CS 3513

Programming Languages

Programming Project 01 Report

RPAL Compiler: Lexical Analyzer, Parser, Standardize Tree Conversion, and CSE Machine Implementation

University of Moratuwa

Group Number: 99

Group Members:

Akshiya R.	210025T
Praveenasarma B.	210493A

Department of Computer Science & Engineering

Table of Contents

1. Introduction

The RPAL programming language, a derivative of PAL originating from MIT in the 1970s, serves as a foundational tool for educational and exploratory programming endeavors. To enhance accessibility and comprehension of RPAL, this project embarks on the development of a comprehensive toolchain encompassing a lexical analyzer, parser, Abstract Syntax Tree (AST) construction, Standardized Tree (ST) transformation, and a specialized interpreter known as the CSE machine. Implemented in Java, this project aims to create a platform that makes it easier to handle RPAL programs. It is designed to help people learn and understand RPAL better by providing tools that simplify RPAL programming. The goal is to encourage people to explore RPAL's features and learn more about programming.

2. Project Scope

The project encompasses the development of several crucial components for processing RPAL programs effectively. These components include:

- Lexical Analyzer: The primary objective is to design and implement a lexer capable of scanning RPAL programs. This lexer will be adept at identifying tokens and adhering to the specified lexical rules outlined in the project requirements.

- Parser: A recursive descent parser will be developed to parse the token stream generated by the lexer. This parser will closely follow the grammar rules provided, ensuring accurate interpretation of the RPAL program's structure.

- Abstract Syntax Tree (AST): A key component of the project is the construction of an Abstract Syntax Tree (AST). This data structure will represent the hierarchical structure of the parsed RPAL program, providing a foundation for subsequent processing.

- Standardize Tree (ST): An algorithm will be devised to transform the AST into a Standardized Tree (ST). This transformation is crucial for optimizing code generation, ensuring efficient execution of RPAL programs.

CSE Machine: A significant aspect of the project involves the development of an interpreter for the CSE (Control-Stack-Environment) machine. This interpreter will execute the ST generated in the previous step, ultimately producing the final output of the RPAL program.

The project aims to provide a comprehensive toolchain for processing RPAL programs. Each component plays a vital role in the overall process, from lexical analysis to code execution, ensuring robustness and efficiency in handling RPAL code.

3. Implementation Approach

The implementation of the project will utilize Java due to its versatility and reliability. The initial phase will focus on the development of a lexer tasked with tokenizing the input RPAL program while adhering to the prescribed lexical rules in RPAL_Lex.pdf. Subsequently, a recursive descent parser will be constructed to recognize the program's grammar outlined in RPAL_Grammar.pdf and construct the corresponding AST. The AST will be further transformed into a Standardize Tree using an algorithm that eliminates redundancy and optimizes the structure for efficient code generation. Finally, the CSE machine will interpret the standardized tree, executing the RPAL program and generating the desired output.

4. Key Concepts in RPAL Constructs

The RPAL language system is designed to construct and handle various fundamental principles essential to functional programming. These concepts are as follows:

Operators:
RPAL supports a set of operators which facilitate the manipulation and evaluation of data. These include arithmetic operations for integers (+, -, *, /, **), truth-value operations (or, &, not, eq, ne), and string operations (eq, ne, Stem S, Stern S, Conc S T).

Function Definitions:
RPAL allows the creation of user-defined functions following a lambda calculus style. Functions, treated as first-class objects, can be assigned to variables or passed as arguments to other functions.

Constant Definitions:
The language supports the declaration of constants, allowing users to define fixed values that remain unchanged throughout program execution.

Conditional Expressions:
RPAL supports conditional expressions via the "if-then-else" construct. This allows programmers to make decisions based on truth values and execute different code blocks accordingly.

Function Application:
RPAL facilitates the application of functions to arguments, a pivotal aspect of functional programming where functions are applied to inputs to produce results.

Recursion:
Recursion is indeed a powerful technique in functional programming, and RPAL supports it, allowing functions to call themselves within their definitions. It offers elegant solutions to iterative problems. In RPAL, it is worth noting that there are no loops; only recursion is used to iterate through tasks or processes.

Data Types:
RPAL supports various data types, including integers (for whole numbers), truth-values (Boolean for binary logic), strings (for sequences of characters), tuples (collections of elements of different data types), and functions (first-class data types that can be used like any other data).
These data types provide the foundation for creating complex and flexible programs in RPAL.

5. Structure of the program

5.1. Parser

This part of the program implements a lexical analyzer and parser for a programming language. It reads input from a file, tokenizes it, and constructs an Abstract Syntax Tree (AST) based on the grammar rules of the language. The lexical analyzer categorizes tokens into different types such as identifiers, integers, strings, operators, reserved words, spaces, and punctuations. The parser then utilizes these tokens to build the AST by following the rules defined in the grammar of the language. Finally, the program prints the generated AST, providing a hierarchical representation of the parsed code structure. Overall, the program enables the analysis and understanding of code written in the specified programming language.

- Lexical class:
  The Lexical class serves as a lexical analyzer, responsible for reading an input file, tokenizing its contents, and providing tokens to the parser.
  - Lexical(String fileName): Constructor that initializes the lexical analyzer with the input file to read. It also initializes collections for allowed letters, digits, operators, spaces, punctuations, and reserved tokens.
  - private String readFile(): Reads a line from the input file.
  - private String parse(): Parses the current line and selects a token to return. It iterates through each character in the input file and identifies tokens based on certain rules. Here are the token types it recognizes:
    - IDENTIFIER: Sequences of letters, digits, or underscores that start with a letter.
    - INTEGER: Sequences of digits.
    - COMMENT: Lines starting with // are considered comments and are skipped.
    - OPERATOR: Various operators defined in the operators collection.
    - STRING: Sequences enclosed in single quotes, possibly containing escape characters.
    - SPACES: Spaces, tabs, or newline characters are skipped.
    - PUNCTUATIONS: Various punctuation symbols defined in the punctuation collection.
  - public String getToken(): Gets a new token for parsing.
  - public String getType(): This method returns the type of the last returned token. If there are no more tokens left in the file, it returns "<EOF>" to indicate the end of the file.

- Node class:
  The Node class represents a node in the Abstract Syntax Tree (AST). The tree is in the FIRST CHILD NEXT SIBLING form, where each node has a child and a sibling.
  - setToken(String token): Sets the token value for the node.
  - setChild(Node child): Sets the first child node.
  - setSibling(Node sibling): Sets the next sibling node.

    - getToken(): Gets the token value of the node.
    - getChild(): Gets the first child node.
    - getSibling(): Gets the next sibling node.

- Parser class:
  The Parser class is responsible for parsing the input tokens generated by the Lexical class and constructing an Abstract Syntax Tree (AST) based on the grammar rules of the language. It implements recursive descent parsing to build the AST based on the RPAL grammar.

    - Parser(String fileName): Constructor that initializes the parser with a file name, which is passed to the Lexical class constructor to create a lexical analyzer.
    - private void read(String s): Reads and compares the token with the expected value.
    - The Parser class contains a series of parsing methods, each corresponding to a non-terminal symbol in the grammar. These methods recursively parse the input tokens according to the grammar rules. Methods corresponding to grammar rules are:
        - E(),Ew(),T(),Ta(), Tc(), B(), Bt(), Bs(), Bp(), A(), At(), Af(), Ap(), R(), Rn(), D(), Da(), Dr(), Db(), Vb(), Vl()
    - private void buildTree(String root, int n): Builds the abstract syntax tree by popping the specified number of trees from the stack and joining them with the root node specified.
    - private void preOrder(Node n, int level, StringBuilder AST): Traverses the abstract syntax tree in pre-order and constructs a string representation of the tree.
    - private String printAST(): Prints the generated abstract syntax tree.
    - public String startParsing(): Starts parsing and returns the string representation of the abstract syntax tree. It also handles the end of file condition (<EOF>) to ensure that parsing is completed before generating the AST.
    - public Node getAST(): Returns the root node of the AST constructed during parsing.

These functions collectively implement the lexical analysis and parsing of a programming language, generating an abstract syntax tree as output.

5.2. Vertex

This program defines a class called 'Vertex', which represents a node in an abstract syntax tree (AST) and semantic tree (ST). Each vertex has a label indicating its type and optionally a value. The main functionalities of this program and each function are as follows:

1. Vertex Class:
   - Fields:
     - children: ArrayList to store child vertices of the current vertex.
     - parent: Reference to the parent vertex. It represents the hierarchical relationship in the tree structure.
     - label: String representing the type of the vertex.
     - value: String representing the value of the vertex.
   - Constructors:
     - Vertex(String label): Constructs a vertex with the given label and initializes an empty list of children.
     - Vertex(String label, String value): Constructs a vertex with the given label and value, and initializes an empty list of children.
   - Methods:
     - copy(): Creates a deep copy of the vertex and its subtree. It recursively copies all the children of the current vertex, creating a new vertex with the same label and value.
     - getParent(): Returns the parent vertex.
     - getLabel(): Returns the label of the vertex.
     - getValue(): Returns the value of the vertex.
     - getNumChild(): Returns the number of children of the vertex.
     - hasChildren(int n): Checks if the vertex has exactly 'n' children.
     - isLabel(String label): Checks if the vertex has the given label.
     - getChild(int i): Returns the i-th child vertex
     - forEachChild(Consumer<? super Vertex> action): : Iterates over each child vertex of the current vertex and applies the specified 'action'.

        - setLabel(String label): Sets the label of the vertex.
        - clearChildren(): Removes all children of the vertex.
        - addChild(Vertex child): Adds a child vertex to the list of children and sets its parent reference.

Overall, this program provides a flexible and reusable implementation for managing vertices in ASTs and STs, allowing for easy traversal and manipulation of tree structures.

5.3. depthOfVertex

This program defines a class called 'depthOfVertex', which extends the 'Vertex' class and adds the functionality of tracking the depth of each vertex in the abstract syntax tree (AST).

1. depthOfVertex Class:

- Fields:
    - depth: An integer representing the depth of the vertex in the tree. The depth value represents the level at which the node exists in the tree hierarchy.
    - depthOfVertex(Vertex parent, String label, int depth): Creates an intermediate node with the specified parent, label, and depth. It calls the superclass constructor to initialize the vertex with the given label and adds itself as a child to the parent.
    - depthOfVertex(Vertex parent, String label, String value, int depth): Creates an intermediate node with the specified parent, label, value, and depth. It calls the superclass constructor to initialize the vertex with the given label and value, and adds itself as a child to the parent.
    - Both constructors include logic to automatically add the created node as a child to its parent if the parent is not null. This ensures that the tree structure is maintained appropriately as nodes are created.
- Methods:
    - getDepth(): Returns the depth of the vertex. It allows external code to access the depth value of a 'depthOfVertex' object

    - getParent(): Overrides the ‘getParent()’ method of the superclass to return a ‘depthOfVertex’ type instead of the superclass ‘Vertex’. This ensures that the parent returned also maintains depth information.

This class allows for the creation of vertices in an AST or semantic tree with depth information, facilitating operations that require knowledge of the hierarchical structure of the tree.

5.4. CreateTree

This program, ‘CreateTree’, is responsible for constructing a tree data structure from input file content, representing an Abstract Syntax Tree (AST).

- nodeFromFile(String content):
    - Reads the content of the input file, which contains the representation of the AST.
    - Parses each line of the content to create vertex nodes and constructs the tree accordingly.
    - It also processes different types of nodes, such as ‘ID’, ‘EleValue’, ‘Str’, ‘Int’, ‘Truth’, and other nodes. The values of these nodes are extracted and stored appropriately in the ‘depthOfVertex’ objects.
    - Maintains the hierarchical structure of the tree using the depth information.-The method uses ‘.’ in the AST content to determine the depth of each node in the tree.
    - Utilizes the ‘depthOfVertex’ class to represent each vertex with depth details.
    - The method constructs and returns the root of the tree, which represents the complete AST structure.
- getJavaValue(String st):
    - The method is a utility function designed to convert Java string literals containing escape sequences into their corresponding unescaped characters. It processes each character of the input string st, replacing escape sequences with their respective characters, including octal, special, and Unicode escape sequences. This allows the method to produce a string where escape sequences are replaced by their actual character

representations, facilitating the interpretation and utilization of Java string literals in their unescaped form.

This program serves as an essential component in the process of constructing an AST from input file content, facilitating further analysis and processing of the AST within the context of a compiler or interpreter.

5.5. ASTtoST

This program, ‘ASTtoST’, is responsible for converting an Abstract Syntax Tree (AST) to a semantic tree (ST) representation. The AST is a hierarchical representation of the syntactic structure of a program, while the ST is a more organized representation of the program's semantics. The class provides a static method ‘astToSt’ that takes a ‘Vertex’ object as input. The ‘Vertex’ class is a representation of a node in the AST, containing information about the node's label and children. The ‘astToSt’ method recursively traverses the AST starting from the given ‘Vertex’, and it performs various transformations to standardize the AST nodes and create the corresponding ST nodes.

- expectChildren(Vertex vertex, int expect):
  - Error throwing function used to ensure that a vertex has the expected number of children.
  - Throws an exception if the number of children does not match the expected count.

- expectMoreChildren(Vertex vertex, int minimum):
  - Error throwing function used to ensure that a vertex has at least a minimum number of children.
  - Throws an exception if the number of children is less than the specified minimum.

- checkLabel(Vertex vertex, String expect):
  - Error throwing function used to check if a vertex has the expected label.
  - Throws an exception if the label of the vertex does not match the expected label.

- astToSt(Vertex vertex):
  - Main function responsible for converting the AST to ST.
  - Traverses the AST recursively and applies transformations based on the node labels. Handles specific AST node types (e.g., ‘let’, ‘where’, ‘function_form’) and represent them in ST form.
  - It performs transformations on the tree structure based on predefined rules to standardize certain nodes and their relationships. These transformations include reorganizing nodes, changing node labels, and adjusting child-parent relationships to ensure the resulting tree conforms to the desired syntax tree format.

This program facilitates the conversion of ASTs, commonly used in parsing and analyzing programming languages, into STs, which are crucial for symbol resolution and semantic analysis. It standardizes the AST representation into a format suitable for further processing by compilers or interpreters.

5.6. EleValueOrTuple

The ‘EleValueOrTuple’ class is an abstract class that serves as a base class for elements in the RPAL interpreter. It provides common functionality for elements, both for simple values (like ‘EleValue’) and tuple elements (like ‘EleTuple’). This abstract class encapsulates the common behavior shared by all elements, such as storing a label and providing methods to access and compare labels.

- Constructor:
  - Initializes the ‘name’ attribute, representing the label of the element value or tuple.

- isLabel(String label):
  - Method to check if the label of the instance matches a given label.
  - Returns ‘true’ if the labels match, indicating that the instance represents the specified label and ‘false’ otherwise.

3. getLabel():
  - Method to retrieve the label of the instance.
  - Returns the label stored in the ‘name’ attribute.

This abstract class provides a foundation for subclasses to represent specific types of element values or tuples. It encapsulates common functionality related to labels, allowing subclasses to focus on additional attributes or behaviors specific to their respective types.

5.7. EleValue

The ‘EleValue’ class represents individual elements (excluding tuples) and provides functionality to create, access, compare, and represent these elements. It extends the abstract class ‘EleValueOrTuple’, which serves as a base class for elements in the interpreter

- Constructor:
    - EleValue(String label): Initializes the element with a given label and sets the value to ‘null’.
    - EleValue(String label, String value): Initializes the element with both label and value.
    - EleValue(Vertex vertex): Creates an element from a ‘Vertex’ object, setting the label and value based on the vertex.

- getValue():
    - Method to retrieve the value of the element.

- equals(Object o):
    - Overrides the `equals` method to compare two ‘EleValue’ objects for equality based on their values.

- hashCode():
    - Overrides the `hashCode` method to generate a hash code for the ‘EleValue’ object based on its value.

- toString():
    - Overrides the ‘toString’ method to provide a string representation of the ‘EleValue’ object.
    - If the value is `null`, it returns the label.
    - Otherwise, it returns the label along with the value in parentheses.

This class encapsulates the functionality to represent individual elements, ensuring proper handling of labels and values. Additionally, it enables comparison and string representation of 'EleValue' objects. This class is essential in representing individual elements and their values during the evaluation process in the RPAL interpreter.

5.8. EleTuple

The 'EleTuple' class represents a tuple element that can store multiple elements. It extends the abstract class 'EleValueOrTuple', which serves as a base class for elements in the RPAL interpreter.

- Constructor:
  - EleTuple(EleValueOrTuple[] value): Initializes a tuple element with the given array of elements. The label is set to "tuple".

- getValue():
  - Method to retrieve the array of elements contained within the tuple.

- equals(Object o):
  - Overrides the 'equals' method to compare two 'EleTuple' objects for equality based on their contained elements.

- hashCode():
  - Overrides the 'hashCode' method to generate a hash code for the 'EleTuple' object based on its contained elements.

- toString():
  - Overrides the 'toString' method to provide a string representation of the 'EleTuple' object.
  - It returns a string representation of the array of elements contained within the tuple.

This class encapsulates the functionality to represent tuple elements and enables comparison and string representation of 'EleTuple' objects based on their contained elements. This class is crucial in representing complex data structures and elements during the evaluation process in the RPAL interpreter.

5.9. ElementParser

The ‘ElementParser’ class is responsible for converting an abstract syntax tree (AST) into an array of control structures using a preorder traversal.

- generateCS(Vertex root):
  - Public method that initiates the generation of control structure (Stack<EleValue>) based on the given root vertex of the AST.
  - It creates an ArrayList of stacks (ArrayList<Stack<EleValue>>) to store the control structures, then calls the private helper method ‘generateCS’ to traverse the AST in a preorder manner and generate the control structures.

- generateCS(Vertex vertex, ArrayList<Stack<EleValue>> controls, Stack<EleValue> currentControl)**:
  - Private method that recursively generates the control structures by traversing the AST in a preorder manner.
  - Depending on the type of vertex encountered, it delegates the traversal to specific helper methods (‘generateCSLambda’, ‘generateCSIf’, ‘generateCSTau’), or pushes elements onto the current control stack.

- generateCSLambda(Vertex vertex, ArrayList<Stack<EleValue>> controls, Stack<EleValue> currentControl):
  - Handling the traversal and generation of control structures when encountering a vertex labeled as ‘’lambda” during the preorder traversal of the abstract syntax tree (AST).
  - It retrieves the left and right children of the current ‘’lambda” vertex.
  - If the left child is labeled as “,”, indicating multiple parameters, it combines their values into a single ID node.
  - It creates a new control element representing the “lambda” node, containing information about the index of the new control

structure and the value of the left child (either a single parameter or multiple parameters combined).
  - It pushes the newly created control element onto the current control stack.
  - It creates a new empty control structure stack to handle traversal in the right child subtree.
  - It then recursively traverses the right child subtree, passing the newly created control structure stack for further processing.

- generateCSIf(Vertex vertex, ArrayList<Stack<EleValue>> controls, Stack<EleValue> currentControl):
  - Splits the control structure on "if" nodes into "then" and "else" children.
  - It creates delta nodes for both the then and else subtrees and pushes them to the current control structure. The then and else children are traversed in these new structures, respectively.
  - Finally, a beta node is pushed to the current control structure, and the condition child is traversed in the current structure.

- generateCSTau(Vertex vertex, ArrayList<Stack<EleValue>> controls, Stack<EleValue> currentControl):
  - Handling the traversal and generation of control structures when encountering a vertex labeled as "tau" during the preorder traversal of the abstract syntax tree (AST).
  - It pushes a new control element onto the current control structure stack representing the "tau" node. This element contains information about the number of children (elements) associated with the current vertex.
  - It iterates over each child vertex of the current vertex and recursively traverses each child subtree, continuing the preorder traversal.

This class facilitates the transformation of AST nodes into an array of control structures using a preorder traversal approach, which helps in generating the control structures that will be used during the evaluation of the RPAL code.

5.10. OperationHandler

The ‘OperationHandler’ class is responsible for applying various functions and operators on RPAL elements. It performs operations on both numerical and boolean values and handles unary and binary operations.

- Interfaces:
    - operationHandlerBoolean: Defines an interface for boolean operations.
    - operationHandlerString: Defines an interface for string operations.
    - operationHandlerNumerical: Defines an interface for numerical operations.

- Methods:
    - extract(EleTuple operation, EleValueOrTuple operand): Extracts elements from a tuple based on the provided index operand.
    - checkMathematicalOperation(EleValueOrTuple op): Checks if a binary mathematical operation is applicable.
    - checkArrayOperation(EleValueOrTuple op): Checks if a unary array operation is applicable.
    - applyOperations(EleValueOrTuple operation, EleValueOrTuple operand1, EleValueOrTuple operand2): Applies binary operators to two operands.
    - apply(EleValueOrTuple operation, EleValueOrTuple operand): Applies unary functions and operators to a single operand.

- Private Helper Methods:
    - numericalOperator(EleValueOrTuple operand1, EleValueOrTuple operand2, operationHandlerNumerical operation): Performs numerical operations on integer operands.

- binaryBooleanOperator(EleValueOrTuple operand1, EleValueOrTuple operand2, operationHandlerBoolean operation): Performs boolean operations on boolean operands.
- covertToString(EleValueOrTuple element): Converts elements and tuples into string expressions.
- booleanCondition(boolean condition): Converts boolean primitives into boolean elements.
- getSubString(EleValueOrTuple operand, operationHandlerString operation): Gets substrings from a string operand.
- order(EleValueOrTuple operand): Returns the number of elements in a tuple.
- conc(EleValueOrTuple operand): Concatenates strings.
- iToS(EleValueOrTuple operand): Converts integers to strings.
- not(EleValueOrTuple operand): Applies the boolean NOT operator.
- or(EleValueOrTuple operand1, EleValueOrTuple operand2): Applies the boolean OR operator.
- greater(EleValueOrTuple operand1, EleValueOrTuple operand2): Compares two elements.
- aug(EleValueOrTuple operand1, EleValueOrTuple operand2): Adding an element to the end of a tuple.

Overall, the class provides a comprehensive set of methods for performing various operations on elements and tuples.Handles error cases gracefully by throwing runtime exceptions when necessary. This class plays a crucial role in applying various operations to RPAL elements and is utilized in the RPAL interpreter to evaluate expressions and perform operations during execution.

5.11. Stack

The 'Stack' class is a simple implementation of a stack data structure used to store elements of type 'T', which extends 'EleValueOrTuple'.

- Constructor: The constructor initializes an internal `java.util.Stack` to store elements.

- Methods
    - push(T element): The ‘push’ method adds an element to the top of the stack.
    - pop(): The ‘pop’ method removes and returns the element at the top of the stack.
    - isEmpty(): The ‘isEmpty’ method checks if the stack is empty.
    - size(): The ‘size’ method returns the number of elements in the stack.
    - toString(): The ‘toString’ method returns a string representation of the stack.
- Iterable : The class implements the ‘Iterable’ interface, allowing the stack to be iterated using a foreach loop.

Overall, the ‘Stack’ class provides a basic stack data structure with essential operations for managing elements. It ensures type safety by restricting the types of elements that can be stored in the stack to those that extend ‘EleValueOrTuple’. It keeps track of elements during expression evaluation and control flow handling.

5.12. CSEMachine

The ‘CSEMachine’ class represents a CSE machine used to evaluate the traversed tree generated by the parser based on a control stack. The class applies the reduction rules of RPAL to compute the final result of the program.

- Fields:
    - eleValues: A stack of EleValue objects representing elements in the evaluation process.
    - eleValueOrTuples: A stack of EleValueOrTuple objects representing elements or tuples in the evaluation process.
    - operationHandler: An instance of the OperationHandler class to handle mathematical and array operations.
    - environments: An ArrayList of Environment objects representing the runtime environments created during the evaluation process. This is a data structure that holds variable bindings and enables scoping and variable lookups during evaluation.

    - CS: An ArrayList of Stacks representing the control structures of the tree.
- Constructor: Initializes the control structures (‘CS’), element stacks (‘eleValues’ and ‘eleValueOrTuples’), operation handler (‘operationHandler’), and environment list (`environments`) required for evaluation.
- Methods:
    - toString(): Overrides the toString method to provide a string representation of the current state of the machine.
    - currentEnvironmentIndex(): Finds the index of the current environment by peeking into the stack.
    - currentEnvironment(): Retrieves the current environment based on its index.
    - evaluateTree(): The main method in the class is “evaluateTree()”, which processes the control structures and applies the reduction rules to evaluate the RPAL code. The method uses a stack-based approach to handle evaluation, pushing and popping elements as it performs reductions.
    - extractDelta(int controlIndex): Extracts elements from the control structure specified by its index.
    - Rule1 to Rule13(EleValue name): Represents the rules followed during the evaluation process. Each rule corresponds to a specific action to be taken based on the current element being evaluated.

The class follows a modular approach by separating different functionalities into methods, making it easier to understand and maintain. It effectively applies rules to evaluate the expression based on the control structures and element stacks. The class is an essential component of an RPAL interpreter or evaluator, allowing RPAL code to be executed and produce the desired outcomes.

5.13. Environment

The ‘Environment’ class is responsible for managing environments in the RPAL interpreter. An environment is a data structure that keeps track of the names and their corresponding values in the RPAL program.

- Fields:
    - parent: A reference to the parent environment, indicating the scope hierarchy.
    - memory: A HashMap storing variable names as keys and their corresponding values (EleValueOrTuple) as values.

- Constructor:
    - Environment(): Creates the primary environment with predefined entries for built-in functions.
    - Environment(Environment parent): Creates an empty sub-environment with a reference to its parent.
    - Environment(Environment parent, String key, EleValueOrTuple value): Creates a sub-environment with one entry and a reference to its parent.
- Methods:
    - remember(String key, EleValueOrTuple value): Adds a new entry to the current environment. Throws an error if the variable is already defined.
    - lookup: Retrieves the value of a variable given its name (`id`) from the environment. It first checks if the variable exists in the current environment's "memory". If the variable is not found in the current environment, it checks the parent environment (if available) by recursively calling the `lookup` method.If the variable is not found in any environment, and it is in the primary environment, it throws a "RuntimeException" indicating that the variable is undefined.
    - toString: The 'toString' method is overridden to provide a human-readable representation of the environment and its entries. It includes the entries from the current environment and, if available, its parent environment and concatenates the entries of each environment.

The class provides a way to manage and access variables and their values within different environments, facilitating variable scoping and lookup during evaluation in the CSE machine. It prevents redefinition of variables within the same scope and supports the creation of primary and sub-environments. Overall, it ensures the integrity of variable assignments and enables proper scope resolution throughout the program.

5.14. ExceptionHandlerOfAST

The ‘ExceptionHandlerOfAST’ class extends ‘RuntimeException’ and is used for handling exceptions specific to standardizing the abstract syntax tree.

- Constructor:
  - It provides a constructor that takes a message parameter, allowing the caller to specify the details of the exception.

This custom exception class allows for more specific handling of errors related to the standardization process of the abstract syntax tree, providing better clarity and control over exception handling in the codebase.

5.15. ExceptionHandlerOfCSE

The ‘ExceptionHandlerOfCSE’ class is an exception specifically designed for handling errors encountered during the evaluation process of the CSE machine. It extends the ‘RuntimeException’ class, indicating that instances of this exception can be thrown during runtime.

- Constructor: It provides a constructor that takes a message parameter, allowing the caller to specify the details of the exception.

This custom exception class is useful for capturing and handling errors that occur during the evaluation of the CSE machine, providing a more structured approach to error management in the codebase.

5.16. rpal20

This ‘rpal20’ class serves as the entry point for executing the RPAL interpreter. It reads an AST file, constructs an Abstract Syntax Tree (AST), converts it into a Symbol Table (ST), generates control structures, and finally evaluates the RPAL program using the CSEMachine.

- “main” Method:
    - This method is the entry point of the RPAL interpreter and is responsible for executing the RPAL program.
    - Inside the `try` block, it checks if command-line arguments are provided. If not, it throws an exception indicating that the AST filename should be provided as an argument.
    - If command-line arguments are provided, it initializes a ‘Parser’ object with the filename and starts parsing, obtaining the content of the AST.
    - It then creates a tree structure from the AST content using the ‘CreateTree.nodeFromFile’ method.
    - Converts the AST to ST (Standardized Tree) using ‘ASTtoST.astToSt’.
    - Generates control structures from the tree using ‘ElementParser.generateCS’.
    - Initializes a ‘CSEMachine’ with the generated control structures and evaluates the tree using ‘cseMachine.evaluateTree()’.
    - Any exceptions thrown during parsing, AST to ST conversion, CSE evaluation, or runtime exceptions are caught and handled appropriately.
    - If any exception occurs, it prints an error message indicating the type of exception and the message associated with it.

Overall, this class organizes the parsing, conversion, and evaluation processes for RPAL programs, handling exceptions that may occur during these operations.

6. Running the Program: Step-by-Step Guide

Prerequisite: Java runnable environment, Chocolatey

Step 1: Extract the Submission File

Step 2: Open the terminal and navigate to the directory containing the extracted file.

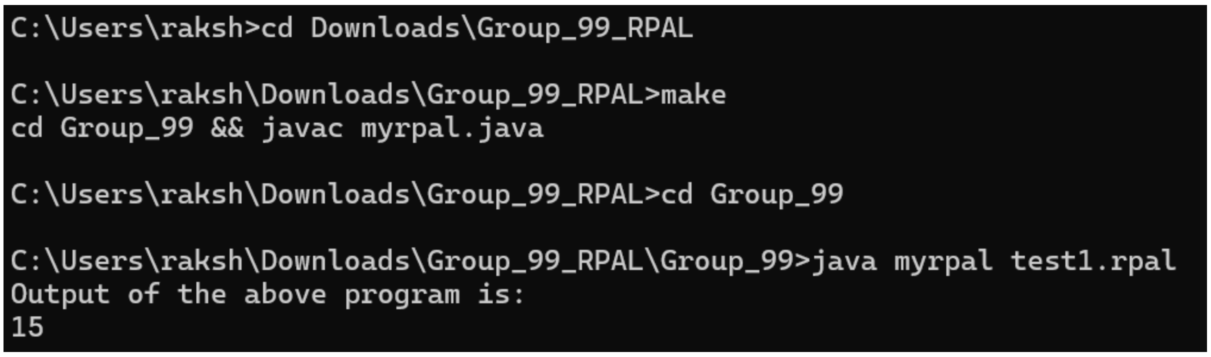
Step 3: Build the Project

- To build the myrpal class, execute the make command in the terminal:
```
>> make
```
- The make command reads the makefile and compiles myrpal.java using the Java compiler (javac). If everything is set up correctly, this should create the myrpal .class file.

Step 4: Run the RPAL Test Programs

- Navigate to the directory 'Group_99'
```
>> cd Group_99
```
- Now, you can execute the RPAL test programs using the java command along with the myrpal class:
```
>> java myrpal test1.rpal
```
- This command executes the Java program myrpal with the RPAL input file test1.rpal as an argument. The output of the RPAL interpreter will be printed in the terminal.
- The same procedure is repeated for input file test2.rpal.
- To get the Abstract Syntax Tree:
```
>> java myrpal test1.rpal -ast
```

Note: If we have to run another test program, we have to put that test program file into the folder 'Group_99' and do the same procedure as above by changing the input file name accordingly.

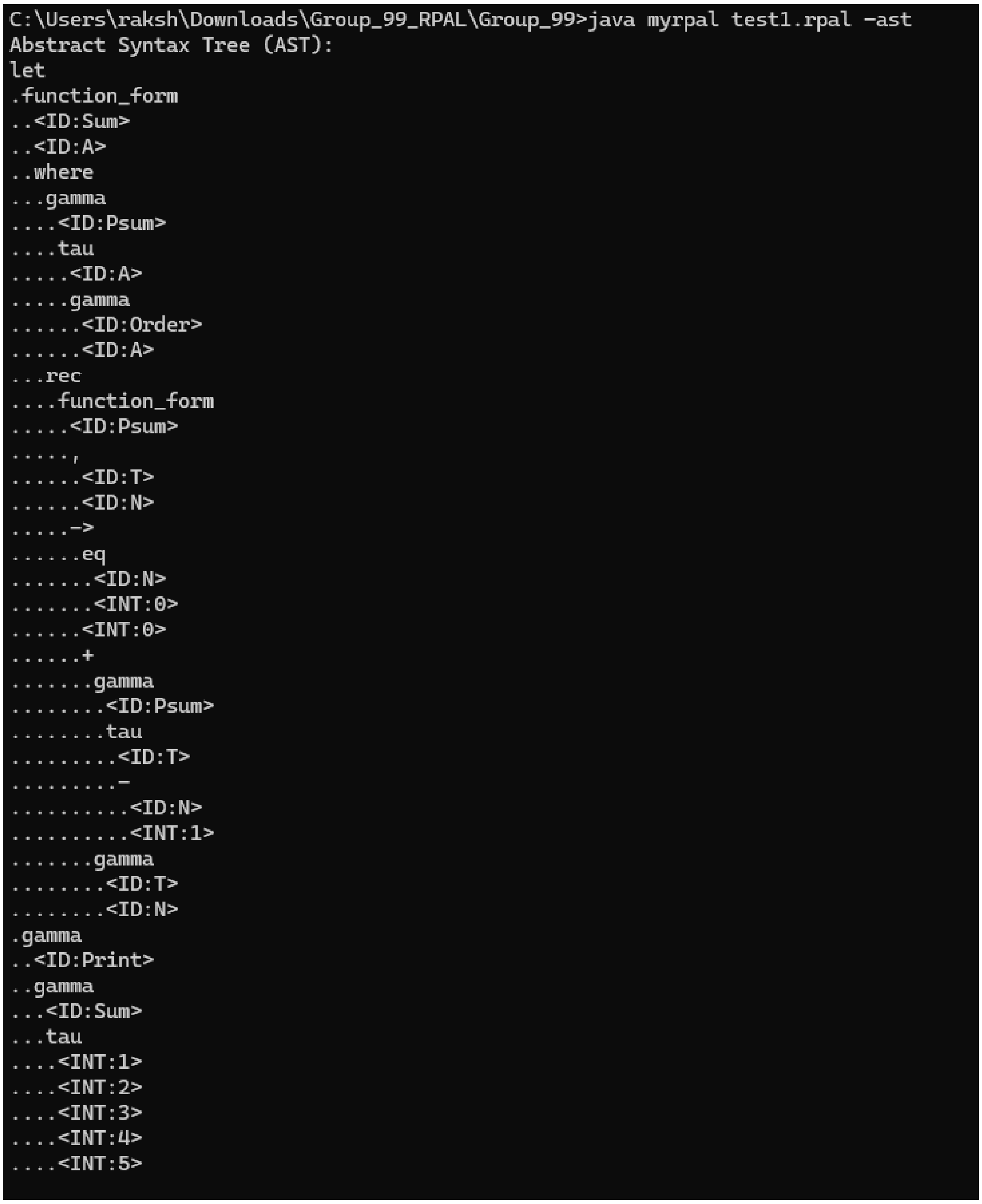


```
C:\Users\raksh>cd Downloads\Group_99_RPAL

C:\Users\raksh\Downloads\Group_99_RPAL>make
cd Group_99 && javac myrpal.java

C:\Users\raksh\Downloads\Group_99_RPAL>cd Group_99

C:\Users\raksh\Downloads\Group_99_RPAL\Group_99>java myrpal test1.rpal
Output of the above program is:
15
```



```
C:\Users\raksh\Downloads\Group_99_RPAL\Group_99>java myrpal test1.rpal -ast
Abstract Syntax Tree (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>
```

7. Conclusion

The RPAL interpreter is built on key components such as the Lexical Analyzer, responsible for tokenizing RPAL code, and the Parser, which constructs the Abstract Syntax Tree (AST). This AST is then converted into the Standardized Tree (ST), organizing symbols for efficient resolution. The Control-Stack-Environment (CSE) machine generates control structures to manage program flow during evaluation. With these components in place, the CSE machine executes RPAL programs, generating the desired output. This comprehensive interpreter offers users a robust platform for functional programming, enabling effective problem-solving and computation.