CS 3513 – Programming Languages

Programming Project

RPAL Interpreter – Project Report

Group 24

Kumarasekara G.K. 210314E

Nayanathara P.M.C. 210417X

• Problem Description

The task is to implement a lexical analyzer and a parser for the RPAL language referring to the RPAL_Lex.pdf for the lexical rules and RPAL_Grammar.pdf for the grammar details.

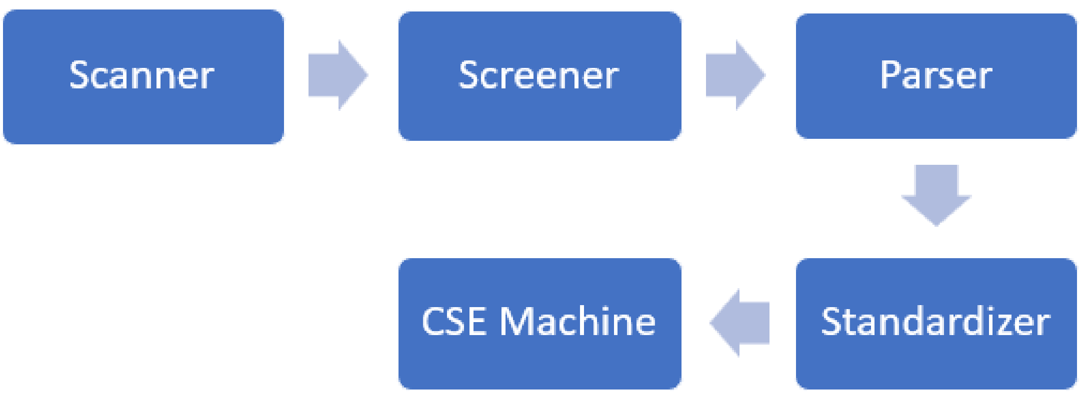
The output of the parser should be the Abstract Syntax Tree (AST) for the given input program. Then it is needed to standardize the AST to return the Standardized Tree (ST) and then should implement the CSE machine to evaluate the programme.

The interpreter should be able to read an input file containing a RPAL program and should output the following:

★ Output of **-ast** switch -> AST
★ Output without -ast switch -> output of the program

• Solution

The following diagram shows a rough sketch of our implementation process.



In this assignment, we used Python as the programming language to implement the RPAL interpreter.

1. Lexical Analyzer

- Read the input file and separate the input to a list of tokens based on the lexical rules of RPAL.
- Identify the type of each token as identifiers, integers, operators etc.
- Merge the required tokens where necessary.
- Screen the set of tokens to drop the unwanted tokens such as comments and whitespaces.
- Identify reserved keywords.
- Return a list of meaningful tokens.

2. Parser

- Iterate over the list of tokens and build the Abstract Syntax Tree.
- Uses recursive descent parsing and the RPAL grammar rules to generate the AST.

3. Standardizer

- Traverse through the AST and standardize the nodes to generate the Standardized Tree with lambda and gamma nodes.

4. Control Stack Environment (CSE) Machine

- Flatten the standardized tree (ST) into a control structure and evaluate the program based on some predefined rules to give the final output.

5. Exception Handling

- Add error handling to the implemented code to detect and report syntax errors if the input program does not match with the RPAL syntax.

• File Structure

myrpal.py The main file that executes the interpreter. Reads the command line input and outputs the result to the terminal.

The following files can be found in the ***src*** folder.

csemachine.py Implements the Control Structure Environment Machine that helps in simplifying and evaluating the ST based on control structure rules.

environment.py Implements the CS environment structure.

lexical_analyzer.py Tokenization of the input and output of an array of tokens with their types and values.

node.py Handles creation, manipulation and pre-order traversal of tree nodes.

rpal_parser.py The parser is implemented in this file. The grammar rules in the RPAL language are coded here to parse and verify the inputs and generate the respective AST.

rpal_token.py Implements the creation and manipulation of tokens using methods such as *make_first_token, make_last_token, make_keyword.*

screener.py Screens the output from the lexical_analyzer and removes unwanted tokens like spaces and comments.

stack.py Implements a stack data structure which helps in AST generation. Includes methods for stack functionalities such as push, pop, is_empty.

standardizer.py Standardizes AST nodes to create the ST with lambda and gamma nodes.

structures.py Defines the control structures used in the CSE machine such as delta, tau & lambda.

• Implementation Details

1. Main Program Execution:

- Implemented in myrpal.py.
- Reads the user input from the command line.
- Reads the file name and the switches ([-l], [-ast], [-st]).
- Generates the following outputs for the given switches.

(input format: python .\myrpal.py <switches> file_name)

Switch	Function Called	Output
No switch	**get_result(file_name)** in csemachine.py	Output of the input program
-l	Reads the file and prints the content	File content
-ast	**parse(file_name)** in rpal_parser.py	Prints the AST for the input program
-st	**parse(file_name)** and **make_standardized_tree(root)** in standardizer.py	Prints the ST for the input program

- Outputs relevant error messages for the incorrect inputs.

2. Lexical Analyzer (Scanner) :

- Implemented in lexical_analyzer.py.
- **tokenize(characters) :** This function takes a list of characters as input and iterates through its elements to separate tokens based on the RPAL lexical rules in RPAL_Lex.pdf. Each token content, token type and the line number is stored in separate lists. Finally, separate token objects are created for each token in the list.

3. RPAL Tokens :

- Implemented in rpal_token.py.
- **Token class:** This initializes Token objects. Token objects have attributes such as content, type, line number, is_first_token and is_last_token. Making an identifier a keyword can be done through the function **make_keyword()** in this class.

4. Screener :

- Implemented in screener.py.
- A list of keywords is defined in this file.

keywords = ["let", "in", "where", "rec", "fn" , "aug", "or", "not", "gr", "ge", "ls", "le", "eq", "ne", "true", "false", "nil", "dummy", "within", "and"]

- **screen(file_name) :** This function takes the input file name as input. It reads the file line by line and separates the whole content in the input file into a list of single characters. Then it calls the **tokenize()** function in the lexical_analyzer.py and gets the processed list of tokens. Then it iterates over the token list and removes the tokens of type <DELETE> which contain spaces and comments. Then it checks for <IDENTIFIER> tokens with the keyword list and makes their type to <KEYWORD> using the **make_keyword()** method in Token class.

- This checks for invalid tokens and returns them as well. And also, error messages are popped where necessary.

5. Parser :

- Implemented in rpal_parser.py.
- **parse(file_name) :** This function gets the input file name from myrpal.py and calls the **screen()** function to get the filtered set of tokens. If invalid tokens are present, it alerts on such invalid tokens. Then it calls **procedure_E(),** from where the RPAL program execution starts.

Separate functions are implemented for non-terminals available in the RPAL language.

procedure_Ew(), procedure_T(), procedure_Ta(), procedure_Tc(), procedure_B(), procedure_Bt(), procedure_Bs(), procedure_Bp(), procedure_A(), procedure_At(), procedure_Af(), procedure_Ap(), procedure_R(), procedure_Rn(), procedure_D(), procedure_Da(), procedure_Dr(), procedure_Db(), procedure_Vb(), procedure_Vl()

- **read(expected_token) :** This function consumes the token at the start of the token list and compares it with the expected token content. If they are matching, the parsing continues. Otherwise, parsing stops with an error message.

- **build_tree(value, num_children) :** This function is called at the places where we need to create AST nodes. When this function is called with the node name and the number of children, a new node is initialized, and the children are initialized from the elements in the stack. And the new node is pushed into the stack.

- **print_tree(root) :** When the root node is given as the argument to this function, it calls **preorder_traversal()** and prints the AST.

6. Stack :

- Implemented in stack.py.

- **Stack class :** The initialization and the functionalities of a stack such as push, pop are implemented here.

7. Node :

- Implemented in node.py.

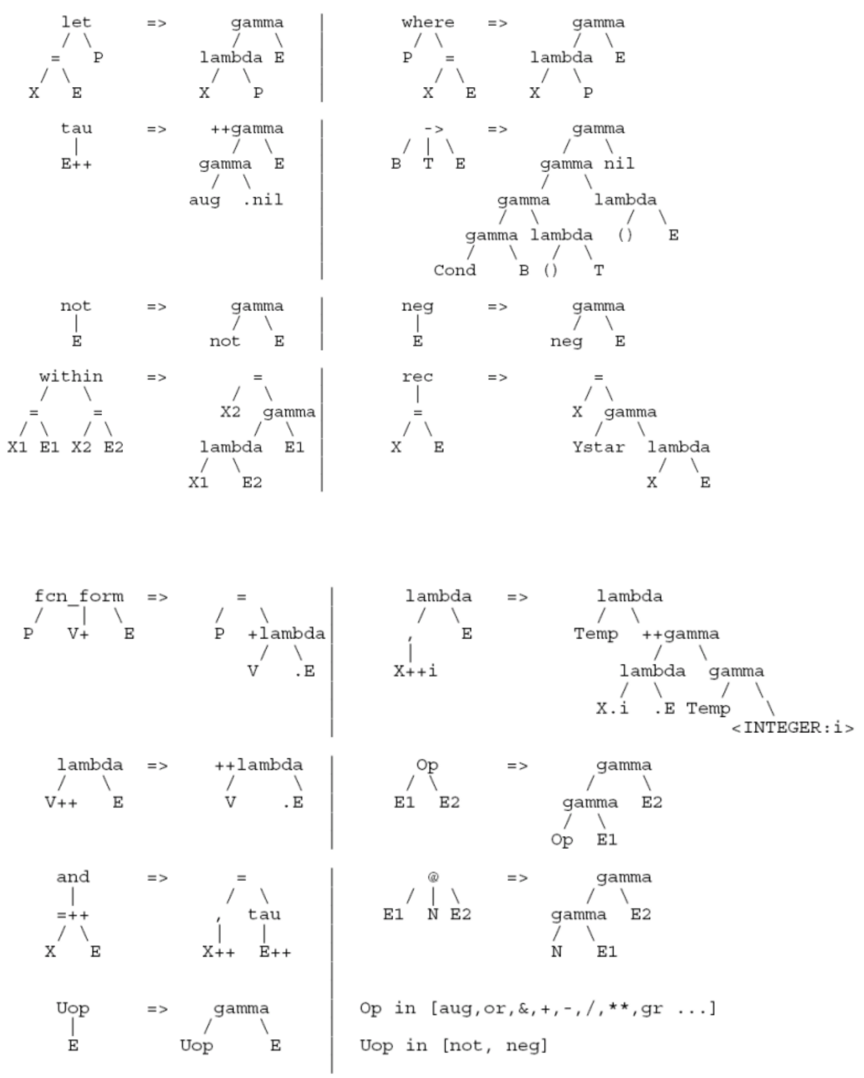
- **Node class :** This initializes a node with attributes such as value, children list and the level number.

- **preorder_traversal(root) :** This recursively traverses through the tree in preorder traversal and prints the AST.

8. Standardizer :

- Implemented in standardizer.py.
- **standardize(file_name) :** This function is called from myrpal.py with the input file name. Then this function calls the **parse()** function in rpal_parser.py to get the AST for the given program. Then it calls **make_standardized_tree()** to get the ST and returns the result.
- **make_standardized_tree(root) :** This function gets the AST as input and standardizes the AST based on the following RPAL subtree transformational grammar. This function uses node and stack structures for this operation. Finally returns the ST.

RPAL SUBTREE TRANSFORMATIONAL GRAMMA



9. Structures :

- Implemented in structures.py.
- **Delta, Tau, Lambda and Eta classes :** These classes initialize respective structures used in the CSE machine.

10. Environment :

- Implemented in environment.py.
- **Environment class :** This class provides a structured representation of an environment within the CSE machine. Environments are used to store variable bindings and facilitate scope management during the program execution.

  This class initializes a new environment with the given number and parent environment. It assigns a unique name to the environment, initializes an empty dictionary for variables and sets the parent reference.
- **add_child(child) :** This function adds a child environment to the current environment. It appends the child to the list of children and and updates the child's variable dictionary with the current environment's variables.
- **add_variable(key, value) :** This function adds a variable binding to the current environment. It stores the variable name as the key and the corresponding value in the variable dictionary.

11. CSE Machine :

- Implemented in csemachine.py.

- **get_result(file_name) :** This function is called from myrpal.py with the input file name. It then calls the **standardize()** function in standardizer.py to get the ST. Then it initializes the control stack and environments, and then calls the **apply_rules()** function to execute the program. Finally, it returns the result of the program execution.

- **apply_rules() :** This function simulates the execution of the CSE machine by applying CSE machine rules based on the control structures and the current state of the control stack and environments. It iteratively applies the rules until the control stack is empty, updating the stack and environments as needed.

- **generate_control_structure(root, i) :** This function is responsible for generating control structures based on the ST. It recursively traverses the ST and constructs the control structures such as Lambda, Delta, Tau and Eta.

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	

	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]

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

	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	

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

- **lookup(name) :** This function handles the lookup of variables and values in the current environment. It takes a variable name as input and returns its corresponding value from the environment.

- **built_in(function, argument) :** Implements built-in functions such as string operations and type checks. It takes a function name and its arguments as input and performs the specified operation.

● Testing and Documentation

Test cases were used for debugging purposes and to ensure that the interpreter works accurately and generates the correct output.

Running the RPAL interpreter :

- To get the output of the program :

  python .\myrpal.py file_name

- To get the AST of the program :

  python .\myrpal.py -ast file_name

- To get the ST of the program :

  python .\myrpal.py -st file_name

- To read the input program :

  python .\myrpal.py -l file_name

GitHub repository link for source files :

https://github.com/Gayan-Kaushalya/RPAL

The GitHub repository contains programs to test the screener and the parser separately.

• Special Notes

1. When running the myrpal.py file in Ubuntu,

   python .\myrpal.py file_name

   Command may not execute properly. In such cases,

   - Try using the word “python3” instead of “python” in your command.
   - If the problem is not resolved, use the slash (/) symbol instead of backslash (\) in your command.
   - If the problem is still not resolved, check whether the file path is correct with spellings and capitalization.

     eg :- The command below does not execute properly because the file path is not properly capitalized. (T in “Tests” should be capital.)

     ```
     gayan@GKKLaptop:~/rpal$ python3  ./myrpal.py tests/add
     File not found.
     ```

     When the letter T is capital, the command executes properly.

     ```
     gayan@GKKLaptop:~/rpal$ python3  ./myrpal.py Tests/add
     15
     ```

2. Since the rpal.exe file identifies strings only when they are within single quotation marks, we emulated that behaviour in our interpreter too. Therefore, strings represented as ‘rpal’, ‘CS3513’ are accepted as strings while strings represented as ‘‘rpal’’, ‘‘CS3513’’ and strings represented as “rpal”, “CS3513” are not accepted.

   If the RPAL program has a string which is not in the accepted format, you may get error messages like this while testing the program.

```
gayan@GKKLaptop:~/rpal$ python3 ./myrpal.py Tests/conc1
Syntax error in line 1: Identifier, Integer, String, 'true', 'false', 'nil', 'dummy' or '(' expected
gayan@GKKLaptop:~/rpal$ python3 ./myrpal.py Tests/conc1
Syntax error in line 2: ')' expected
```

   In such cases, it is advised to format your strings in the correct format and test the program again.

• Conclusion

Scanning, screening, and parsing are some of the main steps in the process of an Interpreter.

This implementation gives a clear understanding of the step-by-step procedure of the interpretation process and how to implement such an interpreter using a high-level language like Python.