RPAL Interpreter

Prepared by Group 116 of Intake-21 following CS3513

Department of Computer Science & Engineering,
University of Moratuwa.

25/03/2024

Group Number: 116

Authors:

- Fernando T.H.L. - 210167E
- Gamage M.S - 210176F

Table of Contents

Introduction

This report delves into the development process of an RPAL interpreter, built using Python, with a specific emphasis on parsing and executing RPAL programs. RPAL, characterized by its lexical rules and grammar specifications as outlined in RPAL_Lex.pdf and RPAL_Grammar.pdf, forms the foundation of this endeavor. The primary goal of this project is to create a robust software solution proficient in parsing RPAL code, constructing an Abstract Syntax Tree (AST), converting it into a Standardized Tree (ST), and executing RPAL programs using a Control Structure Environment (CSE) machine.

Problem Description

- Lexical Analysis and Parsing: The implementation of a lexical analyzer and parser for RPAL (Right-reference Pedagogic Algorithmic Language). Refer to RPAL_Lex for the lexical rules and RPAL_Grammar for grammar details. Additionally, consult About RPAL for information about the RPAL language.
- Abstract Syntax Tree (AST) Generation: The parser's output should be the Abstract Syntax Tree (AST) for the given input program.
- Conversion Algorithm: Implement an algorithm to convert the Abstract Syntax Tree (AST) into a Standardized Tree (ST) and implement the CSE machine. Refer to the semantics document, which contains the rules for transforming the AST into the ST.
- Input and Output Handling: The program should be capable of reading an input file containing an RPAL program and returning output that matches the output of rpal.exe for the relevant program.

For further details, refer to the Project_Requirements document.

Key Development Requirements

1. Lexical Analysis: Develop a lexical analyzer to tokenize RPAL programs based on predefined rules.
2. Parsing: Implement a parser to generate an Abstract Syntax Tree (AST) from tokenized RPAL code.
3. AST to ST Conversion: Design an algorithm to convert the AST into a Standardized Tree (ST) for efficient execution.
4. CSE Machine: Create a Control Structure Environment (CSE) machine to execute RPAL programs.
5. Input Handling: Ensure robust handling of RPAL program files as input.
6. Output Consistency: Guarantee that program output matches that of "rpal.exe" for corresponding inputs.
7. Language Choice: Implement the interpreter using Python, Java, or C/C++. (Python was chosen for this project)
8. Documentation: Provide clear and concise documentation for ease of understanding and maintenance.
9. -ast Switch: Support the -ast switch to print the Abstract Syntax Tree, fulfilling a specific requirement for program functionality.

About Our Solution

- Programming Language: Python
- Development & Testing: Visual Studio Code, Command Line, Cygwin, Pytest, GitHub Actions

By addressing these key development requirements, the RPAL interpreter will emerge as a reliable and efficient tool for parsing and executing RPAL programs, contributing to the broader landscape of programming language interpretation and execution.

Program Execution Instructions

Prerequisites

Ensure that Python and pip are installed on your local machine.

To use the RPAL-Interpreter, follow these steps:

1. Clone the repository to your local machine or download the project source code as a ZIP file.
2. Navigate to the root directory of the project in the command line interface.
3. Install the dependencies by running the following command in the project directory
   pip install -r requirements.txt
4. Place your RPAL test programs in the root directory. We have provided a sample input program test.txt, as specified in the Project Requirements.
5. Running the Program

- Run the main script `main.py` with the file name as an argument:
    - python main.py file_name
- To generate the Abstract Syntax Tree:
    - python main.py -ast file_name

Below are screenshots showcasing the functionality of switches controlling various features. Navigate to the docs/working_switches directory to view them.

Additional Switches for Analysis (Debugging)

You can use additional switches for various debugging purposes:

- To generate the token list from the lexical analyzer:
    - python main.py -t file_name
- To generate the filtered token list from the screen:
    - python main.py -ft file_name
- To generate the Standardized Tree:
    - python main.py -st file_name

Using Make Commands (Alternative Method)

Note: Your local machine must be able to run the make command.

For Windows Users

If you are using Windows, you may need to install Cygwin or similar Unix-like environment tools to execute the make command.

Alternatively, you can use the following make commands:

- Install Dependencies:
    - *make install*
- Run Program (test.txt):
    - *make run*
- Run Tests:
    - Run all tests:
        - *make test_all*
    - Run tests for the final result:
        - *make test*
    - Run tests for AST results:
        - *make test_ast*
    - Run tests for ST result:
        - *make test_st*
    - All in One (Install, Run, Test(test_all)):
        - *make all*

Note for Python 3 Users

If you have both Python 2 and Python 3 installed, you may need to use `python3` instead of `Python` in the commands above. Similarly, use `pip3` instead of `pip` for installing packages.

Structure of the Project

The RPAL interpreter project is organized into several components, each responsible for a specific aspect of the parsing and execution process:

1. **Lexical Analyzer:**

- **Description:** Tokenizes the input RPAL program based on defined lexical rules.

- **Functionality:**
  - Scans the RPAL source file.
  - Identifies tokens based on RPAL lexical rules.
  - Outputs a token list containing token objects with type and value attributes.

- **Input:** RPAL source file
- **Output:** Token list (list of token objects)

- **Package:** lexical_analysis
- **Module:** scanner.py (located in the lexical_analysis package)

2. **Screener:**

- **Description:** Further processes the token list generated by the Lexical Analyzer by filtering out unnecessary tokens.

- **Functionality:**
  - Filters the token list to remove unnecessary tokens.
  - Identifies keywords and categorizes them appropriately.
  - Outputs a filtered token list suitable for parsing.

- **Input:** Token list generated by the Lexical Analyzer
- **Output:** Filtered token list (list of token objects with attributes: type, value)

- **Package:** screening
- **Module**: screener.py (located in the screening package)

3. **Parser:**

- **Description:** Constructs the Abstract Syntax Tree (AST) and Standard Tree (ST) from the token list generated by the Lexical Analyzer.

- **Functionality:**
  - Receives the token list from the Lexical Analyzer.
  - Construct the Abstract Syntax Tree (AST) and Standard Tree (ST) based on the provided tokens.
  - Outputs the Standard Tree (ST).

- **Input:** Token list generated by the Screener
- **Output:** Standard Tree (ST)

- **Package:** parser
- **Module**: parser_module.py (located in the parser package)

4. **CSE Machine:**

- **Description:** Executes the RPAL source program by traversing the Standard Tree (ST) in a pre-order manner and applying the 13 CSE rules.

- **Functionality:**
  - Traverses the Standard Tree (ST) in a pre-order manner.
  - Generates control structures during traversal.
  - Applies the 13 CSE rules to evaluate the source program.
  - Produces the output of the source program.

- **Input:** Standard Tree (ST)
- **Output:** Output of the source program

- **Package:** cse_machine
- **Module**: parser_module.py (located in the cse_machine package)

This structure facilitates modularity and separation of concerns, allowing for a clear delineation of responsibilities among different components of the RPAL interpreter. Each component plays a crucial role in the overall parsing and execution process, contributing to the successful interpretation of RPAL programs.

Package Structure

The RPAL interpreter project is structured into several main packages, each responsible for different aspects of the parsing and execution process. Additionally, utility packages are included to provide common functionalities utilized throughout the interpreter.

Main Packages:

lexical_scanner: Package for lexical analysis functionality.

- *scanner.py*: Contains the logic for lexical analysis, responsible for scanning the RPAL source code and generating tokens.

- `token_scan(self,input_string):` This function takes an input string as an argument and scans it to generate a list of tokens. It returns a list of Token objects representing the tokens found in the input string. If any invalid characters or tokens are encountered, it raises a ScannerError.

screener: Package for screening functionality.
- *token_screener.py:* Implements screening functionality, filtering and categorizing tokens based on specific criteria.

- `screener(self, tokens):` The `screener` method takes a list of tokens as input and removes unwanted tokens from it. It then returns the filtered list of tokens. Tokens marked for deletion (type 'DELETE') or end-of-file tokens (type 'EOF') are excluded from the filtered list. Additionally, any identifier tokens (type 'ID') that match keywords are replaced with keyword tokens before being added to the filtered list.

parser: Package for parsing functionality.

- *parser_module.py:* Implements parsing functionality, constructing the Abstract Syntax Tree (AST) and Standard Tree (ST) from tokenized input.

- `parse(self,token_list)`: The `parse` method is the entry point of the recursive descent parser. It takes a list of tokens as input and attempts to parse them into an Abstract Syntax Tree (AST). If parsing is successful, it constructs the AST using the provided tokens; otherwise, it raises appropriate parsing errors.
- `build_tree(tree):` The `build_tree` method is used internally by the parser to construct nodes of the AST during parsing. It takes a token representing the type of node and the number of children nodes as input. It then creates a node with the given token and adds the specified number of children nodes to it.
- `readToken(self):` The `readToken` method is used internally by the parser to read the next token from the input list of tokens and update the `next_token` variable accordingly. It helps in advancing through the input token stream during parsing.
- `get_ast_tree(self)` The `get_ast_tree` method retrieves the root node of the constructed Abstract Syntax Tree (AST). It allows users to access the AST after parsing is complete, enabling further analysis or manipulation of the parsed code structure.
- `grammer_rule(self):`
  - Represents a grammar rule for parsing a specific language construct in the input code.
  - This function defines the parsing logic for a particular grammar rule, enabling the construction of an Abstract Syntax Tree (AST) during the parsing process. Each grammar rule corresponds to a specific language construct or expression in the input code.

- build_standard_tree.py:

- `build_standard_tree(tree)`: This function builds the standard tree from the input tree by applying various transformations.
- `_transform_let(tree)`: Transforms a let expression into a gamma expression.
- `_transform_tau(tree)`: Transforms a tau expression into a lambda expression.
- `_transform_and(tree)`: Transforms an and expression with equality into a comma expression.
- `_transform_function_form(tree)`: Transforms a function_form expression into a lambda expression.
- `_transform_lambda(tree)`: Transforms a lambda expression.
- `_transform_within(tree)`: Transforms a within expression.
- `_transform_uop(tree)`: Transforms a unary operator.
- `_transform_conditional(tree)`: Transforms a conditional expression.
- `_transform_where(tree)`: Transforms a where expression.
- `_transform_rec(tree)`: Transforms a rec expression.
- `_transform_op(tree)`: Transforms a binary operator.
- `_transform_at(tree)`: Transforms an at expression.

Each transformation function modifies the input tree according to specific rules, resulting in a standard tree representation.

cse_machine: Package for cse_machine functionality.

Here's a brief explanation of the contents of the `cse_machine` package:

- `apply_operation`: This sub-package contains modules for applying different types of operations in the CSE machine. It likely includes modules such as `apply_binary_operations.py` for applying binary operations and `apply_unary_operations.py` for applying unary operations.
- `data_structure`: This sub-package contains modules related to data structures used by the CSE machine. It likely includes modules such as `control_structure.py` for control structures, `environment.py` for environment-related functionalities, and `stack.py` for stack operations.
- `utils`: This sub-package contains utility modules that provide various helper functions for the CSE machine. It may include modules such as `util.py` for general-purpose utilities and `STlinearlizer.py` for linearizing syntax trees.

The `machine.py` module likely serves as the main entry point for the CSE machine functionality, possibly importing and orchestrating the functionality provided by the sub-packages and modules within the `cse_machine` package.

- machine.py:

    - `initialize()`: Initializes the CSEMachine with the necessary components.
    - `execute(st_tree)`: Executes the given ST.
    - Methods representing CSE rules (e.g., `CSErule1()`, `CSErule2()`): Implement the rules for executing different control structures.
    - Helper methods for applying binary and unary operations (`_apply_binary()`, `_apply_unary()`).
    - Methods for managing table data and generating output (`_add_table_data()`, `_generate_output()`).

Interpreter:

- *execution_engine.py:* Module containing the interpreter logic.

This package likely orchestrates the functionalities of lexical analysis, screening, and parsing, potentially importing and utilizing modules from lexical_scanner, screener, and parser to execute RPAL programs effectively.

- `interpret(file_name)`: This method interprets the content of a given file. It reads the content from the file, tokenizes it, filters the tokens, parses them into an Abstract Syntax Tree (AST), converts the AST into a Standard Tree (ST), evaluates the ST using the CSE machine, and generates raw and formatted output.
- `print_tokens()`: This method prints the tokens generated from the input file.
- `print_filtered_tokens()`: This method prints the filtered tokens after screening.
- `print_cse_table()`: This method prints the CSE table used by the CSE machine.
- `print_AST()`: This method prints the Abstract Syntax Tree (AST) of the program.
- `get_ast_list()`: This method retrieves the AST in list representation.
- `print_ST()`: This method prints the Syntax Tree (ST) of the program.
- `get_st_list()`: This method retrieves the ST in list representation.
- `get_raw_output()`: This method retrieves the raw output generated by the CSE machine.
- `get_output()`: This method retrieves the formatted output generated by the CSE machine.
- `print_output()`: This method prints the formatted output generated by the CSE machine.
- `clean_up()`: This method resets all attributes of the `Evaluator` class to their initial state.

Utility Packages:

- **table_routines:** Package for table routines functionality.
  - *char_map.py:* Contains logic for character mapping.
  - *fsa_table.py:* Implements Finite State Automaton table logic.
  - *accept_states.py:* Containing accept states logic.
  - keywords.py: Stores keyword sets used in the interpreter.

- **error_handling:** Package for error handling functionality.
  - *error_handler.py:* Provides functionality for error handling within the interpreter.

- **utils:** Package for utility functionalities.
  - *tokens.py:* Defines the token class used in lexical analysis.
  - *node.py:* Defines the node class used in constructing AST and ST.
  - *stack.py:* Implements the stack data structure.
  - *token_printer.py:* Offers token printing functionality for debugging.
  - *file_handler.py:* Contains file handling functions.
  - *tree_printer.py:* Provides functionality for printing the Tree.
  - *tree_list.py:* Module for listing tree elements.

In addition to these packages, main.py serves as the entry point for the application, handling command-line arguments and potentially orchestrating the execution of RPAL programs.

Overall, the project's structure promotes modularity, making it easier to maintain and extend the interpreter's functionalities.

Code Analysis and Explanation:

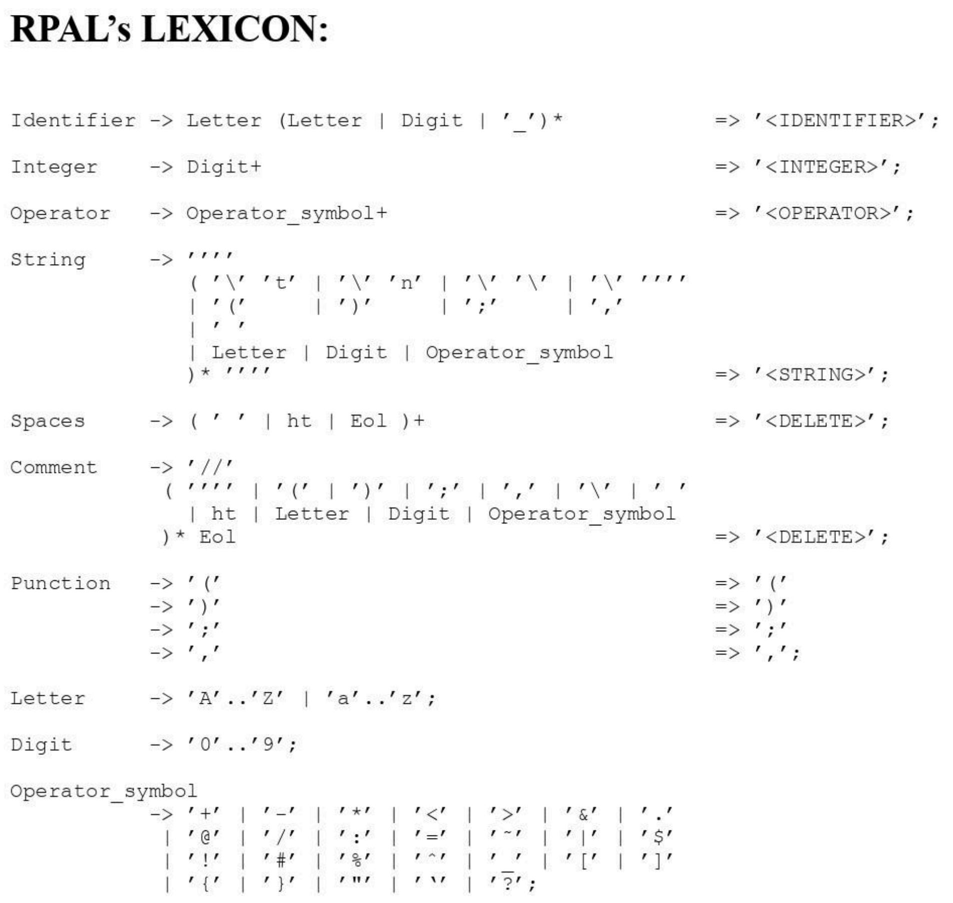
The Code Analysis and Explanation section provides an in-depth exploration of the RPAL interpreter's implementation, shedding light on the underlying logic, algorithms, and design choices. This walkthrough aims to elucidate the codebase's functionality and intricacies, offering valuable insights into each major component

Lexical Scanner :

In lexical analysis, the goal is to break down the input source code into meaningful tokens based on predefined rules.

Rules for the Lexical Analyzer:

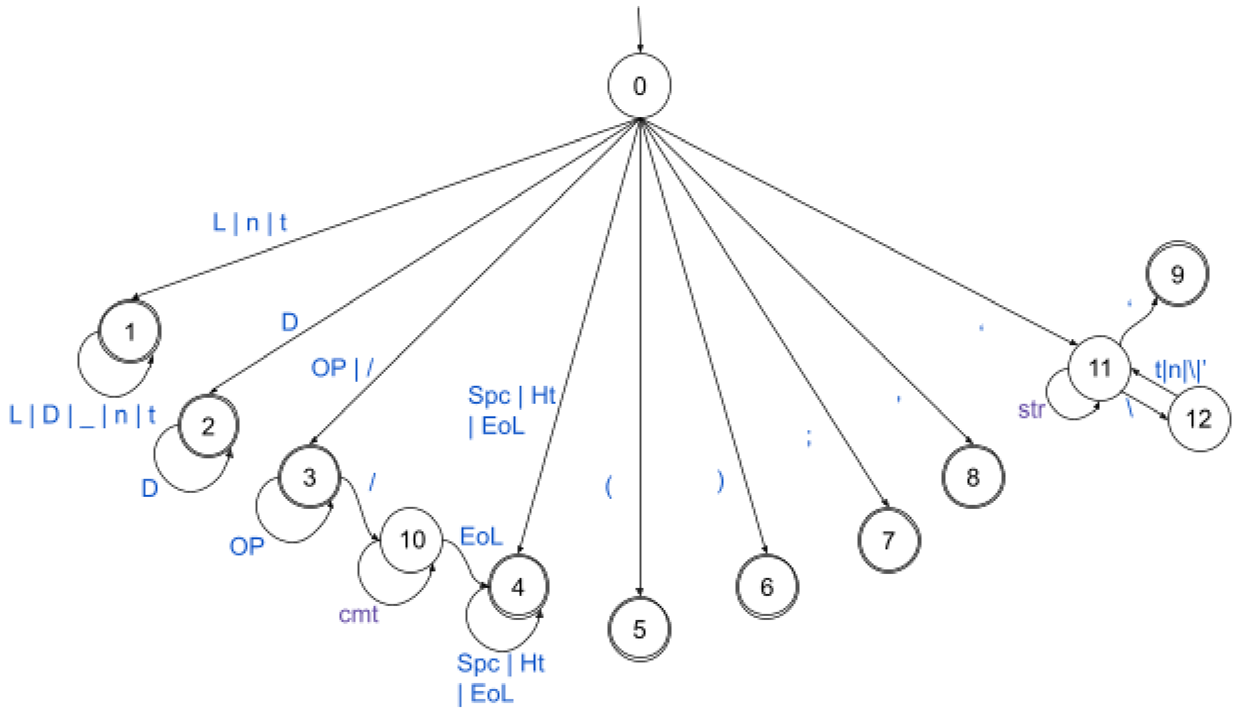
Lexical analysis involves tokenizing the input source code based on the following rules:



RPAL's LEXICON:

```
Identifier -> Letter (Letter | Digit | '_')*             => '<IDENTIFIER>';

Integer    -> Digit+                                     => '<INTEGER>';

Operator   -> Operator_symbol+                           => '<OPERATOR>';

String     -> ''''
              ( '\' 't' | '\' 'n' | '\' '\' | '\' ''''
              | '('     | ')'     | ';'     | ','
              | ' '
              | Letter | Digit | Operator_symbol
              )* ''''                                    => '<STRING>';

Spaces     -> ( ' ' | ht | Eol )+                        => '<DELETE>';

Comment    -> '//'
             ( '''' | '(' | ')' | ';' | ',' | '\' | ' '
              | ht | Letter | Digit | Operator_symbol
             )* Eol                                      => '<DELETE>';

Punction   -> '('                                        => '('
           -> ')'                                        => ')'
           -> ';'                                        => ';'
           -> ','                                        => ',';

Letter     -> 'A'..'Z' | 'a'..'z';

Digit      -> '0'..'9';

Operator_symbol
           -> '+' | '-' | '*' | '<' | '>' | '&' | '.'
            | '@' | '/' | ':' | '=' | '~' | '|' | '$'
            | '!' | '#' | '%' | '^' | '_' | '[' | ']'
            | '{' | '}' | '"' | '`' | '?';
```

The following figure depicts a finite-state automaton designed for a scanner based on the provided rules:



The finite state automaton (FSA) presented here serves as a mathematical abstraction to elucidate the functionality of the lexical scanner. Comprising states and transitions delineated by input characters, this FSA is pivotal in delineating the scanner's operation. By traversing through various states, it adeptly discerns and classifies tokens in conformity with the specified rules.

Accepted states :

Accepted states in the FSA correspond to different token types:

1. <ID>
2. <INT>
3. <OPERATOR>
4. <DELETE>
5. <(>
6. <)>
7. <;>
8. <,>
9. <STR>

Input definitions :

1. L - Letters (A-Z, a-z) except (n, t)
2. D - Digits (0-9)
3. OP - Operators (| , + , - , * , < , > , & , . , = , ˜ , , , $, ! , # , % , ^ , [,] , { , } , “ , ?)
4. Spc - Space
5. Ht - Tab
6. EoL - End of Line
7. Cmt - { L , D , OP , n , t , _ , ‘ , \ , (,) , ; , , , Spc , Ht , EoL , / }
8. Str - { (,),;, Spc, L, D, OP, n, t }

Transition table :

	L	n,t	D	_	OP	‘	\	(	)	;	,	Spc	Ht	EoL	/
0	1	1	2		3	11		5	6	7	8	4	4	4	3
1	1	1	1	1											
2			2												
3					3										10
4												4	4	4	
5															
6															
7															
8															
9															
10	10	10	10	10	10	10	10	10	10	10	10	10	10	4	10
11	11	11	11	11	11	9	12	11	11	11	11	11			11
12		11				11	11								

The transition table represents the behavior of the lexical scanner as it moves from one state to another based on the input characters encountered. It outlines the transitions between states and the corresponding actions taken for each character type.

This detailed explanation provides a comprehensive overview of the lexical scanner's rules, the finite state automaton, accepted states, input definitions, and the transition table, aiding in the understanding of how the scanner tokenizes the input source code.

Follow the code explaining about lexical analyzer :

lexical_analyzer/scanner.py

Description:

The `scanner.py` file contains the `Scanner` class, which is responsible for scanning the input string and returning a list of tokens. It utilizes a character map, a Finite State Automaton (FSA) table, and a set of accepted states to perform lexical analysis.

Usage:

To use the `Scanner` class, follow these steps:

1. Create an instance of the `Scanner` class.
2. Call the `token_scan()` method with the input string to be scanned.
3. The `token_scan()` method returns a list of `Token` objects representing the identified tokens in the input string.
4. If the input string contains any invalid characters or tokens, the `Scanner` class raises a `ScannerError`.

Attributes:

- error (method): A method to handle errors encountered during scanning.
- charMap (dict): A dictionary mapping characters to their corresponding indices for lookup.
- fsaTable (list of lists): A 2D list representing the Finite State Automaton table for transition states.
- acceptStates (set): A set containing accept states in the Finite State Automaton.
- status (bool): A flag indicating the current status of the scanner.

Methods:

- `__init__(self)`: Initializes the scanner and sets up necessary attributes.
- `token_scan(self, str)`: Scans the input string and returns a list of tokens.

`token_scan(self, str)` Method:

- Parameters:
  - `str` (str): The input string to be scanned.
- Returns:
  - List[Token]: A list of tokens generated from the input string.
- Raises:
  - ScannerError: If an invalid character or token is encountered during scanning.

Code Explanation:

- **Initialization**: In the constructor `__init__()`, the `Scanner` class initializes necessary attributes such as `error`, `charMap`, `fsaTable`, `acceptStates`, and `status`.
- **`token_scan()` Method:** This method is responsible for scanning the input string and identifying tokens. It iterates over each character in the input string, tracking the current state according to the FSA table. The method constructs tokens based on transitions between states and identifies accepted states. If an invalid character or token is encountered, it raises a `ScannerError`.
- **Error Handling**: The `token_scan()` method handles errors such as encountering invalid characters or tokens. It raises a `ScannerError` and provides appropriate error messages indicating the nature of the error and the line number where it occurred.
- **Return Value**: The `token_scan()` method returns a list of `Token` objects representing the identified tokens in the input string. If the input string is successfully scanned without encountering errors, the method sets the `status` flag to `True`.

Screener :

The `Screener` class is responsible for filtering unwanted tokens from a given list of tokens. Let's break down the code and provide explanations for each part:

Usage:

1. To utilize the `Screener` class, follow these steps:
2. Instantiate an object of the `Screener` class.
3. Call the `screener()` method with the list of tokens to be screened.
4. The `screener()` method returns a filtered list of tokens, removing any unwanted tokens according to the defined rules.
5. If the input list contains any tokens marked for deletion or identifiers matching keywords, they are filtered out.

Import Statements:

```
from table_routines.keywords import Keywords

from utils. tokens import Token
```

This module imports the `Keywords` class from the `table_routines. keywords` module and the `Token` class from the `utils.tokens` module. These imports are necessary for accessing the keywords set and creating token objects.

Attributes:

- Initializes the `keywords` attribute by retrieving the set of keywords from the `Keywords` class.

Methods:

def screener(self, tokens):

- Defines the `screener` method, which takes a list of tokens as input and returns a filtered list of tokens.

Token Screening:

```
for the token in tokens:
  if token.get_type() == 'DELETE':
    continue
  elif token.get_type() == 'ID' and token.get_value() in self.keywords:
    filtered_tokens.append(Token(token.get_value(), "KEYWORD"))
  else:
    filtered_tokens.append(token)
```

Iterates through each token in the input list.

- Check if the token type is `'DELETE'`. If so, it skips this token.
- If the token type is `'ID'` (identifier) and its value is found in the `keywords` set, it replaces the token with a new token of type `'KEYWORD'` with the same value.
- Otherwise, it adds the token to the `filtered_tokens` list.

EOF Token:

```
filtered_tokens.append(Token("EOF", "EOF"))
```

Appends an end-of-file (EOF) token to the end of the filtered list of tokens.

Return:

```
return filtered_tokens
```

Returns the filtered list of tokens.

Overall, the `Screener` class ensures that unwanted tokens, such as those marked for deletion or identifiers that match keywords, are removed from the input list of tokens, resulting in a filtered list of tokens.

Parser :

The `parser_module.py` contains the `Parser` class responsible for parsing the input code and constructing an Abstract Syntax Tree (AST) using a recursive descent parsing technique. Let's break down the code and understand its components:

Description:

The `Parser` class analyzes the structure of the input RPAL code based on predefined grammar rules. It identifies expressions, definitions, variables, and other language constructs, building a hierarchical representation of the code in the form of an AST.

We leverage Recursive Descent Parsing for the parsing process.

Recursive Descent Parsing:

The parser utilizes a recursive descent parsing technique, where each non-terminal symbol in the grammar corresponds to a parsing function. These functions recursively parse the input tokens, building the AST along the way.

Usage:

To use the `Parser` class:

1. Create an instance of the class.
2. Call the `parse()` method with a list of tokens representing the code to be parsed.
3. The parser will analyze the code and construct an AST accordingly.
4. Check the `status` attribute of the parser instance to determine if parsing was successful.
5. Access the constructed AST using the `stack` attribute of the parser instance.

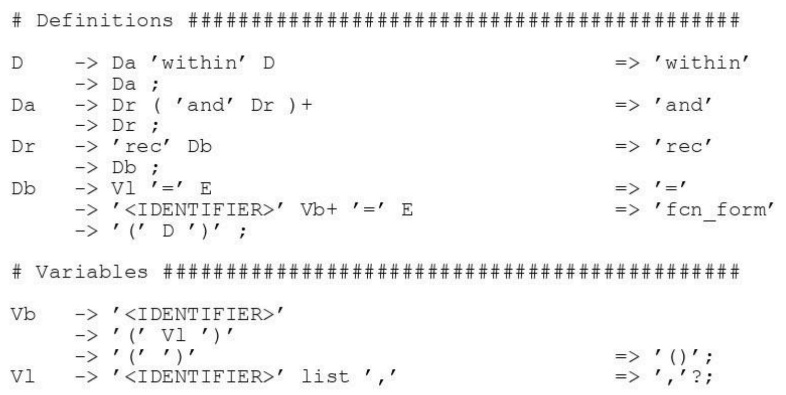
Grammar Rules:

The grammar rules provided dictate the syntax and structure of valid RPAL programs. The parser adheres to these rules during the parsing process to ensure that the input code is correctly analyzed and interpreted.

The parser in the RPAL Interpreter has been implemented based on the provided grammar rules using a Recursive Descent approach.

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 ','                => ','?;
```

Code Structure:

The parser module consists of:

- The `Parser` class is responsible for parsing.
- Various parsing functions correspond to different grammar rules.
- Helper functions to read tokens and build nodes in the AST.

Execution Flow:

- The `parse()` method serves as the entry point for parsing. It initiates parsing and handles the overall execution flow.
- Parsing functions are called recursively based on the grammar rules.
- Tokens are consumed from the input token list using the `readToken()` function.
- AST nodes are built using the `build_tree()` function, incorporating parsed information.
- Error handling ensures proper handling of parsing errors and incomplete parsing.

Error Handling:

The parser includes error handling mechanisms to deal with various parsing errors, such as unexpected tokens or incomplete parsing. Error messages provide information about the nature of the error encountered during parsing.

AST Construction:

As parsing progresses, AST nodes are constructed and organized on a stack data structure. Each parsing function contributes to building the AST by pushing nodes onto the stack, ultimately forming a complete representation of the input code.

Completion Status:

Upon successful parsing, the `status` attribute of the parser instance is set to True, indicating that parsing is completed without errors. If parsing fails or encounters errors, appropriate error messages are generated.

Standardizing RPAL AST's

Continuing with the parser, our next step involves standardizing the RPAL AST tree. RPAL programs are inherently a mix of lambda calculus and syntactic sugar, designed to be programmer-friendly. However, to ensure seamless execution within a Control Structure Environment (CSE) machine, we undertake a process known as 'desugaring' the tree. This involves refining internal nodes to exclusively feature 'gamma' and 'lambda' nodes, adhering strictly to predefined rules. Essentially, we're simplifying the tree to its core structure to align it more closely with the execution expectations of the CSE environment.

Here are the rules guiding our transformation process from the Abstract Syntax Tree (AST) to the Symbol Table (ST):

```
     let      =>      gamma    |      where   =>      gamma
     / \              /   \    |       / \            /  \
    =   P         lambda E     |      P   =       lambda  E
   / \              / \        |         / \        / \
  X   E            X   P       |        X   E      X   P
                               |
     tau      =>    ++gamma    |       ->     =>      gamma
      |               /  \     |      / | \           /  \
     E++           gamma  E    |     B  T  E      gamma nil
                    / \        |                   / \
                 aug  .nil     |              gamma    lambda
                               |               / \      / \
                               |          gamma lambda ()  E
                               |           / \   / \
                               |       Cond   B ()  T
                               |
     not      =>      gamma    |      neg     =>      gamma
      |               /  \     |       |              /  \
      E             not   E    |       E            neg   E
                               |
    within    =>        =      |      rec     =>        =
    /   \              / \     |       |               / \
   =     =           X2  gamma |       =              X  gamma
  / \   / \              / \   |      / \               / \
 X1 E1 X2 E2        lambda  E1 |     X   E          Ystar  lambda
                     / \       |                            / \
                    X1  E2     |                           X   E
                               |
    fcn_form  =>       =       |     lambda   =>      lambda
    /  _|  \          / \      |      / \             / \
   P   V+   E        P  +lambda|     ,   E         Temp  ++gamma
                          / \  |     |                    / \
                         V  .E |    X++i             lambda  gamma
                               |                     / \      / \
                               |                   X.i .E Temp   \
                               |                               <INTEGER:1>
                               |
     lambda   =>    ++lambda   |      Op      =>      gamma
     /  \            /     \   |     / \             /    \
   V++   E          V      .E  |    E1  E2        gamma   E2
                               |                   / \
                               |                  Op  E1
                               |
     and      =>       =       |      @       =>      gamma
      |               / \      |     / | \            /  \
     =++             ,   tau   |   E1  N  E2      gamma  E2
     / \             |    |    |                   / \
    X   E           X++  E++   |                  N   E1
                               |
     Uop      =>     gamma     |   Op in [aug,or,&,+,-,/,**,gr ...]
      |              /   \     |
      E            Uop    E    |   Uop in [not, neg]
```

Implementation:

The transformation process is handled by the `build_standard_tree.py` module, particularly within the `StandardTree` class.

Here's how we can proceed:

- Define a method in the `StandardTree` class to handle this transformation.
- Implement the transformation logic in the method.
- Call the method from the `_apply_transformations` method based on the corresponding condition in the grammar rules.

Description:

Within the `build_standard_tree.py` module resides the `StandardTree` class, purpose-built to convert an Abstract Syntax Tree (AST) into a standardized tree format. This transformation is crucial for seamless execution within the Control Structure Environment (CSE) machine. The class implements a series of predefined rules and transformations outlined in the "semantics.pdf" documentation. Each rule corresponds to specific patterns found within the structure of the input AST. By adhering to these rules, the class ensures that the resulting standardized tree is optimized for CSE machine execution.

Usage:

To utilize the `StandardTree` class for transforming an AST into a standard tree:

```
from parser.build_standard_tree import StandardTree
# Instantiate StandardTree
standard_tree_builder = StandardTree()

# Obtain the AST tree (let's assume it's stored in variable 'ast_tree')
# Perform standardization
standard_tree = standard_tree_builder.build_standard_tree(ast_tree)
```

The `build_standard_tree` method meticulously applies the defined transformations to the input tree, resulting in a standard tree that aligns with the requirements of the CSE environment. By encapsulating the transformation logic within the `StandardTree` class, this module offers a comprehensive solution for standardizing ASTs, essential for efficient execution within the CSE environment.

Now, we need to call this method from the `_apply_transformations` method when appropriate:

By adding this method and calling it from `_apply_transformations`, we can handle variable assignment expressions according to the grammar rules.

CSE Machine

To make RPAL work using the CSE (Control-Stack-Environment) machine, we have to follow some steps. First, we look at the code and make a structure called the control structure (CS). We do this by going through the code in a specific order and including things like names, operators (γ), and lambda expressions (λ). This helps us know what to do next when we run the program.

At the same time, we set up something called an environment (E). In the beginning, it's based on something called the Primitive Environment (PE). But as we run the program, it changes to keep track of things like names and what they mean. This helps us find things quickly when we need them while running the program.

When we see lambda expressions in the code, we change them into something called lambda closures. These closures include information about the environment, the code inside the lambda, and the variable it's working with. This helps us use the lambda functions correctly while running the program.

During the process of running the program, we follow certain rules to make things work smoothly and efficiently. These rules help us handle things like recursion (when a function calls itself) and make sure the program runs correctly. In total, 13 rules guide how the CSE machine interprets RPAL programs, ensuring they work as expected.

These rules are utilized :

CSE Machine Rules:

	CONTROL	STACK	ENV
Initial State	$e_0\ \delta_0$	e_0	e_0 = PE
CSE Rule 1 (stack a name)	.... Name	.... Ob	Ob=Lookup(Name,e_c) e_c:current environment
CSE Rule 2 (stack λ)	.... λ_k^x	.... ${}^c\lambda_k^x$	 e_c:current environment
CSE Rule 3 (apply rator)	.... γ	Rator Rand .... Result	 Result=Apply[Rator,Rand]
CSE Rule 4 (apply λ)	.... γ .... $e_n\ \delta_k$	${}^c\lambda_k^x$ Rand .... e_n	e_n = [Rand/x]e_c
CSE Rule 5 (exit env.)	.... e_n	value e_n .... value	

Optimizations for the CSE Machine.

CSE Rules 6 and 7: Unary and Binary Operators.

	CONTROL	STACK	ENV
CSE Rule 6 (binop)	.... binop	Rand Rand .... Result	 Result=Apply[binop,Rand,Rand]
CSE Rule 7 (unop)	.... unop	Rand .... Result	 Result=Apply[unop,Rand]

CSE Rule 8: Conditional.

	CONTROL	STACK	ENV
CSE Rule 8 (Conditional)	.... δ_{then} δ_{else} β	true	
	.... δ_{then}	....	
	.... δ_{then} δ_{else} β	false	
	.... δ_{else}	....	

CSE Rules 9 and 10: Tuples.

	CONTROL	STACK	ENV
CSE Rule 9 (tuple formation)	.... τ_n	V_1 ... V_n	
	....	$(V_1,...,V_n)$	
CSE Rule 10 (tuple selection)	.... γ	$(V_1,...,V_n)$ I	
	....	V_I	

CSE Rules:

	CONTROL	STACK	ENV
CSE Rule 12	.... γ	Y $^{c}\lambda_{i}^{v}$	
(applying Y)	....	$^{c}\eta_{i}^{v}$	
CSE Rule 13	.... γ	$^{c}\eta_{i}^{v}$ R	
(applying f.p.)	.... γ γ	$^{c}\lambda_{i}^{v}$ $^{c}\eta_{i}^{v}$ R	

CSE Evaluation:

RULE	CONTROL	STACK	ENV
2	e_0 γ λ_1^f γ Y λ_2^f	e_0	e_0=PE
1	e_0 γ λ_1^f γ Y	$^0\lambda_2^f$ e_0	
12	e_0 γ λ_1^f γ	Y $^0\lambda_2^f$ e_0	
2	e_0 γ λ_1^f	$^0\eta_2^f$ e_0	
4	e_0 γ	$^0\lambda_1^f$ $^0\eta_2^f$ e_0	
1	e_0 e_1 γ f 3	e_1 e_0	e_1=[$^0\eta_2^f$/f]e_0
1	e_0 e_1 γ f	3 e_1 e_0	
13	e_0 e_1 γ	$^0\eta_2^f$ 3 e_1 e_0	

CSE Rule 11: n-ary functions.

	CONTROL	STACK	ENV
CSE Rule 11	.... γ	${}^{c}\lambda_k^{V_1,\ldots,V_n}$ Rand	e_m=[Rand $1/V_1$]...
(n-ary function)	.... e_m δ_k	e_m	[Rand n/V_n]e_c

Example:

Applicative expression	Control Structures
(λ(x,y).x+y) (5,6)	$\delta_0 = \gamma\ \lambda_1^{x,y}\ \tau_2$ 5 6
	δ_1 = + x y

RULE	CONTROL	STACK	ENV
1,1	$e_0\ \gamma\ \lambda_1^{x,y}\ \tau_2$ 5 6	e_0	e_0=PE
9	$e_0\ \gamma\ \lambda_1^{x,y}\ \tau_2$	5 6 e_0	
2	$e_0\ \gamma\ \lambda_1^{x,y}$	(5,6) e_0	
11	$e_0\ \gamma$	${}^{0}\lambda_1^{x,y}$ (5,6) e_0	
1,1	e_0 e_1 + x y	e_1 e_0	e_1=[5/x][6/y]e_0
6	e_0 e_1 +	5 6 e_1 e_0	
5,5	e_0 e_1	11 e_1 e_0	
		11	

Code Structure:

In the `cse_machine` package, the code architecture is structured into three main sub-packages: `data_structures`, `utils`, and `apply_operations`, each serving distinct roles in facilitating the operational semantics of the RPAL interpreter.

- **Data Structures** (`data_structures`):

  This package contains the fundamental data structures crucial for the CSE machine's operation. Specifically, it defines classes for `ControlStructure`, `Stack`, and `Environment`, representing the control, stack, and environment components of the CSE machine, respectively. These classes encapsulate the functionality required for manipulating and managing these data structures during the evaluation process.

- **Utilities** (`utils`):

  The `utils` package provides auxiliary functions and utilities that support various operations within the CSE machine. The `util.py` module likely contains helper functions used across the interpreter, while `STLinearizer.py` is responsible for flattening RPAL programs into a standardized tree (ST) format. This flattening process is essential for subsequent interpretation and execution.

- **Apply Operations** (`apply_operations`):

  This package contains modules for defining and applying binary and unary operations within the RPAL interpreter. Specifically, `apply_bi.py` defines functions for binary operations, while `apply_un.py` defines functions for unary operations. These modules play a crucial role in executing the operations specified in RPAL programs during evaluation.

- **Machine Initialization and Rule Definitions** (`machine.py`):

  The `machine.py` module serves as the central component responsible for initializing the CSE machine and defining the 13 rules governing its operation. This module likely contains functions for setting up the initial state of the machine, as well as implementing the logic for applying each of the 13 rules during program evaluation. These rules are essential for accurately interpreting RPAL programs and ensuring the proper functioning of the CSE machine.

main.py:

- Description:
  - `main.py` serves as the primary entry point for interpreting RPAL programs.
  - It provides functionality to interpret RPAL code, print Abstract Syntax Trees (ASTs), tokens, and filtered tokens, as well as execute the original RPAL interpreter on a file and print the AST.
  -
- Usage:
  - Execute `main.py` with appropriate command-line arguments to trigger the interpretation of RPAL programs and print ASTs if specified.
  - The usage format is:

```
Python main.py [-ast] [-t] [-ft] [-st] [-r] [-rast] file_name
```

- Arguments:
  - `file_name`: The name of the RPAL file to interpret.

- Optional Switches:
  - `-ast`: Print the Abstract Syntax Tree (AST) for the given RPAL program.
  - `-t`: Print the tokens generated by the lexical analyzer for the given RPAL program.
  - `-ft`: Print the filtered tokens generated by the screener for the given RPAL program.
  - `-st`: Print the Standard Tree for the given RPAL program. (Not yet implemented)
  - `-r`: Execute the original RPAL interpreter on the given RPAL program file (file should be in `rpal_tests/rpal_source`).
  - `-rast`: Execute the original RPAL interpreter on the given RPAL program file and print the AST. (file should be in `rpal_tests/rpal_source`)
  - Examples:

    ```
    Python main.py -ast file_name
    Python main.py -ast file_name
    ```

- Handling Command-Line Arguments:
  - The script checks the number of command-line arguments to ensure it has at least the required filename argument.
  - It extracts the filename from the command-line arguments.
- Evaluator Initialization:
  - The script initializes an instance of the `Evaluator` class to interpret the RPAL code.
  - The `Evaluator` class contains methods for interpreting RPAL code and printing ASTs, tokens, and filtered tokens.
- Handling Switches:
  - Each optional switch is handled by a specific function:
    - `handle_ast_option()`: Prints the AST.
    - `handle_standard_tree_option()`: Prints the Standard Tree.
    - `handle_tokens_option()`: Prints the tokens.
    - `handle_filtered_tokens_option()`: Prints the filtered tokens.
    - `handle_original_rpal_eval()`: Executes the original RPAL interpreter.
    - `handle_original_rpal_ast()`: Generates and prints the AST using the original RPAL interpreter.
    - `handle_original_rpal_st()`: Generates and prints the Standard Tree using the original RPAL interpreter.
  - The script checks the operating system before executing the original RPAL interpreter or generating AST/ST to ensure compatibility.
- Error Handling:
  - Basic error handling is implemented to handle cases where the required files are not found or errors occur during interpretation.
  - Error messages are displayed to inform the user about the issue.

Data Structures:

utils/node.py:

- Introduction: Defines the node data structure used in the interpreter.
- Functionality: Represents nodes within various data structures such as trees and lists.
- Usage: Used to construct data structures like ASTs and STs during parsing and execution.

Purpose:

The `Node` class represents a node in a binary tree data structure. It provides functionalities to store data and manage child nodes.

Components:

- `__init__(self, data)`: Constructor method to initialize a new node with the given data. It also initializes an empty list to store child nodes.
- `add_child(self, child)`: Method to add a child node to the current node. It inserts the child node at index 0 to ensure the most recently added child appears first.
- `remove_child(self, child)`: Method to remove a child node from the current node. It checks if the provided child exists in the children's list and removes it if found.
- `__repr__(self)`: Method to return a string representation of the node. It displays the node's data and the data of its children.

Use Cases:

- **Binary Tree Construction**: The `Node` class is used to construct binary trees where each node can have at most two children.
- **Tree Traversal Algorithms**: Algorithms like depth-first search (DFS) or breadth-first search (BFS) can be implemented using the `Node` class to traverse the tree structure.
- **Expression Trees**: In compilers or interpreters, the `Node` class can represent nodes in an expression tree where operators are stored in internal nodes and operands are stored in leaf nodes.
- **Syntax Trees**: During the parsing of programming languages, the `Node` class can represent nodes in a syntax tree, capturing the hierarchical structure of the code.

Project Integration:

- In an interpreter or compiler project, the `Node` class can be used to represent nodes in various tree structures such as abstract syntax trees (ASTs), parse trees, or expression trees.
- It can be integrated into the parsing module to build the parse tree or syntax tree during the parsing process.
- During the interpretation or compilation phase, the `Node` class can be utilized to traverse the constructed tree and perform necessary operations based on the structure and data stored in the nodes.

utils/stack.py:

- Introduction: Implements the stack data structure for use within the interpreter.
- Functionality: Provides functionalities for stack operations such as push, pop, and peek.
- Usage: Utilized various interpreter components for managing program state and execution context.

Purpose:

The `Stack` class implements the stack data structure, which follows the Last-In, First-Out (LIFO) principle. It provides operations to push elements onto the stack, pop elements off the stack, peek at the top element without removing it, and check if the stack is empty.

Components:

- `__init__(self)`: Constructor method to initialize an empty stack. It initializes the stack with an empty list as the underlying data structure.
- `is_empty(self)`: Method to check if the stack is empty. It returns `True` if the stack is empty, and `False` otherwise.
- `push(self, item)`: Method to add an item to the top of the stack. It appends the item to the end of the list, effectively adding it to the top of the stack.
- `pop(self)`: Method to remove and return the item from the top of the stack. It removes the last element from the list and returns it.
- `peek(self)`: Method to return the item at the top of the stack without removing it. It accesses the last element of the list.
- `size(self)`: Method to return the number of items in the stack. It returns the length of the list representing the stack.

Use Cases:

- **Expression Evaluation:** Stacks are commonly used in evaluating expressions, especially infix to postfix conversion and postfix expression evaluation.
- **Function Call Stack:** Stacks are used by programming languages to manage function calls and store local variables and return addresses.
- **Backtracking Algorithms:** Stacks can be used in backtracking algorithms such as depth-first search (DFS) to store the path taken so far.
- **Syntax Parsing:** In compilers and interpreters, stacks are used in syntax analysis for parsing expressions and statements.

Project Integration:

- The `Stack` class can be integrated into various parts of a project where stack-like behavior is required, such as in-depth first search algorithms, expression evaluation, or function call management.
- In a compiler or interpreter project, the `Stack` class can be utilized in the parsing phase to implement expression parsing or syntax analysis algorithms.
- It can also be used in implementing algorithms for evaluating arithmetic expressions or converting between different representations of expressions.

utils/tokens.py:

- Introduction: Defines the token class used to represent lexical tokens.
- Functionality: Contains attributes and methods to manage token properties and behavior.
- Usage: Tokens are generated during lexical analysis and utilized throughout the parsing and execution phases of interpretation.

Purpose:

The `Token` class represents a token in a program. A token is a fundamental element of a program that carries a specific meaning and can be interpreted by a computer. Each token typically consists of a type and a value, where the type defines its meaning and the value provides additional information.

Components:

`__init__(self, value: str, type_: str)`: Constructor method to initialize a new `Token` instance with the provided value and type.

Attributes:

- `type`: Represents the type of the token.
- `value`: Represents the value of the token.

Getter and Setter Methods:

- `get_type(self) -> str`: Returns the type of the token.
- `get_value(self) -> str`: Returns the value of the token.
- `set_type(self, type_: str)`: Sets the type of the token.
- `set_value(self, value: str)`: Sets the value of the token.

String Representation:

- `__str__(self)`: Returns a string representation of the token in the format `<type: value>`.
- `__repr__(self)`: Returns an unambiguous string representation of the token in the format `<type: value>`.

Equality and Hashing:

- `__eq__(self, other)`: Defines equality comparison between two tokens based on their type and value.
- `__ne__(self, other)`: Defines inequality comparison between two tokens.
- `__hash__(self)`: Returns a hash value for the token based on its type and value.

Use Cases:

- Lexical Analysis: Tokens are generated during lexical analysis, where the input program is scanned and broken down into individual tokens for further processing.
- Parsing: Tokens are utilized during parsing to recognize syntactic elements of the language and construct meaningful structures such as abstract syntax trees (ASTs) or parse trees.
- Interpretation and Compilation: Tokens play a crucial role in the interpretation and compilation process, where they are used to understand the semantics of the program and execute or translate it accordingly.

Project Integration:

- The `Token` class can be integrated into various components of a compiler, interpreter, or language processing tool where tokenization is required.
- It serves as a fundamental building block for implementing lexical analysis, parsing, and semantic analysis stages of language processing pipelines.
- In a compiler project, the `Token` class can be utilized to represent tokens generated by the lexer and passed to the parser for syntactic analysis.

Error Handling

error_handling/error_handler.py:
- **Introduction:** The error_handler module facilitates robust error management within the interpreter.
- **Functionality:** It introduces the ErrorHandler class, which manages various error types encountered during interpretation.
- **Usage:** By implementing the ErrorHandler class, the interpreter ensures graceful handling of errors, enhancing user experience and reliability.

cse_error_handling/error_handler.py:
- **Introduction:** This module is pivotal for error management within the CSE (Control Structure Environment) interpreter.
- **Functionality:** It defines the CseErrorHandler class, offering static methods to handle diverse error scenarios encountered during interpretation.
- **Usage:** Incorporate the CseErrorHandler class to efficiently manage errors throughout interpreter execution, ensuring smooth handling and improved user interaction.

Additional Functions:

utils/file_handler.py:
- **Introduction:** This module handles file input and output operations.
- **Functionality:** Provides functions for reading from and writing to files.
- **Usage:** Used to manage RPAL program files, ensuring seamless interaction between the interpreter and external files.

utils/token_printer.py:
- **Introduction:** Responsible for printing tokens, and aiding in debugging procedures.
- **Functionality:** Contains functions to print tokens extracted during lexical analysis.
- **Usage:** Helps developers debug the lexical analysis process by visualizing token streams.

utils/tree_printer.py:
- **Introduction:** This module contains a function to print the tree with appropriate indentation.
- **Usage:** The `print_tree()` function is provided in this module to print the AST with appropriate indentation.

Tables:

table_routines/accept_states.py:
- **Introduction:** Manages logic related to accepting states within the interpreter.
- **Functionality:** Defines accept states for various parsing processes.
- **Usage:** Used during parsing to determine valid states and transitions.

table_routines/keywords.py:
- **Introduction:** Contains a set of RPAL keywords used in lexical analysis and parsing.
- **Functionality:** Defines a collection of keywords recognized by the interpreter.
- **Usage:** Employed during lexical analysis and parsing to identify language-specific keywords.

table_routines/fsa_table.py:
- **Introduction:** Implements the Finite State Automaton (FSA) table logic for lexical analysis. (FSA)
- **Functionality:** Defines the transition rules for the FSA used in tokenization.
- **Usage:** Utilized during lexical analysis to determine token boundaries and classifications.

table_routines/char_map.py:
- **Introduction:** Contains character mapping logic for lexical analysis.
- **Functionality:** Defines mappings between characters and their corresponding token types.
- **Usage:** Used during lexical analysis to identify and classify characters within the input stream.

This detailed breakdown offers a comprehensive understanding of each component's purpose, functionality, and usage within the RPAL interpreter, facilitating further exploration and development.

Testing and Makefile

Testing

About Testing

To conduct testing, navigate to the 'rpal_tests' directory within the project repository. Inside this directory, you'll find 56 RPAL programs intended for testing, located specifically in the 'rpal_tests/rpal_sources' subdirectory. These tests are categorized into three groups: abstract tree evaluation (test_ast), standard tree evaluation (test_st), and default evaluation (test). In total, you'll execute 56 tests for each category, amounting to 168 tests overall. During testing, you'll compare the output of our RPAL interpreter against the output generated by 'rpal.exe,' created by Steven V. Walstra.

After manually executing the tests, take a screenshot displaying the passed test cases. Save this screenshot in the doc/test_case_pass directory.

The testing infrastructure in the `rpal_tests` directory employs pytest's parameterized testing extensively to generate and execute test cases efficiently. Here's a more detailed overview:

- **Parameterized Testing:** Pytest allows parameterized testing using the `@pytest.mark.parametrize` decorator. This feature enables generating multiple test cases based on different input parameters. In the context of the RPAL interpreter, parameterized testing is used to generate tests for different RPAL source code files and their corresponding outputs.
- **Test Generation Modules:** The `test_generate_tests.py`, `test_generate_ast_tests.py`, and `test_generate_st_tests.py` modules utilize pytest's parameterized testing to dynamically generate test cases for the interpreter. These modules programmatically iterate over the RPAL source code files in the `rpal_sources` directory and define test cases based on various criteria, such as AST output, ST output, etc.
- **Test Execution:** Once the test cases are generated, pytest executes them in parallel, efficiently utilizing system resources. Each test case runs independently, allowing for fast and parallelized testing of the interpreter's functionality.
- **Assertion and Validation:** The `assert_program.py` module contains functions responsible for validating the correctness of the interpreter's output against the expected output. These assertion functions compare the actual output generated by the interpreter (`rpal.exe`) with the expected output defined in `output.py`, `output_ast.py`, and `output_st.py`.
- **Continuous Integration:** The testing infrastructure is often integrated with continuous integration (CI) pipelines, such as GitHub Actions or Jenkins. This integration automates the execution of tests whenever changes are made to the interpreter codebase, ensuring that any modifications do not introduce regressions or bugs.
- **Comprehensive Coverage:** By using parameterized testing, the testing suite covers a wide range of scenarios and edge cases, enhancing the overall robustness and reliability of the interpreter. Additionally, the use of parameterization allows for easy scalability, enabling the addition of new test cases with minimal effort.

Overall, pytest's parameterized testing, coupled with the modular test generation approach, enables efficient and comprehensive testing of the RPAL interpreter, ensuring its correctness and reliability across different input scenarios.

In the `rpal_tests` directory, the testing infrastructure for the RPAL interpreter is organized. It consists of various modules and directories dedicated to different aspects of testing:

- rpal_sources: This directory contains the RPAL source code files used for testing.
- test_generate_tests.py: This module is responsible for generating tests for the interpreter.
- test_generate_ast_tests.py: This module generates tests for the Abstract Syntax Tree (AST) using pytest for the RPAL source code files located in the `rpal_sources` directory.
- test_generate_st_tests.py: Similarly, this module generates tests for the Standard Tree (ST) using pytest for the RPAL source code files.
- test_generate_tests_with_rpal_exe.py: This module generates tests for the final output produced by the RPAL interpreter (`rpal.exe`) using the output generated by the actual `rpal.exe` program.
- test_generate_ast_tests_with_rpal_exe.py: This module generates tests for the AST output produced by the RPAL interpreter.
- test_generate_st_tests_with_rpal_exe.py: Similarly, this module generates tests for the ST output produced by the RPAL interpreter.
- assert_program.py: This module contains functions for checking the correctness of the tests.
- program_name_list.py: It provides a list of program names in the `rpal_sources` directory.
- output.py: Contains a list of outputs generated by the RPAL interpreter (`rpal.exe`).
- output_ast.py: Contains a list of AST outputs generated by the RPAL interpreter.
- output_st.py: Contains a list of ST outputs generated by the RPAL interpreter.
- rpal.exe: The RPAL interpreter executable.
- cygwin1.dll: Cygwin DLL required for execution if applicable.

This testing setup ensures comprehensive testing of the interpreter functionality. With 56 test cases in total, including 3 sets of tests for AST, ST, and final output, the interpreter is rigorously tested to ensure its correctness and reliability.

Makefile

The provided Makefile offers a comprehensive set of targets for managing the RPAL interpreter project, including dependency installation, running the interpreter, executing tests, and cleaning up generated files. Here's a detailed report on its functionality:

Targets and Usage:

The Makefile includes various targets, each serving a specific purpose:

`install` install project dependencies specified in the `requirements.txt` file.
`run`: Executes the RPAL interpreter with the provided test file.
`test`: Executes all test cases related to the interpreter.
`test_ast`: Executes test cases specifically related to the Abstract Syntax Tree (AST).
`test_st`: Executes test cases specifically related to the Standardized Syntax Tree (ST).
`test_all`: Executes all test cases for both AST and ST.
`clean`: Cleans up generated files and directories.
`all`: Installs dependencies, runs the interpreter, executes tests, and cleans up.

Platform Handling:

- The Makefile includes platform-specific conditionals to ensure compatibility across different operating systems (Windows and Linux).
- It dynamically sets the shell for execution based on the platform.

Dependency Management:

- The `install` target checks for the presence of Python and pip, ensuring they are installed before proceeding with dependency installation.
- It installs project dependencies listed in the `requirements.txt` file using pip.

Test Execution:

- The Makefile allows running tests using different targets (`test`, `test_ast`, `test_st`, `test_all`).
- It leverages pytest for test execution and parametrization, enabling efficient and organized testing.

Cleanup:

- The `clean` target removes generated Python bytecode files (`*.pyc`), cached directories (`__pycache__`), and pytest cache files (`pytest_cache`).

Error Handling:

- The Makefile includes error-handling mechanisms to ensure the smooth execution of commands and targets. For instance, it checks for the presence of the `requirements.txt` file before proceeding with dependency installation.

Reporting:

- The Makefile provides informative messages during execution, including status updates, error notifications, and cleanup acknowledgments.

Overall, the Makefile offers a robust and organized workflow for managing the RPAL interpreter project, streamlining dependency management, test execution, and cleanup tasks. It enhances project maintainability and facilitates collaboration by providing a standardized set of commands for developers to interact with the project.

Directory Structure(**Clear Repository Blueprint:** Link Here)

```
RPAL-Interpreter/
├── myrpal.py                          # Main entry point of the RPAL interpreter application
|
├── src/                          # Source code directory
|   ├── lexical_analyzer/            # Package for lexical analysis functionality
|   |   ├── scanner.py               # Module containing logic for the lexical scanner
|   |   └── __init__.py              # Marks the directory as a Python package
|   |
|   ├── screener/                    # Package for screening functionality
|   |   ├── screener.py              # Module containing logic for screening
|   |   └── __init__.py              # Marks the directory as a Python package
|   |
|   ├── parser/                      # Package for parsing functionality
|   |   ├── build_standard_tree.py     # Module for converting Abstract Syntax Tree (AST) to standard tree
|   |   ├── parser_module.py          # Module containing parser logic (converts tokens to AST)
|   |   └── __init__.py              # Marks the directory as a Python package
|   |
|   ├── cse_machine/                 # Package for CSE (Control Stack Environment) machine functionality
|   |   ├── apply_operations/         # Package for applying operations in the CSE machine
|   |   |   ├── apply_bi.py           # Module for applying binary operations
|   |   |   ├── apply_un.py           # Module for applying unary operations
|   |   |   └── __init__.py           # Marks the directory as a Python package
|   |   ├── data_structures/          # Package for data structures used in the CSE machine
|   |   |   ├── enviroment.py         # Module for environment data structures
|   |   |   ├── stack.py              # Module for stack data structure
|   |   |   ├── control_structure.py   # Module for control structure data structure
|   |   |   └── __init__.py           # Marks the directory as a Python package
|   |   ├── utils/                   # Package for utilities used in the CSE machine
|   |   |   ├── STlinearlizer.py      # Module for linearizing the standard tree
|   |   |   ├── util.py               # Module for utility functions for the CSE machine
|   |   |   └── __init__.py           # Marks the directory as a Python package
|   |   ├── machine.py               # Module for the CSE machine
|   |   └── __init__.py              # Marks the directory as a Python package
|   |
|   ├── interpreter/                 # Package for interpreter functionality
|   |   ├── execution_engine.py       # Module containing the logic for the RPAL interpreter
|   |   └── __init__.py              # Marks the directory as a Python package
|   |
|   ├── error_handling/              # Package for error handling functionality
|   |   ├── error_handler.py          # Module containing logic for error handling
|   |   └── __init__.py              # Marks the directory as a Python package
|   |
|   ├── table_routines/              # Package for table routines functionality
|   |   ├── char_map.py              # Module containing logic for character mapping
|   |   ├── fsa_table.py             # Module containing logic for Finite State Automaton (FSA) table
|   |   ├── accept_states.py          # Module containing logic for accept states
|   |   ├── keywords.py              # Module containing set of keywords
|   |   └── __init__.py              # Marks the directory as a Python package
|   |
|   └── utils/                       # Package for utility functionalities
|       ├── tokens.py                # Module containing token class definition
|       ├── node.py                  # Module containing node data structure class definition
|       ├── stack.py                 # Module containing stack class definition
|       ├── token_printer.py          # Module containing token printing function (for debugging purposes)
|       ├── tree_printer.py           # Module containing tree printing function (for debugging purposes)
|       ├── tree_list.py             # Module for listing tree elements
|       ├── file_handler.py           # Module containing file-handling functions
|       └── __init__.py              # Marks the directory as a Python package
|
├── rpal_tests/                       # Directory for tests
|   ├── rpal_sources/                 # Directory for RPAL source code files to test
|   ├── test_generate_tests.py        # Module for generating tests
|   ├── test_generate_ast_tests.py     # Module for generating tests in rpal_sources by pytest for AST
|   ├── test_generate_st_tests.py      # Module for generating tests in rpal_sources by pytest for standard tree
|   ├── test_generate_tests_with_rpal_exe.py           # Module for generating tests in rpal_sources by pytest output generated by real rpal.exe
|   ├── test_generate_ast_tests_with_rpal_exe.py       # Module for generating tests in rpal_sources by pytest for AST output generated by real rpal.exe
|   ├── test_generate_st_tests_with_rpal_exe.py        # Module for generating tests in rpal_sources by pytest for standard tree output generated by real rpal.exe
|   ├── assert_program .py            # Module for checking tests
|   ├── program_name_list.py          # List of program names in rpal_sources directory
|   ├── output.py                     # List of outputs generated by rpal.exe
|   ├── output_ast.py                 # List of AST outputs generated by rpal.exe
|   ├── output_st.py                  # List of standard tree outputs generated by rpal.exe
|   ├── rpal.exe                      # RPAL interpreter
|   ├── cygwin1.dll                   # Cygwin DLL required for execution (if applicable)
|   └── __init__.py                   # Marks the directory as a Python package
|
├── doc/                              # Directory for documentation files
|
├── .vscode/                          # Directory for VS Code settings
|   └── settings.json                 # VS Code settings file
|
├── requirements.txt                  # File containing project dependencies
|
├── Makefile                          # Makefile for automating tasks such as installation, running tests, and cleaning up
|
└── __init__.py                       # Marks the directory as a Python package
```

Conclusion:

The RPAL interpreter project represents a significant accomplishment in the realm of programming language interpretation and execution. Through meticulous planning, implementation, and documentation, the project has delivered a robust software solution capable of parsing and executing RPAL programs effectively.

The development process began with a clear problem description and set of objectives, guiding the creation of key components such as the lexical analyzer, parser, and execution engine. Each component was designed with modularity and extensibility in mind, allowing for future enhancements and modifications to the interpreter's functionality.

The project's architecture is structured around well-defined interfaces between modules, promoting code readability, maintainability, and scalability. By adopting industry-standard practices such as unit testing, error handling, and documentation, the project ensures reliability and usability for end-users.

The interpreter's core functionalities, including lexical analysis, parsing, and execution, are implemented using efficient algorithms and data structures. Finite state automata are employed for lexical analysis, while recursive descent parsing is used for constructing abstract syntax trees. These design choices reflect a thoughtful approach to algorithm design and implementation, resulting in an interpreter capable of handling complex RPAL programs efficiently.

As the project concludes, the RPAL interpreter stands as a testament to the collaborative effort, technical expertise, and innovation of its developers. It serves as a valuable tool for students, researchers, and enthusiasts to explore the intricacies of the RPAL language and its underlying concepts.

Looking ahead, the RPAL interpreter holds the potential for further development and refinement. With ongoing support and feedback from the programming community, it can evolve to meet the changing needs and challenges of language interpretation and execution.

In summary, the RPAL interpreter project represents a significant contribution to the field of programming languages, demonstrating the power of software engineering principles and practices in tackling complex computational problems. It stands ready to empower users with the ability to explore, experiment, and innovate in the domain of language theory and practice.

References:

- **RPAL Language Documentation:** https://sourceforge.net/projects/resil/ - Official documentation providing detailed information about the RPAL language syntax, semantics, and usage.
- **RPAL_Lex.pdf**
- **RPAL_Grammar.pdf**
- **CS3513 Course Materials**
- **Python Documentation:** https://docs.python.org/3/ - Documentation for the Python programming language used to implement the interpreter.
- **Software Engineering Principles and Practices** (general reference)

These references serve as valuable resources for understanding the RPAL language specifications and guidelines, aiding in the development and implementation of the RPAL interpreter.

Additional Tools and Resources:

- **pytest: Python testing framework** (https://docs.pytest.org/en/7.1.x/contents.html) - This reference can help explain how you might use pytest to write unit tests for your RPAL interpreter code. Unit tests ensure individual components of your code function as expected, improving overall code quality and reliability.
- **Markdown: A lightweight markup language** (https://www.markdownguide.org/) - While not directly related to the core functionality of the interpreter, markdown can be a valuable tool for creating clear and concise documentation, like the report itself. It allows for formatting text, including headings, code blocks, and lists, making your documentation more readable and informative.
- **Makefile: Automate build and testing processes** (https://www.gnu.org/s/make/manual/make.html) - A makefile can be used to automate tasks like compiling code, running tests, and creating documentation. This can streamline the development process and improve efficiency. You can explain how a makefile could be used to simplify running the interpreter, its tests, and generating documentation.