (q1-q8: Lab exercises, test.txt: the given example in the project description)