CS 3513 - Programming Languages

Programming Project 01 - RPAL Interpreter

Group Members – Group 125

- Jayasooriya J.M.D.C – 210252K
- Pabasara K.A.Y – 210442T

Problem Overview

Implement a lexical analyzer and a parser for the RPAL (Right-reference Pedagogic Algorithmic Language). Refer the RPAL_Lex for the lexical rules and RPAL_Grammar for the grammar details.

Output of the parser should be the Abstract Syntax Tree (AST) for the given input program.

Implement an algorithm to convert the Abstract Syntax Tree (AST) in to Standardize Tree (ST) and implement CSE machine.

Program should be able to read an input file which contains a RPAL program and return Output which should match the output of myrpal.exe for the relevant program.

Our implementation

Language & Tools

Programming Language: Java (Version: java 18.0.1.1 2022-04-22)

Development & Testing: Eclipse, Visual Studio Code, Command line

Program Execution instructions.

The following are the Instructions to run the project in a command line:

- Your local machine must be able to run make command in command line, and have java installed.
- Run `make` in the root directory.
- Put your RPAL test programs in the root directory. We had added the input.txt to the root directory, which contains the sample input program of the Project_Requirements document.
- For more test programs look at the test-programs directory.
- The following are the available commands that can be used for running different RPAL programs:
- Run `<path>\Group125>java myrpal <file_name>` for running the file which contains any RPAL programs and print only the Output.
- Run `<path>\Group125>java rpal20 -ast <file>` for running the file and print AST (Abstract Syntax Tree).

- Run `<path>\Group125>java rpal20 -st <file>` for running the file and print ST (Standardized Tree).
- `-ast` and `-st` are case insensitive when calling them.
- You can run it with argument passing in any other suitable environments too.

Structure of the program

Basically our program contains seven java packages.

- Lexical_analyaer
- Parser
- Standardizer
- Engine
- Symbols
- Exception
- test-programs

This document outlines the purpose of each of the above parts and presents the function prototypes along with their uses in each part.

Lexical analyzer:

This package contains mainly these 3 java files.

- LexicalAnalyser.java
- Token.java
- TokenType.java

✓ **LexicalAnalyser.java:**

Class Declaration: **`LexicalAnalyser`** is the main class responsible for tokenizing the input source code.

Fields:
`inputFileName`: Stores the name of the input file to be tokenized.
`tokens`: A list to store the generated tokens.

The **LexicalAnalyser** class reads an input file line by line, tokenizes each line based on predefined patterns using regular expressions, and stores the tokens in a list. It handles various token types such as identifiers, keywords, integers, operators, strings, punctuation, spaces, and comments. If an unrecognized character is encountered, it throws a CustomException with a detailed error message.

Tokenization Method:

scan Method: This method reads the input file line by line and tokenizes each line.

- **BufferedReader:** Used to read the file.
- **lineCount:** Keeps track of the current line number.
- **try-catch Block:** Handles any `IOException` during file reading and wraps `CustomException` to provide more context.

Tokenization Logic:

It is implemented in the method **tokenizeLine.**
For tokenization we use the lexical rules mention in the **lexer.pdf.** According to that, our method contains,

Token Patterns:

- `digit`, `letter`: Basic character patterns.
- `operatorSymbol`, `escape`: Patterns for operators and escape sequences.
- `identifierPattern`, `integerPattern`, `operatorPattern`, `punctuationPattern`, `spacesPattern`, `stringPattern`, `commentPattern`: Patterns for various token types.

Tokenization Loop:

- **Spaces and Comments:** Skip spaces and comments.
- **Identifiers:** Match identifiers and check if they are keywords or regular identifiers.
- **Integers:** Match integer literals.
- **Operators:** Match operator symbols.
- **Strings:** Match string literals.
- **Punctuation:** Match punctuation characters.
- **Exception Handling:** Throws a `CustomException` if a character cannot be tokenized.

Apart from this major file there are two more files in the lexical_analyzer package,

They are,

✓ **TokenType.java** : Here, **TokenType** enum is used by the lexical analyzer to categorize each token it generates. By assigning a specific type to each token

and

✓ **Token.java :** The **`Token`** class provides a simple representation of a token produced by the lexical analyzer. Each token consists of a type and a value. This class facilitates the communication between the lexical analyzer and other components of the interpreter, such as the parser, by encapsulating the necessary information about each token in an organized manner.

Parser

This package contains mainly 3 java files.

- Parser.java
- Node.java
- NodeType.java

✓ **Parser.java**

Class Declaration: Defines a class named `Parser`.

Fields:

- `tokens`: Stores the list of tokens to be parsed.
- `AST`: Stores the resulting Abstract Syntax Tree.
- `stringAST`: Stores the string representation of the AST.

The class contains several methods that handle parsing according to the grammar rules of the RPAL language. These methods are named after the non-terminal symbols of the grammar.

Utility Methods

- `convertAST_toStringAST()`: Converts the AST into its string representation.
- `addStrings()`: Adds string representations of nodes to the string AST.

Parsing Rules

The parsing methods (`E()`, `Ew()`, `T()`, etc.) implement the grammar rules of the RPAL language. Each method parses a specific part of the input tokens and constructs corresponding nodes in the AST.

Error Handling

The parser includes basic error handling, printing error messages when parsing errors occur.

Thus, the Parser class serves as the core component of the interpreter, responsible for converting token streams into an AST representation of the input program. It follows the grammar rules of the RPAL language to ensure accurate parsing.

✓ **Node.java**

The `Node` class is used by the `Parser` class to build the AST. For instance, when parsing expressions, the parser creates nodes for operators, literals, and variables, and links them according to the grammar rules.

Fields:

- `type`: An enumeration `NodeType` that represents the type of the node (e.g., identifier, operator).
- `value`: A string representing the value associated with the node (e.g., the actual identifier name or the operator symbol).
- `noOfChildren`: An integer indicating the number of children nodes (used to represent the structure of the tree).

✓ **NodeType.java**

The `NodeType` enum provides a comprehensive list of all possible node types that the parser can recognize and use to build the AST according to tree transduction grammar in RPAL. Each enum constant represents a specific kind of syntax element, such as an operator, a literal, or a control structure.

Standardizer

In the standardizer package there are three main java classes.

- AST.java
- ASTFactory.java
- Node.java
- NodeFactory.java

✓ **AST.java**

The `AST` class in the `Standardizer` package represents an Abstract Syntax Tree. This class provides methods to set and get the root node, standardize the tree, and traverse the tree in pre-order for printing purposes.

- **Class Declaration:** The `AST` class.
- **Field `root`:** A private field representing the root node of the AST.
- **`setRoot` Method:** Sets the root node of the AST.
- **`getRoot` Method:** Returns the root node of the AST.
- **`standardize` Method:** Checks if the root node is already standardized. If not, it calls the `standardize` method on the root node

`preOrderTraverse` Method: A private method that recursively traverses the AST in pre-order.

Functionality:

- Prints dots corresponding to the depth of the current node.
- Prints the data of the current node.
- Recursively calls itself for each child of the current node

`printAst` Method: Initiates a pre-order traversal starting from the root node, printing the entire AST.

✓ **AST Factory.java**

The `ASTFactory` class is responsible for creating an Abstract Syntax Tree (AST) from a list of strings that represent nodes with their respective depths.

Method `getAbstractSyntaxTree`: Converts a list of strings into an AST.

- **Parameters:**
  - `ArrayList<String> data`: A list of strings representing the nodes of the AST. Each string starts with a number of dots representing the depth of the node.
- **Returns:**
  - An `AST` object representing the constructed abstract syntax tree.
- **Local Variables:**
  - `Node root`: The root node of the AST, created from the first element in `data`.
  - `Node previous_node`: Keeps track of the previously processed node.
  - `int current_depth`: Tracks the depth of the current node being processed.

✓ **Node.java**

This `Node` class is designed specifically for standardizing purposes within the `Standardizer` package. It differs from the nodes used in the `Parser` package by having additional functionalities required for the standardization process of an Abstract Syntax Tree (AST).

Class Declaration: The `Node` class for standardizing ASTs.

Fields:

- `data`: Stores the data of the node.
- `depth`: Represents the depth of the node in the tree.
- `parent`: The parent node.
- `children`: A list of child nodes.
- `isStandardized`: A flag to check if the node has been standardized.

Getters and Setters: Methods to access and modify the fields.

Standardize Method: Recursively standardizes the node and its children based on the rules in the language.

Helper Methods for Standardization

`Standardizelet` Method : Standardizes `let` nodes: Transforms `let` nodes to a standardized form using lambda and gamma nodes.

`standardizeWhere` Method : Standardizes `where` nodes: Converts `where` nodes into `let` nodes and calls `standardize` on them.

`standardizeFunctionForm` Method: **Standardizes `function_form` nodes:** Transforms `function_form` nodes into nested lambda expressions.

`standardizeRec` Method: Standardizes rec nodes: Transforms rec nodes into a standardized form involving gamma and lambda nodes.

`standardizeAnd` Method: **Standardizes `and` nodes:** Transforms `and` nodes into a standardized form using `comma` and `tau` nodes.

`standardizeAt` : **Standardizes `@` nodes:** Transforms `@` nodes into a standardized form using `gamma` nodes.

`standardizeWithin` : **Standardizes `within` nodes:** Transforms `within` nodes to a standardized form involving `gamma` and `lambda` nodes.

Thus, The `Node` class in the `Standardizer` package is designed to facilitate the standardization of AST nodes. It includes methods to set and get node attributes, as well as a `standardize` method that processes nodes according to specific rules. The standardization process involves transforming various types of nodes (`let`, `where`, `function_form`, `lambda`, `within`, `@`, `and`, `rec`) into their standardized forms using helper methods.

✓ NodeFactory.java

Class Declaration: The `NodeFactory` class, which provides static methods to create nodes.

The `NodeFactory` class provides methods to create instances of the `Node` class in the `Standardizer` package. This class contains overloaded static methods to generate nodes with varying levels of detail, facilitating the construction of standardized abstract syntax trees.

Static Methods for Node Creation:

`getNode` Method (Simpler Version):

- **Parameters:**
    - `data`: The data to be stored in the node.
    - `depth`: The depth of the node in the tree.

- **Description:** Creates a new `Node` with the specified data and depth. The children list is initialized as an empty `ArrayList`.

`getNode` Method (Advanced Version):

- **Parameters:**
    - `data`: The data to be stored in the node.
    - `depth`: The depth of the node in the tree.
    - `parent`: The parent node.

- `children`: The list of child nodes.
- `isStandardize`: A boolean flag indicating whether the node is standardized.

- **Description:** Creates a new `Node` with detailed specifications, including data, depth, parent, children, and standardization status.

Engine

Engine package has mainly three java classes.

- CSEMachine.java
- CSEMachineFactor.java
- Evaluator.java

✓ **CSEMachine.java**

Class Declaration: The `CSEMachine` class, responsible for executing computations.

The CSEMachine class represents a CSE (Control, Stack, Environment) machine, which is used to execute computations based on control instructions, a stack, and an environment.

Instance Variables:

- `control`: An `ArrayList` of symbols representing the control instructions.
- `stack`: An `ArrayList` of symbols acting as a stack for computations.
- `environment`: An `ArrayList` of environments (`E` objects) for maintaining variable bindings.
- **Setter Methods:** Setters for the control, stack, and environment.
- **Execution Method:** Executes computations based on the control instructions, stack operations, and environment bindings.

Helper Methods

- `applyUnaryOperation`: Applies unary operations like negation and logical negation.
- `applyBinaryOperation`: Applies binary operations like addition, subtraction, etc.
- `getTupleValue`: Gets the value of a tuple represented by `Tup` symbols.
- `getAnswer`: Executes the machine and returns the result.

Printing Methods

- `printControl`: Prints the contents of the control.
- `printStack`: Prints the contents of the stack.
- `printEnvironment`: Prints the contents of the environment.

✓ **CSEMachineFactory.java**

Class Declaration: The `CSEMachineFactory` class, responsible for constructing `CSEMachine` instances.

The `CSEMachineFactory` class is responsible for constructing instances of the `CSEMachine` class by processing abstract syntax trees (ASTs) generated by the `Standardizer` package. It converts AST nodes into symbols, lambdas, deltas, and other components required for initializing a `CSEMachine` instance.

Instance Variables:

- `e0`: Instance of environment `E` with index 0.
- `i`: Counter for generating unique indices for lambdas.
- `j`: Counter for generating unique indices for deltas.

Methods

- `getSymbol`: Converts a `Node` to a `Symbol` object.
- `getB`: Constructs a `B` object from a `Node`.
- `getLambda`: Constructs a `Lambda` object from a `Node`.
- `getPreOrderTraverse`: Traverses the AST in preorder and constructs symbols accordingly.
- `getDelta`: Constructs a `Delta` object from a `Node`.
- `getControl`: Constructs the control list for a `CSEMachine`.
- `getStack`: Constructs the stack for a `CSEMachine`.
- `getEnvironment`: Constructs the environment list for a `CSEMachine`.
- `getCSEMachine`: Constructs a `CSEMachine` instance from an AST.

✓ **Evaluator.java**

Class Declaration: The `Evaluvator` class, responsible for evaluating programs.

The `Evaluvator` class is responsible for evaluating programs by performing lexical analysis, parsing, abstract syntax tree (AST) construction, standardization, and executing the resulting AST using a `CSEMachine`. It provides a static method `evaluvate` that takes a filename, performs the evaluation process, and returns the result of the evaluation.

Static Method: `evaluvate`

```
    public static String evaluvate(String filename, boolean
isPrintAST, boolean isPrintST){…}
```

- **Method Signature:** `evaluvate` is a static method that takes a filename, a boolean flag to indicate whether to print the AST, and another boolean flag to indicate whether to print the standardized AST.
- **Parameters:**
    - `filename`: The name of the file containing the program to be evaluated.
    - `isPrintAST`: A boolean flag indicating whether to print the AST.
    - `isPrintST`: A boolean flag indicating whether to print the standardized AST.
- **Return Type:** `String`: The result of evaluating the program.

Method Body

- **Lexical Analysis and Parsing:**
    - Lexical analysis is performed using a `LexicalAnalyser`, which generates a list of tokens.
    - Parsing is performed using a `Parser`, which generates an AST.
- **AST Construction:**
    - An `ASTFactory` is used to create an abstract syntax tree (AST) from the string representation of the AST obtained from the parser.
- **Standardization:**
    - The AST is standardized using the `standardize` method.
- **Printing AST:**
    - If the `isPrintAST` flag is `true`, the string representation of the AST is printed.
- **CSEMachine Creation:**
    - A `CSEMachineFactory` is used to create a `CSEMachine` instance from the standardized AST.
- **Execution and Result:**
    - The `getAnswer` method is called on the `CSEMachine` instance to execute the AST and obtain the result of evaluation.
- **Exception Handling:**
    - Custom exceptions are caught and their messages are printed.
- **Return:**
    - The result of the evaluation is returned.

Exception

Class Declaration: The `CustomException` class extends the Java `Exception` class, making it a checked exception.

The `CustomException` class defines a custom exception that can be used to represent various exceptional conditions in a Java program. It extends the standard `Exception` class and provides constructors for creating instances with custom messages and causes.

Constructors:

- The class provides multiple constructors to create instances of the exception with different messages and causes.

- Each constructor invokes a corresponding constructor of the superclass (`Exception`) using the `super` keyword.

Symbols

Symbols package contains the representations of various symbols used in the CSE machine.

test-programs.

This directory contains various RPAL test programs that can be used to test the interpreter.