CS 3513 - Programming Languages

Programming Project - REPORT

Designing an interpreter for RPAL language

Prepared by Group 29

DE SILVA N.Y.D.	200110P
PERERA K.O.S.	200458M
WARNAKULASURIYA A.K.	200694G
WEERASINGHE C.D.R.M.	200700B

Introduction

The project is to implement a lexical analyzer and a parser for the RPAL language. The parser should return the Abstract Syntax Tree (AST) for the provided input program. Then the Abstract Syntax Tree should be transformed into the Standardized Tree (ST) using an algorithm and evaluate the program using 13 rules of the Control Stack Environment (CSE) machine.

Implementation uses Java version 19 and Visual Studio IDE is used for development.

Executing the Program

• **Building with Makefile**

To compile the program run makefile. For that, issue the command “make” in the same directory where the makefile is stored. Then, it will compile the code in the same folder creating the respective .class files.

• **Executing the program**

To execute the program run

```
java rpal20 rpal_test_programs/rpal_01 > output.01
```

“rpal_01” can be replaced with the path to the program file. The output will be written to the src directory(root) a file named “output.01”

• **Testing the program**

The directory called "rpal_test_programs" must be created with the test programs and the expected outputs following the name convention given in the project description.

To test the program after executing the program with the above command run
diff output.01 rpal_test_programs/output01.test.

This will test the output written to the file output.01 and check the expected output stored in the rpal_test_programs directory named output01.test

• **Build, execute, and test with a single command**

To build, execute and test program in a single command run,

make check

Project Structure

The following are the main directories of the project.

- The root directory where the makefile is stored contains Java files, the following packages of the project, and the report.

In the project, four tasks are identified. The Java classes are decomposed into four packages.

- **lexical_analyzer** - Include classes needed for breaking the source code into individual tokens, which are the basic building blocks of the language.
- **Parser** - Include classes to take the stream of tokens generated by the lexical analyzer and construct an Abstract Syntax Tree (AST) based on the syntactic rules of the RPAL language. Then transform it into Standardized Tree (ST).
- **control_structures** - Include classes to create the Control Structures using the Standardized Tree.
- **cse_machine** - Include classes to evaluate the program using the generated control structures.

Classes and Significant Functions

- **rpal20.java**

  Contains the main function of the program. Accept the filename from the terminal and use lexical_analyzer and Parser packages to build and standardize the tree. Then, use control_structures and cse_machine packages to evaluate the standardized tree.

Classes in the lexical_analyzer package

- **LexicalAnalyzer.java**

  The lexicalAnalyzer class is responsible for scanning and screening the RPAL source code to output a list of tokens for the parser. The tokens are created according to the RPAL lexicon [1].

  Methods:

  - **`public ArrayList<Token> getTokenList()`**
    Returns the token list that has been scanned and screened for the parser
  - **`private void constructToken(String nextChr, String buffer)`**
    Takes the scanned character and tokenizes according to the RPAL lexicon
  - **`private String nextChr()`**
    Read the next character from the file and return it.

- **Token.java**

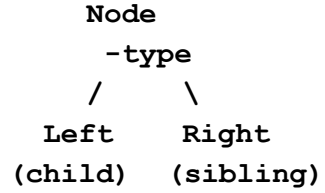
  The token class contains the definition of a token which consists of two main parameters, namely, the type and value. The type can have values like IDENTIFIER, STRING, OPERATOR, DELETE, etc. that are again defined in the lexicon [1], and values are the scanned string. Typical getters and setters are defined to be accessible by the parser, and the toString method has been modified to print.

  Eg: <IDENTIFIER: ‘fibonacci’>, <STRING: ‘group_29’>, <DELETE: ‘//comment’> , etc.

Classes in the Parser package

- **Node.java**

The Node class is to represent the nodes in the abstract syntax tree using the first child next sibling representation. There are functions to create nodes, getters, and setters for the left and right nodes of the node.

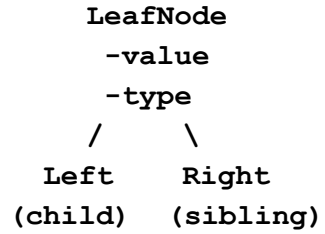


Methods:

- **`public String getType()`**
  Returns the type of the node.
- **`public void setType (String type)`**
  Modifies the type of the node.
- **`public Node getLeft()`**
  Returns the child of the node.
- **`public void setLeft(Node leftNode)`**
  Updates the child node.
- **`public Node getRight()`**
  Returns the right node (sibling) of the node.
- **`public void setRight(Node rightNode)`**
  Updates the sibling node.
- **`public void appendRight (Node newNode)`**
  Adds new siblings to the node. If a node already has a sibling, traverse through the set of siblings and add the new node to the last sibling.

- **LeafNode.java**

It is the child class that extends the Node class used to represent terminal (leaf nodes) in AST such as identifiers, integers, strings, etc.



Methods:

- **`public String getValue()`**
  Returns the value of the node.
- **`public void setValue (String value)`**
  Modifies the value of the node.

- **AST.java**

This class represents the abstract syntax tree and an AST is identified using its root node. The **`std`** attribute indicates whether or not the tree has been standardized. This class consists of methods to standardize the tree and print the tree using preorder traversal.

Methods:

- **`public Node getRoot()`**
  Returns the root of the AST.
- **`public void setRoot (Node root)`**
  Modifies the root of the tree.
- **`public boolean isStandardized()`**
  Returns the value of the std attribute.
- **`public void standardize()`**
  Converts the AST to the standardized tree and updates the std attribute.
- **`private void standardizeNode()`**
  Helper method to standardize the AST. standardize each node of the AST according to standardization rules from the bottom-most node to the top of the tree.
- **`private Node createLambdas()`**
  Creates a chain of lambda nodes.
- **`public void print()`**
  Prints the AST or standardized tree.
- **`private void printPreorder(Node node, String printPrefix)`**
  Helper function to print the tree by performing preorder traversal through the tree. The parameter printPrifix is used to format the print pattern of the tree.
- **`private void printNode(Node node, String printPrefix)`**
  Prints the type and value details of a given node formatted using printPrefix.

- **ParseTree.java**

The parseTree class handles the parsing process of RPAL programs according to RPAL grammar using the sequence of tokens produced by the lexical analyzer. This class includes methods for each nonterminal in the productions and each method commits to one production based on the next input symbol. Also, a ParseTree object maintains a stack to store processed tokens and trees.

Methods:

- **`public AST buildAst()`**
  Parses the sequence of tokens and returns the abstract syntax tree.
- **`private void parse()`**
  This function is used by the **`buildAst()`** method to parse the token list.
- **`private void buildTree(String type, int n)`**
  Pop n number of contents from the stack and build the first child next sibling tree with the root node of type **`'type`**' and push the tree into the stack.

- **`private void readNext()`**
  Reads the next token in the token list and updates the attribute **`curr_token`**. Ignores the tokens of type “DELETE”.
- **`private boolean isType(Token token, String type)`**
  Returns true if the input token is of the input type, otherwise returns false.

- Methods for productions in RPAL grammar:

- **`private void procE()`**
- **`private void procEW()`**
- **`private void procT()`**
- **`private void procTA()`**
- **`private void procTc()`**
- **`private void procB()`**
- **`private void procBt()`**
- **`private void procBs()`**
- **`private void procBp()`**
- **`private void procA()`**
- **`private void procAt()`**
- **`private void procAf()`**
- **`private void procAp()`**
- **`private void procR()`**
- **`private void procRn()`**
- **`private void procD()`**
- **`private void procDa()`**
- **`private void procDr()`**
- **`private void procDb()`**
- **`private void procVb()`**
- **`private void procVl()`**

- **ParserException.java**

Exceptions in Parser.

Methods:

- **`public ParseException(String message)`**
  Generates Parser exception.

Classes in the control_structures package

- **CSNode.java**

The CSNode class is used for Objects used as Nodes inside the Control Structures and CSE machine.

Methods:

- **`public CSNode(String t, String n)`**

  This constructor method is used for tau node and identifier type variables. Only “type” and “name” are made. But for the tau node need to declare “tauno” (number of variables in the tuple) explicitly using the “setTauno” method.

- **`public CSNode(String t, List<String> varLambda, int lambda_no)`**

  This constructor method is used for lambda variables. The “type” will always be declared as "lambdaClosure". The “lambdavar” indicates the name of the variable to substitute for and “lambdano” gives the number of the delta control structure.

- **`public CSNode(String t, int env_no)`**

  This constructor method is used for environment variables. Only “type” and “envno” are made.

- **`public CSNode(String t, int then_no, int else_no)`**

  This constructor method is used for conditional statements. The “type” is always given as "beta" and needs the “then_no” and “else_no” as the parameters.

- **`public CSNode(String t, int delta_no, List<CSNode> delta_struct)`**

  This constructor method is used to create an object for inserting a delta structure into the control stack and “delta_no” is stored in the “envno” parameter.

- **`public boolean getIsTuple()`**

  Returns whether the node is a tuple or not.

- **`public List<CSNode> getTuple()`**

  Returns a list that stores the elements of the tuple.

- **`public String getType()`**

  Returns the type of node.

- **`public String getName()`**

  Returns a string that indicates the value of the node for Integer, String, and Truthvalue.

- **`public List<String> getLambdavar()`**

  Returns the list of nodes enclosed by the lambda node.

- **`public int getLambdano()`**

  Returns an integer that indicates the delta control structure it connects to.

- **`public int getEnvno()`**

  Returns an integer that indicates the environment number the node belongs to and is used in lambda and env nodes.

- **`public int getThenno()`**

  Returns an integer that indicates the control structure to load if the condition is true and used in Beta nodes.

- **`public int getElseno()`**

  Returns an integer that indicates the control structure to load if the condition is false and used in Beta nodes.

- **`public int getTauno()`**

  Returns an integer that indicates the number of elements in the tau node and used in tau nodes.

- **`public void setIsTuple(boolean isTuple)`**

  Modifies the “isTuple” parameter.

- **`public void setTuple(List<CSNode> tuple)`**

  Modifies the “tuple” parameter.

- **`public void setType(String type)`**

  Modifies the “type” parameter.

- **`public void setName(String name)`**

  Modifies the “name” parameter.

- **`public void setLambdavar(List<String> lambdavar)`**

  Modifies the “lambdavar” parameter.

- **`public void setLambdano(int lambdano)`**

  Modifies the “lambdano” parameter.

- **`public void setEnvno(int envno)`**

  Modifies the “envno” parameter.

- **`public void setThenno(int thenno)`**

  Modifies the “thenno” parameter.

- **`public void setElseno(int elseno)`**

  Modifies the “elseno” parameter.

- **`public void setTauno(int tauno)`**

  Modifies the “tauno” parameter.

- **`public CSNode duplicate()`**

  Duplicate the contents of Control Structure Nodes.

- **ControlStructures.java**

The ControlStructures class is used to create the Control Structures using the Standardized Tree.

Methods:

- **`public void genControlStructures(Parser.Node n1)`**

  Function to generate the control structures starting from the root of the ST.

- **`public void preorder(Node root, ArrayList<CSNode> currentdelta)`**

  Function to perform the preorder traversal from the starting node.

- **`public void display()`**

  Function used to display the control structures.

- **`public List<List<CSNode>> getCS()`**

  Function to obtain the control structures.

- **`public int getDeltano()`**

  Returns the number of Control Structures generated.

Classes in the cse_machine package

- **CSE.java**

The CSE class is used to handle the operations of the CSE Machine and perform the evaluation of the program

Methods:

- **`public CSE(List<List<CSNode>> deltaLists)`**

  Function to initialize a new CSE machine for each program given the list of control structures

- **`public void insertToControl(int delta_num)`**

  Function to insert a control structure into the Control given the delta number

- **`public void expandDelta()`**

  Function to open the control structure and place its constituent nodes into the stack in order

- **`public void setupCSE()`**

  Function to Start up the CSE machine and initialize it

- **`public void runCSE()`**

  The Main Driver Function of CSE is to implement logic to execute the rules of CSE. The rule to implement at each step is decided by checking the topmost control and stack contents.

- **`public Stack<CSNode> getControlList()`**

  Returns the Control list of the CSE machine as a Stack

- **`public void setControlList(Stack<CSNode> controlList)`**

  Modifies the Control list of the CSE machine

- **`public Stack<CSNode> getStackList()`**

  Returns the Stack list of the CSE machine as a Stack

- **`public void setControlList(Stack<CSNode> stackList)`**

  Modifies the Stack list of the CSE machine

- **`public int getCurr_env()`**

  Returns the current environment number the CSE machine is operating in

- **`public void setCurr_env(int curr_env)`**

  Modifies the current environment number the CSE machine is operating in

- **`public int getEnvCounter()`**

  Returns the number of the lastly created environment

- **EnvNode.java**

Class to represent an Environment Node inserted into the Environment Tree that stores the variables and values for the relevant environment

Methods:

- **`public EnvNode(int env_no, CSNode variable, EnvNode parentEnv)`**

  Function to create a new Environment Node given the environment number, the variables stored and their corresponding values, and the parent Environment Node

- **`public int getEnv_no()`**

  Returns the environment number of an environment node

- **`public void setEnv_no(int env_no)`**

  Modifies the environment number of an environment node

- **`public CSNode getVariable()`**

  Returns a CSNode that contains variables and their corresponding values of an environment node

- **`public void setEnv_variable(CSNode env_variable)`**

  Sets a CSNode that contains variables and their corresponding values of an environment node

- **`public EnvNode getParentEnv()`**

  Returns the parent environment node

- **`public EnvNode setParentEnv()`**

  Sets the parent environment node

- **EnvironmentTree.java**

Class to represent the Environment Tree used for the CSE Machine. It is represented using a list of EnvNodes

Methods:

- **`public EnvironmentTree()`**

  Initialize an Environment Tree

- **`public void addEnv(int env_no, CSNode variable, EnvNode parentEnv)`**

  Creates a new Environment node and will add it to the Environment Tree

- **`public void removeEnv(int env_no)`**

  Removes the Environment node from the Tree given its number

- **`public EnvNode getEnvNode(int env_no)`**

  Returns the Environment node from the Tree given its number

- **EvaluationException.java**

Class to generate Exceptions caused during the CSE evaluation phase. Extends the RunTimeException Class

Methods:

- **`public EvaluationException(String message)`**

  Generates Parser exception.

- **RPALBinaryOps.java**

Class of Methods related to Binary Operations between Nodes in RPAL.

Methods:

- **`public static CSNode add(CSNode node1, CSNode node2)`**

  Performs addition operation of two integers and returns their sum

- **`public static CSNode subtract(CSNode node1, CSNode node2)`**

  Performs subtraction operation of two integers and returns their difference

- **`public static CSNode multiply(CSNode node1, CSNode node2)`**

  Performs multiplication operation of two integers and returns the product

- **`public static CSNode divide(CSNode node1, CSNode node2)`**

  Performs division operation of two integers and returns the quotient

- **`public static CSNode power(CSNode node1, CSNode node2)`**

  Performs exponent operation of two integers and returns the power of the 1st integer

- **`public static CSNode isEqual(CSNode node1, CSNode node2)`**

  Returns true when the nodes contain the same value. Applicable for integers, strings, and truth values where both have the same type.

- **`public static CSNode isNotEqual(CSNode node1, CSNode node2)`**

  Returns true when the nodes do not contain the same value. Applicable for integers, strings, and truth values where both have the same type.

- **`public static CSNode isLessThan(CSNode node1, CSNode node2)`**

  Returns true when the first node is less than the second node. Applicable for integers and strings (lexicographically) where both have the same type.

- **`public static CSNode isGreaterThan(CSNode node1, CSNode node2)`**

  Returns true when the first node is greater than the second node. Applicable for integers and strings (lexicographically) where both have the same type.

- **`public static CSNode isLessEqualThan(CSNode node1, CSNode node2)`**

  Returns true when the first node is less than or equal to the second node. Applicable for integers and strings (lexicographically) where both have the same type.

- **`public static CSNode isGreaterEqualThan(CSNode node1, CSNode node2)`**

  Returns true when the first node is greater than or equal to the second node. Applicable for integers, strings, and truth values where both have the same type.

- **`public static CSNode logicOR(CSNode node1, CSNode node2)`**

  Applies the OR operation to the operands.Applicable when both nodes are truth values.

- **`public static CSNode logicAND(CSNode node1, CSNode node2)`**

  Applies the AND operation to the operands.Applicable when both nodes are truth values.

- **`public static CSNode augment(CSNode node1, CSNode node2)`**

  Appends node2 into the tuple of node1. Applicable when node1 is of type tuple or NIL.

- **RPALFunc.java**

Class of Methods related to in-built functions used in RPAL.

Methods:

- **`public static boolean checkInBuilt(String name)`**

  Returns true when the identifier name matches with an inbuilt function name

- **`public static void Print(CSNode node)`**

  Prints the contents of the node depending on its type

- **`public static CSNode Stem(CSNode node)`**

  Returns a CSNode that stores the first character of the String stored in the node. Applicable for Strings only.

- **`public static CSNode Stern(CSNode node)`**

  Returns a CSNode that stores the string without the first character stored in the node. Applicable for Strings only.

- **`public static CSNode ConcOne(CSNode node)`**

  Performs the first step in the Concatenating Process. Forms a ‘ConcOne’ CSNode that stores the first string to be concatenated. Applicable for Strings only.

- **`public static CSNode Conc(CSNode node1, CSNode node2)`**

  Performs the second step in the Concatenating Process. Returns the CSNode that stores the concatenated string. Applicable for Strings only.

- **`public static CSNode Order(CSNode tupleNode)`**

  Returns the number of elements in the tuple node as a CSNode. Applicable for tuple-type nodes (tuple, nil).

- **`public static CSNode Null(CSNode tupleNode)`**

  Returns true when the tupleNode is NIL as a CSNode.

- **`public static CSNode Isinteger(CSNode node)`**

  Returns true when the node is of integer type

- **`public static CSNode Istruthvalue(CSNode node)`**

  Returns true when the node is of truth value type

- **`public static CSNode Isstring(CSNode node)`**

  Returns true when the node is of string type

- **`public static CSNode Istuple(CSNode node)`**

  Returns true when the node is of the tuple type

- **`public static CSNode Isfunction(CSNode node)`**

  Returns true when the node is of function type

- **`public static CSNode Isdummy(CSNode node)`**

  Returns true when the node is of dummy type

- **`public static CSNode intToStr(CSNode intNode)`**

  Sets the type of the integer node into “STRING”. Applicable for Integer-type nodes only.

- **RPALUnaryOps.java**

Class of Methods related to Unary Operations between Nodes in RPAL.

Methods:

- **`public static CSNode logicNot(CSNode node)`**

  Performs the NOT operation of a Truth Value value

- **`public static CSNode neg(CSNode node)`**

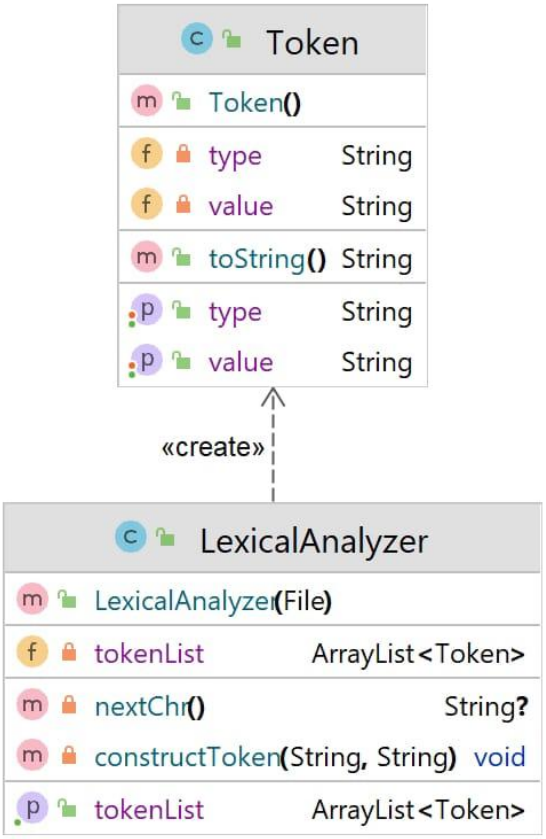
  Performs the negation operation of an Integer value

References

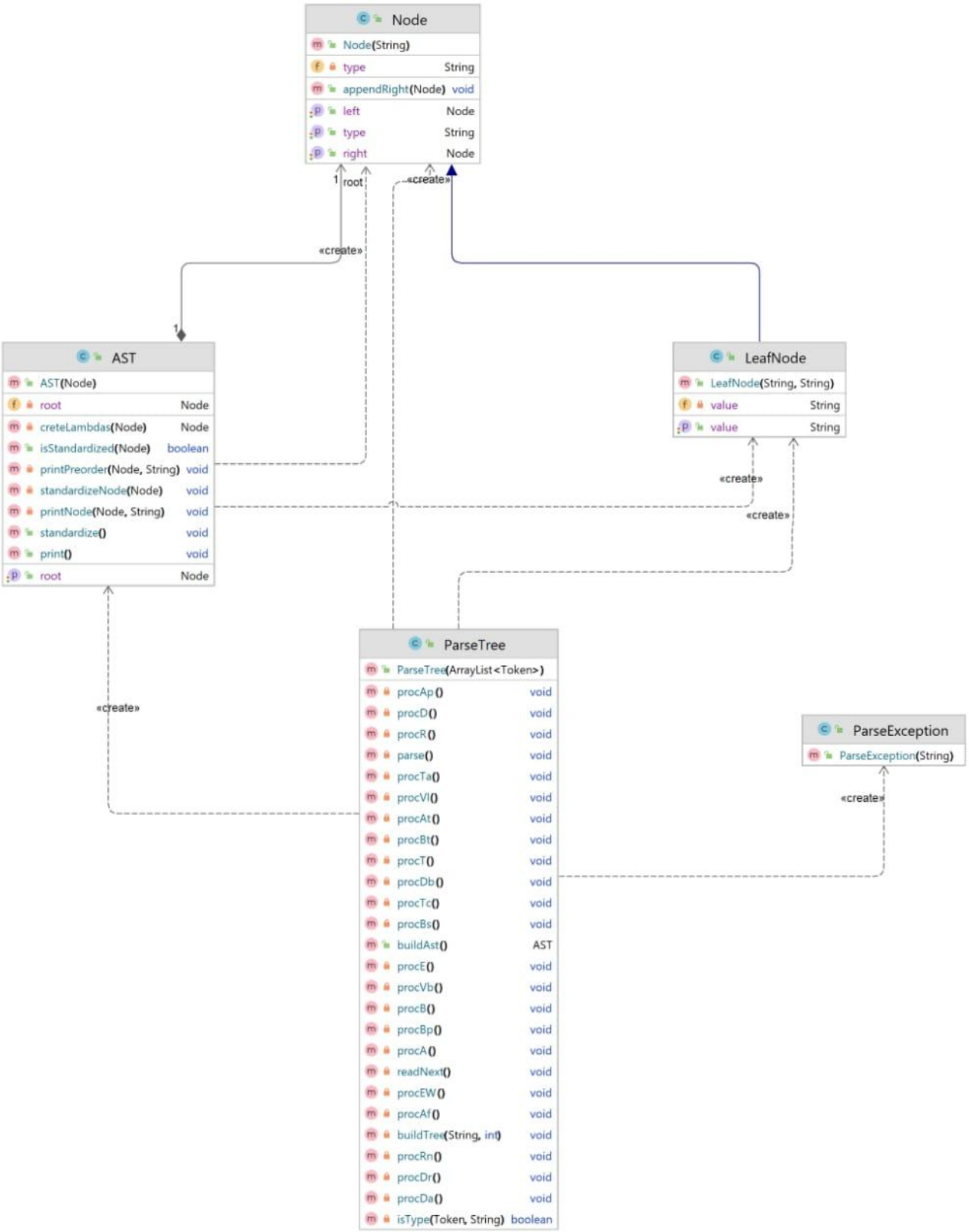
[1] “RPAL lexicon.” [Online]. Available: https://rpal.sourceforge.net/doc/lexer.pdf
[2] “RPAL Grammar.” [Online]. Available: https://rpal.sourceforge.net/doc/grammar.pdf
[3] “RPAL Semantics.” [Online]. Available: https://rpal.sourceforge.net/doc/semantics.pdf
[4] “RPAL - Right-reference Pedagogic Algorithmic Language.” https://rpal.sourceforge.net/ (accessed Jul. 24, 2023).
[5] “RPAL documentation.” [Online]. Available: https://rpal.sourceforge.net/doc/intro.pdf

Appendix - Class diagrams for packages

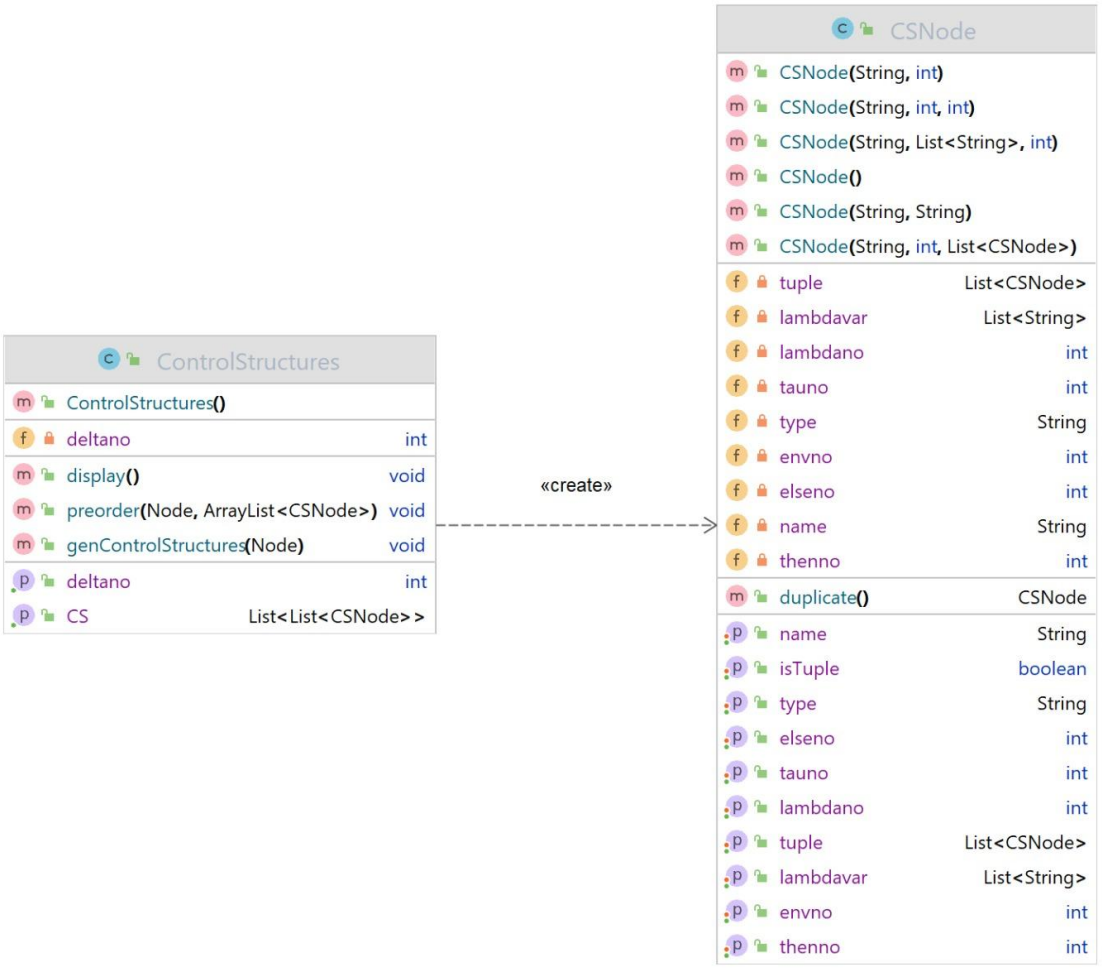
lexical_analyzer package



Parser package



control_structures package



cse_machine package