CS 3513 - Programming Languages
Programming Project 01

- Group: Hashira
- Members:
    - 210172N - Galappaththi A. S.
    - 210266G - Kahapola K. V.

Problem Description

The task involves developing a lexical analyzer and parser for the RPAL language without relying on external tools like 'lex' or 'yacc'. The lexical rules and grammar details for RPAL are provided in separate documents. The output of the parser should be an Abstract Syntax Tree (AST) representing the input program.

Furthermore, the implementation must include an algorithm to convert the AST into a Standardized Tree (ST), followed by the implementation of a CSE machine. The program should read an input file containing a RPAL program and produce output consistent with the output generated by "rpal.exe" for the corresponding program.

The implementation was carried out using Java.

Execution Instructions

- **Compilation:**
    - Ensure you have Java installed on your system. If not, download and install it from the official website: Java Downloads. (Version 20 was used.)
    - Compile the Java source code using the following command
        - `> javac myrpal.java`

- **Execution:**
    - Basic Execution:
        - Execute the program without any command-line arguments to analyze the sample RPAL program provided (sample.txt):
            - `> java myrpal`
        - Specify Input File: To analyze a specific RPAL program file, provide the file path as a command-line argument:
            - `> java myrpal <input_file>`
        - Print AST: Optionally, you can request to print the Abstract Syntax Tree (AST) for the input program by including the -ast flag as the first argument:

- `> java myrpal -ast <input_file>`

- **Output:**
    - The program will output the result of evaluating the RPAL program. If the -ast flag is used, it will also print the corresponding AST.

Ensure that the input RPAL program file exists and is accessible from the location specified. Follow the provided instructions to accurately execute and analyze RPAL programs using the implemented Java program.

Structure

The project adopts a modular architecture, organized into distinct packages, each serving a specific role in the evaluation and processing of RPAL (Recursive Programming in Algorithmic Language) programs. This modular design provides code clarity, maintainability, and extensibility, facilitating future enhancements.

- CSEMachine:
    - Contains classes related to the CSE (Context-Sensitive Expression) machine, responsible for evaluating RPAL (Recursive Programming in Algorithmic Language) programs.
    - Includes classes such as `ProgramEvaluator`, `CSEMachine`, `E`, `Delta`, and various symbol classes.

- Lexer:
    - Handles lexical analysis of RPAL programs, breaking down the input into tokens.
    - Includes the `Lexer` class responsible for tokenizing the input and the `Token` and `TokenType` classes to represent tokens and their types.

- Parser:
    - Manages the parsing of RPAL programs, converting tokens into an abstract syntax tree (AST) representation.
    - Contains the `Parser` class, which parses tokens into an AST, and the `Node` and `NodeType` classes to represent nodes in the AST.

- Symbols:
    - Contains classes representing symbols used in RPAL programs, such as identifiers, integers, strings, operators, lambdas, and others.
    - Includes classes like `Symbol`, `Lambda`, `Int`, `Bool`, `Bop`, `Uop`, and more.

- Main Class - myrpal:
    - Serves as the entry point to the application.

    - Parses command-line arguments, determines whether to print the AST, and invokes the `ProgramEvaluator` to evaluate RPAL programs.

- Other Files:
    - `sample.txt`: A sample RPAL program provided for testing and demonstration purposes.

- Overall Interaction:
    - The `Lexer` package tokenizes the input RPAL program.
    - The `Parser` package parses the tokens into an abstract syntax tree (AST).
    - The `ProgramEvaluator` in the `CSEMachine` package then evaluates the AST using the CSE machine.
    - The `Main` class coordinates the entire process, taking user input and displaying the output, including the possibility to print the AST for debugging purposes.

Function Prototypes and Program Structure

- Abstract Syntax Tree (AST):
    - ASTNode Class:
        - `public ASTNode(NodeType type, String value)`
            - Constructor to create a new AST node with the specified type and value.
        - `public void addChild(ASTNode child)`
            - Adds a child node to the current node.
        - `public List<ASTNode> getChildren()`
            - Returns the list of child nodes of the current node.
        - Represents nodes in the abstract syntax tree (AST) used for representing the syntactic structure of the input program.

- Control-Stack-Environment (CSE) Machine:
    - CSEMachine Class:
        - `public CSEMachine(ArrayList<Symbol> control, ArrayList<Symbol> stack, ArrayList<E> environment)`
            - Constructor to create a new CSE machine with the specified control, stack, and environment.
        - `public void execute()`
            - Executes the CSE machine until termination.
        - Represents the control-stack-environment machine used for interpreting and executing the input program.
- Lexer Class:
    - `public List<Token> tokenize()`
        - Tokenizing the input program and returning a list of tokens.

    - `private void processLine(String line, int lineCount)`
        - Processes each line of the input program to identify tokens. This method implements the lexer logic.
    - Contains various helper methods for identifying different types of tokens such as identifiers, integers, strings, operators, etc.

- Token Class:
    - `public Token(TokenType type, String value)`
        - Constructor to create a new token with the specified type and value. Represents individual tokens identified by the lexer.

- TokenType Enum:
    - Enumerates different types of tokens such as KEYWORD, IDENTIFIER, INTEGER, OPERATOR, STRING, PUNCTUATION, DELETE, and EndOfTokens. Provides a structured way to classify tokens.

- Parser Class:
    - `public List<Node> parse()`
        - Parses the tokenized input program and constructs the Abstract Syntax Tree (AST).
    - `public ArrayList<String> ASTtoString()`
        - Converts the AST to a list of strings representing its structure. Helps in visualizing the AST.
    - Contains various methods for parsing different parts of the input program such as expressions, declarations, functions, etc. Implements the parser logic.

- Node Class:
    - `public Node(NodeType type, String value, int children)`
        - Constructor to create a new node with the specified type, value, and number of children. Represents nodes in the Abstract Syntax Tree.

- NodeType Enum:
    - Enumerates different types of AST nodes such as let, fcn_form, identifier, integer, string, where, gamma, lambda, tau, rec, aug, conditional, etc. Provides a structured way to classify AST nodes.

- Symbol Class:
    - `public Symbol(String data)`
        - Constructor to create a new symbol with the specified data. Represents symbols used in the CSE machine.

- Lambda Class:
    - `public Lambda(int i)`

        - Constructor to create a new lambda symbol with the specified index. Represents lambda symbols used in the CSE machine.

- Int Class:
    - `public Int(String data)`
        - Constructor to create a new integer symbol with the specified data. Represents integer symbols used in the CSE machine.

- Main Class (myrpal):
    - `public static void main(String[] args)`
        - Entry point of the program. Parses command-line arguments and invokes the ProgramEvaluator to evaluate the input program. Provides a high-level overview of the program structure and functionality.

- Additional Symbol Classes:
    - Includes classes such as `B`, `Beta`, `Bool`, `Bop`, `Delta`, `Dummy`, `E`, `Err`, `Eta`, `Gamma`, `Id`, `Rand`, `Rator`, `Str`, `Symbol`, `Tau`, `Tup`, `Uop`, `Ystar`, each serving specific purposes in the project.

This modular structure facilitates maintainability, extensibility, and readability of the codebase by organizing it into distinct components with well-defined responsibilities. Each class and method contributes to the overall functionality of the RPAL interpreter, ensuring clarity and coherence in the implementation.

Summary

The RPAL interpreter project encapsulates the development of a modular system for interpreting programs written in the RPAL language. It comprises components such as lexer, parser, AST, and CSE machine, all working together seamlessly.

The lexer breaks down input code into tokens, which are then parsed by the parser to construct an abstract syntax tree (AST). This tree structure captures the program's syntactic hierarchy. The CSE machine executes the code efficiently using a stack-based approach.

Extensive testing and modularity were key aspects of the project, ensuring its correctness, reliability, and ease of use.

Overall, the RPAL interpreter project demonstrates proficiency in language interpretation and software engineering principles, providing a robust solution for interpreting RPAL programs.

Appendix

- RPAL's Phrase Structure Grammar:

```
# Expressions ################################################

E    -> 'let' D 'in' E                                  => 'let'
     -> 'fn'  Vb+ '.' E                                 => 'lambda'
     ->  Ew;
Ew   -> T  'where' Dr                                   => 'where'
     -> T;

# Tuple Expressions ##########################################

T    -> Ta ( ',' Ta )+                                  => 'tau'
     -> Ta ;
Ta   -> Ta 'aug' Tc                                     => 'aug'
     -> Tc ;
Tc   -> B '->' Tc '|' Tc                                => '->'
     -> B ;

# Boolean Expressions ########################################

B    -> B 'or' Bt                                       => 'or'
     -> Bt ;
Bt   -> Bt '&' Bs                                       => '&'
     -> Bs ;
Bs   -> 'not' Bp                                        => 'not'
     -> Bp ;
Bp   -> A ('gr' | '>' ) A                               => 'gr'
     -> A ('ge' | '>=') A                               => 'ge'
     -> A ('ls' | '<' ) A                               => 'ls'
     -> A ('le' | '<=') A                               => 'le'
     -> A 'eq' A                                        => 'eq'
     -> A 'ne' A                                        => 'ne'
     -> A ;
```

```
# Arithmetic Expressions ###################################

A     -> A '+' At                                    => '+'
      -> A '-' At                                    => '-'
      ->   '+' At
      ->   '-' At                                    => 'neg'
      -> At ;
At    -> At '*' Af                                   => '*'
      -> At '/' Af                                   => '/'
      -> Af ;
Af    -> Ap '**' Af                                  => '**'
      -> Ap ;
Ap    -> Ap '@' '<IDENTIFIER>' R                     => '@'
      -> R ;
# Rators And Rands #########################################

R     -> R Rn                                        => 'gamma'
      -> Rn ;
Rn    -> '<IDENTIFIER>'
      -> '<INTEGER>'
      -> '<STRING>'
      -> 'true'                                      => 'true'
      -> 'false'                                     => 'false'
      -> 'nil'                                       => 'nil'
      -> '(' E ')'
      -> 'dummy'                                     => 'dummy' ;


# Definitions ##############################################

D     -> Da 'within' D                               => 'within'
      -> Da ;
Da    -> Dr ( 'and' Dr )+                            => 'and'
      -> Dr ;
Dr    -> 'rec' Db                                    => 'rec'
      -> Db ;
Db    -> Vl '=' E                                    => '='
      -> '<IDENTIFIER>' Vb+ '=' E                    => 'fcn_form'
      -> '(' D ')' ;

# Variables ################################################

Vb    -> '<IDENTIFIER>'
      -> '(' Vl ')'
      -> '(' ')'                                     => '()';
Vl    -> '<IDENTIFIER>' list ','                     => ','?;
```