CS 3513 - Programming Languages

Programming Language Project

Project Report

Group 46

210173T - GALLAGE D.
210572P - SANJANA K.Y.C.

Introduction:

In this project, we were tasked with implementing a lexical analyzer and a parser for the RPAL language. The RPAL language is defined by specific lexical rules and grammar, which we followed as outlined in the documents provided: RPAL_Lex.pdf and RPAL_Grammar.pdf.

Problem Description:

Implement a lexical analyzer and a parser for the RPAL language. (Refer RPAL_Lex.pdf for the lexical rules and RPAL_Grammar.pdf for the grammar details.)

Output of the parser should be the Abstract Syntax Tree (AST) for the given input program. Then implement an algorithm to convert the Abstract Syntax Tree (AST) into Standardize Tree (ST) and implement a CSE machine.

The program should be able to read an input file which contains a RPAL program. Output of the program should match the output of “rpal.exe“ for the relevant program.

Approach:

Our solution follows these main steps:

- **Lexical Analysis:**
  We implemented a lexical analyzer in python. This analyzer tokenizes the input RPAL program based on the lexical rules specified.

- **Parsing:**
  Using the lexical tokens generated, we built a parser to validate and parse the RPAL syntax according to the grammar rules provided. This parser constructs an Abstract Syntax Tree (AST) representing the structure of the input program.

- **AST to ST Conversion:**
  After generating the AST, we implemented an algorithm to convert it into a Standardized Tree (ST). This process simplifies the AST while preserving the semantics of the RPAL program.

- **Control Structure Generation:**
  We devised a mechanism to generate control structures necessary for efficient execution of RPAL programs, ensuring proper handling of control flow constructs such as conditionals and loops.

- **CSE Machine:**
  Finally, we developed a CSE (Control Stack Environment) machine in python to execute RPAL programs based on the generated ST.

File Structure:

The RPAL interpreter project comprises several critical components, each serving an essential role in interpreting RPAL programs:

1. ***test_cases***: Directory that contains various test cases for validating the interpreter's functionality across diverse scenarios.

2. ***output_files***: Directory designated for storing output, utilized for debugging and testing purposes.

3. ***Tokernizer.py***: Manages the tokenization process of the input RPAL program into meaningful units (tokens).

4. ***ASTNode.py***: Responsible for managing the creation, manipulation, and normalization of nodes within the Abstract Syntax Tree (AST).

5. ***controlStructure.py***: Responsible for the generation and administration of the control structures within the interpreter.

6. ***Envronment.py***: Handles environment configurations such as variable bindings, crucial for the execution phase.

7. ***cseMachine.py***: Implements the Control Structure Environment Machine, essential for executing the standardized tree.

8. ***myrpal.py***: The primary executable script for the interpreter, with the parser.

9. ***Makefile***: Contains rules and commands for automating tasks such as compilation and execution.

10. ***test.py***: Script facilitating the execution of test cases.

Program Structure:

Lexical Analyzer (Scanner):

The lexical analyzer for the RPAL language reads the input source code character by character and groups them into tokens based on predefined lexical rules. These tokens represent meaningful units like identifiers, keywords, operators, constants, and punctuation symbols. The lexical analyzer discards irrelevant characters like whitespace and comments and detects errors for invalid tokens or syntax. It prepares the input for further processing by the parser.

File: `Tokernizer.py`

Here are the function prototypes of the lexer:

1. Tokenizer.error
    - Parameters: `None`
    - Returns: `None`
    - Description: Raises an exception for tokenization errors.

2. Tokenizer.advance
    - Parameters: `None`
    - Returns: `None`
    - Description: Moves to the next character in the source code.

3. Tokenizer.skip_whitespace
    - Parameters: `None`
    - Returns: `None`
    - Description: Skips whitespace characters in the source code.

4. Tokenizer.[token_functions]
    - Parameters: `None`
    - Returns: Varies depending on the specific token function.
    - Description: Parses and returns tokens of different types such as integers, identifiers, comments, and strings.

5. Tokenizer.get_next_token
    - Parameters: `None`
    - Returns: `Token` - The next token in the source code.
    - Description: Retrieves the next token from the source code.

6. Screener.merge_tokens
    - Parameters: `None`
    - Returns: `None`
    - Description: Merges certain consecutive tokens, such as specific operators, into single tokens for better token representation.

7. Screener.remove_comments
    - Parameters: `None`
    - Returns: `None`
    - Description: Removes comment tokens from the list of tokens.

8. Screener.screen_reserved_keywords
    - Parameters: `None`
    - Returns: `None`
    - Description: Identifies and marks reserved keywords in the token list.

9. Screener.screen
    - Parameters: `None`
    - Returns: `List[Token]` - The list of tokens after screening and merging.
    - Description: Screens the tokens and performs token merging to enhance the tokenization process.

Parser:

The parser analyzes the tokens produced by the lexical analyzer and constructs an Abstract Syntax Tree (AST) representing the syntactic structure of the RPAL program. It ensures adherence to the grammar rules specified for the language, handling various constructs such as expressions, tuples, boolean expressions, and definitions. By recursively parsing the input, the parser organizes the tokens into a hierarchical tree, facilitating further processing for execution or code generation.

File: `myrpal.py`

Here are the function prototypes of the parser:

1. ASTParser.read
    - Parameters: `None`

- Returns: `None`
- Description: Reads the next token from the token stream.

2. ASTParser.buildTree
   - Parameters:
     `token (str)` - The type of node to create.
     `ariness (int)` - The arity of the node.
   - Returns: `None`
   - Description: Builds a node in the abstract syntax tree (AST) based on the given token and arity.

3. ASTParser.[process_functions]
   - Parameters: `None`
   - Returns: `None`
   - Description: Parses various elements of the RPAL language according to the grammar rules.

AST to ST Converter:

The AST to ST Converter is a module designed to transform Abstract Syntax Trees (ASTs) into Standardized Trees (STs). This conversion process streamlines the AST structure into a standardized format, facilitating subsequent analysis and optimization steps in the RPAL language processing pipeline.

File: `ASTNode.py`

Here are the function prototypes of the converter:

1. ASTNode.standarize
   - Parameters: `root (ASTNode)` - The root node of the abstract syntax tree to standardize.
   - Returns: `ASTNode` - The root node of the standardized abstract syntax tree.
   - Description: Recursively standardizes the AST based on the type of the node.

2. ASTNode.print_tree
   - Parameters: `None`

- Returns: `None`
- Description: Prints the abstract syntax tree to the console.

3. ASTNode.print_tree_to_cmd
   - Parameters: `None`
   - Returns: `None`
   - Description: Prints the abstract syntax tree to the console with indentation.

4. ASTNode.print_tree_to_file
   - Parameters: `file (file)` - The file object to write the abstract syntax tree.
   - Returns: `None`
   - Description: Prints the abstract syntax tree to the specified file with indentation.

5. ASTNode.createCopy
   - Parameters: `None`
   - Returns: `ASTNode` - A copy of the ASTNode object.
   - Description: Creates a copy of the ASTNode object.

Control Structure Generator:

Control Structure Generation involves the creation or transformation of control flow structures within a program. This process typically includes the generation of constructs like conditionals, loops, and function calls, ensuring that the program's logic and execution flow are properly structured and efficient. In essence, Control Structure Generation aims to organize and manage the flow of program execution, enhancing its readability, maintainability, and performance.

File: `controlStructure.py`

Here are the function prototypes of the control structure generator:

1. ControlStructureGenerator.print_ctrl_structs
   - Parameters: `None`
   - Returns: `None`
   - Description: Prints the control structures generated from the AST.

2. ControlStructureGenerator.generate_control_structures
    - Parameters: `root (ASTNode)` - The root node of the AST.
    - Returns: `dict` - A dictionary containing the generated control structures.
    - Description: Generates control structures from the AST using preorder traversal.

3. ControlStructureGenerator.pre_order_traversal
    - Parameters:
        `root (ASTNode)` - The current node being traversed.
        `delta (list)` - List of elements associated with the current control structure.
    - Returns: `None`
    - Description: Traverses the AST in a pre-order fashion and generates the control structures.

CSE Machine:

The CSE (Compiled Stack Environment) Machine is a crucial component of RPAL program execution. It utilizes a stack-based architecture to efficiently execute RPAL code. By implementing optimizations like memoization and lazy evaluation, it enhances performance by reducing redundant computations and delaying expression evaluation until necessary. Overall, the CSE Machine ensures reliable and efficient execution of RPAL programs.

Files: `Envronment.py` and `cseMachine.py`

Here are the function prototypes of the CSE machine:

1. Environment.set_env_params
    - Parameters:
        `parent_env (Environment)` - The parent environment.
        `key` - The key to set.
        `value` - The value to set for the key.
    - Returns: `None`
    - Description: Sets parameters for the environment.

2. Environment.get_env_idx
    - Parameters: `None`

- Returns: `int` - The index of the environment.
- Description: Retrieves the index of the environment.

3. Environment.get_val
    - Parameters: `key` - The key to retrieve the value for.
    - Returns: The value associated with the key.
    - Description: Retrieves the value associated with the key from the environment.

4. CSEMachine.binOp
    - Parameters:
        `op` - The binary operator.
        `rand1` - The first operand.
        `rand2` - The second operand.
    - Returns: The result of the binary operation.
    - Description: Performs a binary operation.

5. CSEMachine.unaryOp
    - Parameters:
        `op` - The unary operator.
        `rand` - The operand.
    - Returns: The result of the unary operation.
    - Description: Performs a unary operation.

6. CSEMachine.Print
    - Parameters: `obj` - The object to print.
    - Returns: `None`
    - Description: Prints the object.

7. CSEMachine.execute
    - Parameters: `None`
    - Returns: `None`
    - Description: Executes the control structures.

Testing & Functionality:

Test Cases:

Located in the test_cases directory, these cases are crucial for validating the interpreter's functionality across various scenarios, identifying potential bugs and edge cases.

Running the Test Script:

To execute all test cases, run the following command in the terminal:
`python test.py`

Execution Script Usage:

To obtain only the output:
`python myrpal.py file_name`

To print the Abstract Syntax Tree (AST) in the command line:
`python myrpal.py -ast file_name`

Makefile:

A Makefile is provided to streamline common tasks such as running, testing, and cleaning the project. Below are the available targets:

- `run`: Executes the main script.
- `ast`: Prints the Abstract Syntax Tree (AST).
- `test`: Runs all test cases.
- `clean`: Removes any generated files or directories.

To utilize these targets, run the respective command preceded by `make` in the terminal.