**In21-S4-CS3513– Programming Languages**

**Programming Project 01**

**GROUP 50**

**W.A.R.T. FONSEKA – 210170G**

**K.S. RANASINGHE – 210518H**

**Project description**

The program should read an input file which contains a RPAL program. Then the lexical analyser tokenizes the program. The parser then builds a parse tree using the token list. Then we build a Abstract Syntax Tree using the parse tree. Then we standardize the AST. Finally, the CSE machine flattens the standardized AST and evaluate the program and output the result. The complete project is written in python.

As mentioned above there are multiple components in the program which we have implemented separately for ease of use. We added some additional functionality as improvements to program as well. Diagram given below gives a high-level definition for the implementation.

File containing the RPAL program

-cse switch

Result

CSEM Evaluation

Control Structures

AST

Standardized AST

-st switch

-cs switch

-ast switch

Standardized AST

AST

Parse Tree

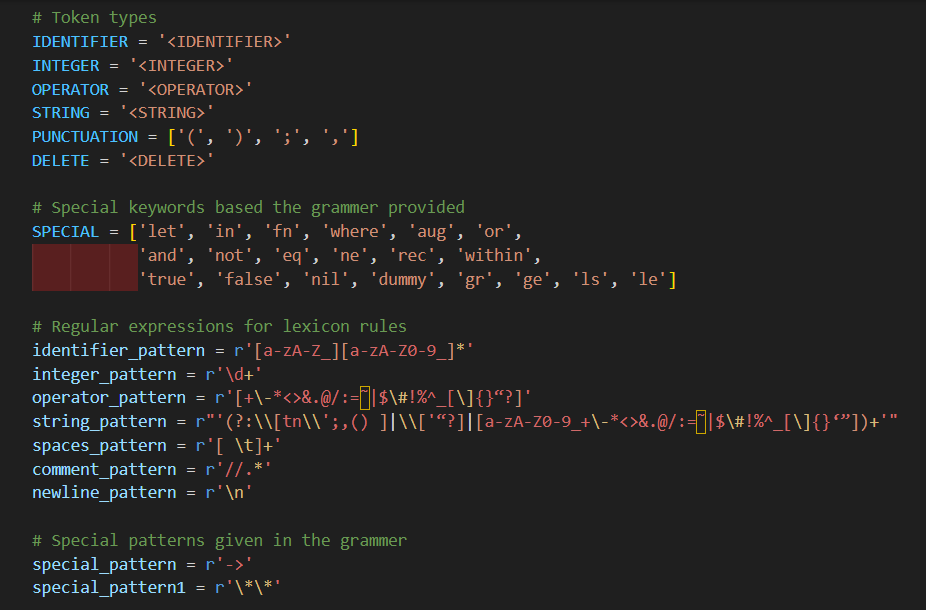
Token List

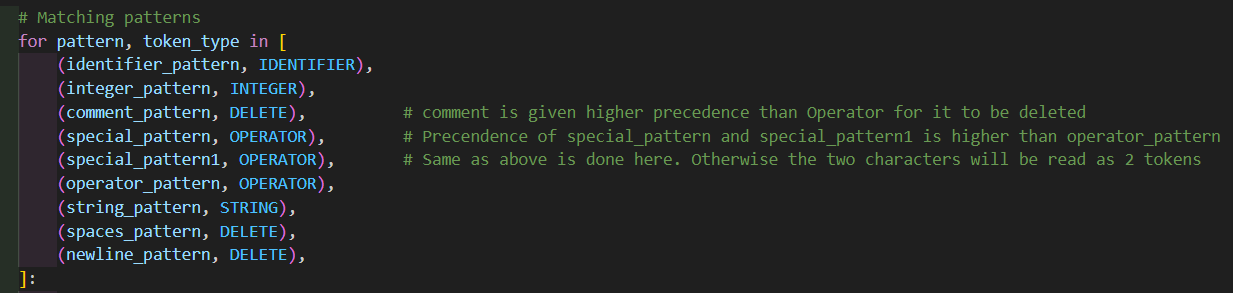
***Program Flow***

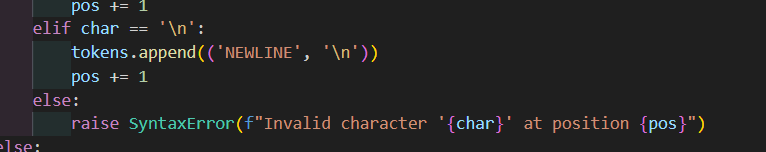
no switch

Next let’s analyse the individual components. For this we will go through the individual python files as the components were initialized in separate files.

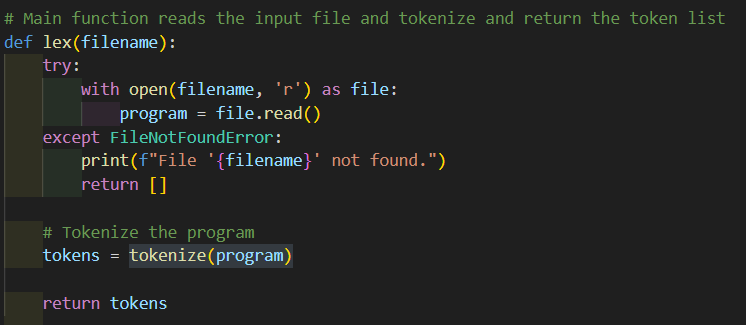
**Lexical Analyser : Scanner.py**

In this we have the tokenize(program) function which breaks the program into a token list. The different lexical rules are defined as shown below.

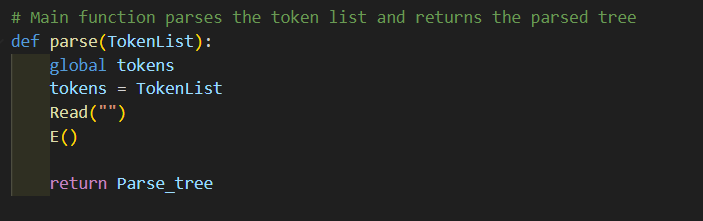
Next for each combination of characters in the program we need to identify the different token types to assign. There is also a precedence for token types to avoid any identification issues as given below.

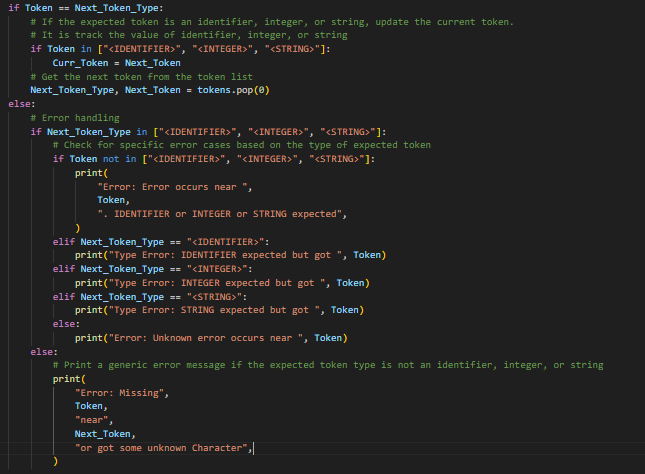
Based on whatever the token type identified we add them to the token list. We handle spacing and new lines also here. Also, if any incorrect character combination would output an error.

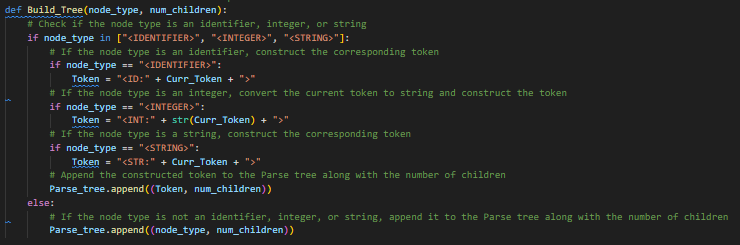
Finally, we have the lex(filename) which reads the file and input the program to the tokenize function and return the token list.

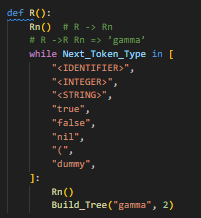


**Parser : Parser1.py**

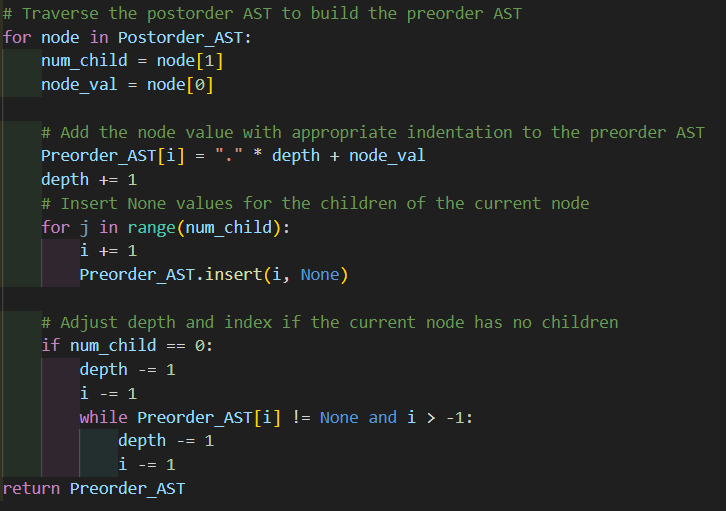
Here, we perform syntax analysis to catch any syntax errors and throw an error message if any are found. Our main function, parse(TokenList), takes the output token list from the scanner, calls the Read(Token) function for initialization itself, and invokes the E(), starting function of the recursive descent parser. And finally, it’s returns the parser tree to next step.

The Read(Token) function is responsible for verifying that the upcoming token is syntactically correct and consuming it. If the token is syntactically incorrect, it tries to provide a comprehensive error message, detailing what the error is, what the expected token is, and where the error occurs.

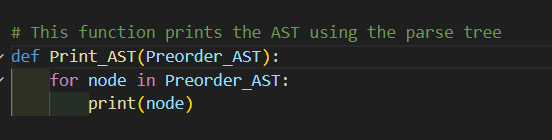
The Build\_Tree(node\_type, num\_children) function is responsible for building the parser tree in postorder and appending it to a list.

 We have separate functions for each grammar rule, and within these functions, we call the Read(Token) function and the Build\_Tree(node\_type, num\_children) function in appropriate places. Some example functions are as follows. Some of these functions have error-handling methods to manage syntax errors that cannot be handled by the Read(Token) function itself.

**AST Generation : AST.py**

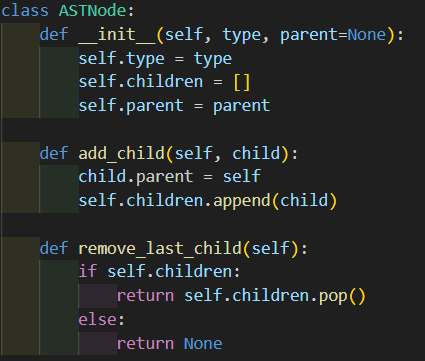
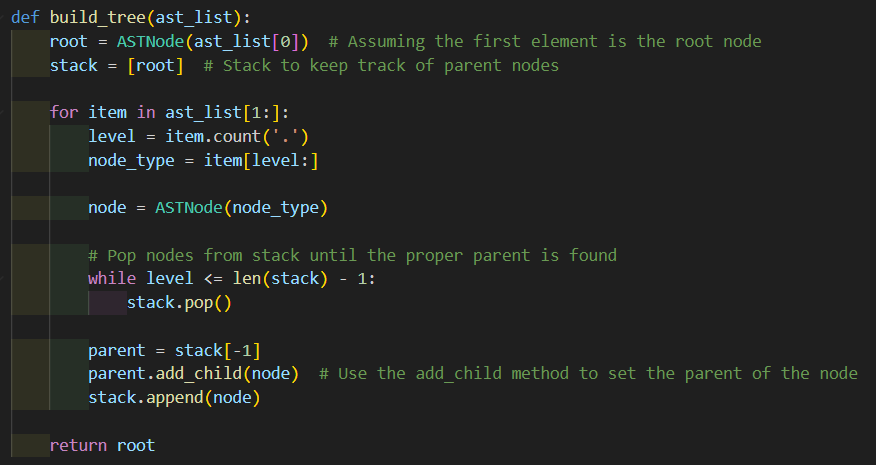
In here we have the function Build\_Preorder\_Tree(AST) which takes the parse tree as the input and convert it to Abstract Syntax Tree. In here we are generating the AST structure that is given in the project document. We start by reversing input parse tree list and iterating the list while creating a new list with the AST structure as shown below.

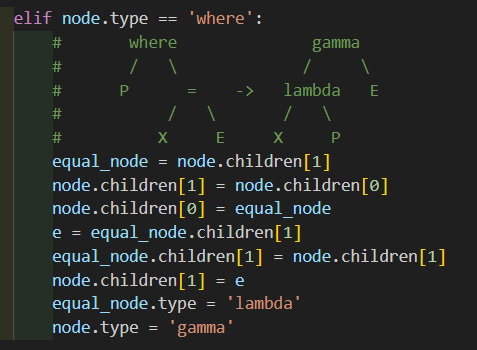
This function returns a list containing the AST structure. Which we will be output using the Print\_AST(Preorder\_AST) if the -ast switch is provided.

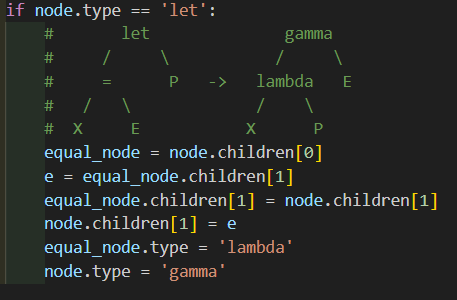


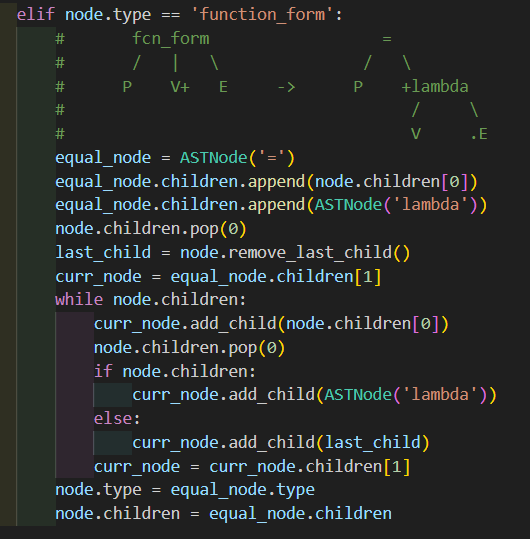
**AST Standardization : ST.py**

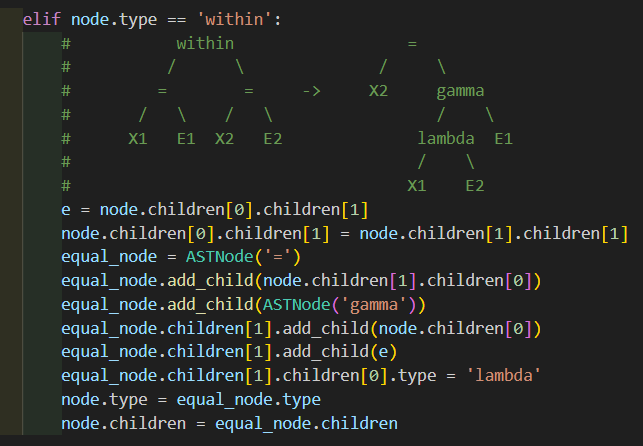
In the previous step(AST generation) we did not create a tree structure using the AST. In here we first create the tree structure using the build\_tree(ast\_list) function as given below.

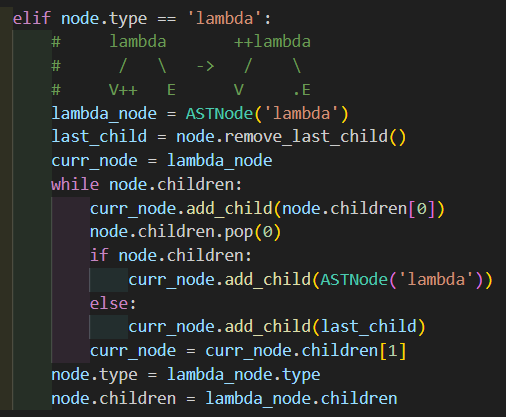
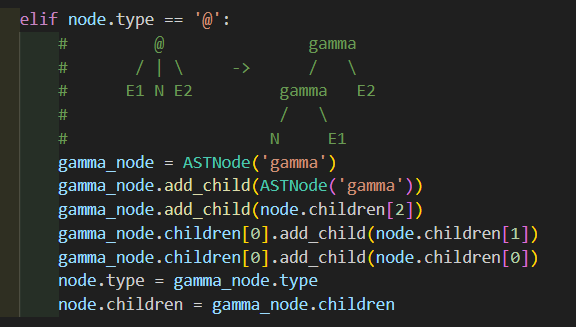


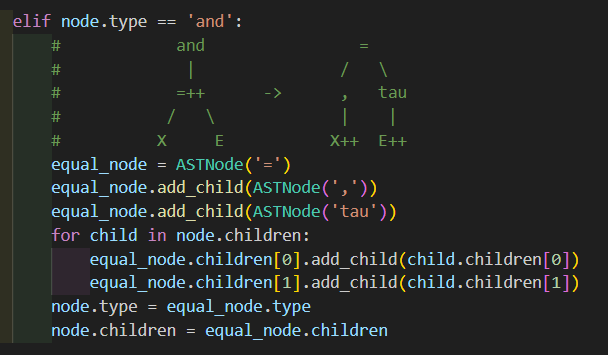
Next we will iterate through this tree while standardizing the necessary nodes. This is carried out by the standardize(node) function. This function takes the root node of the AST as input. As given by the CSE machine rules we don’t have to standardize every node. All the nodes we standardized are given below.

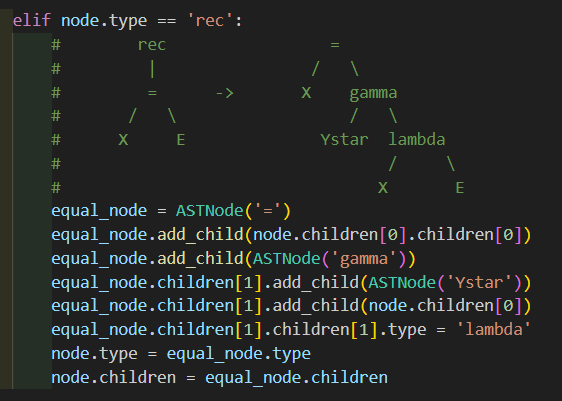






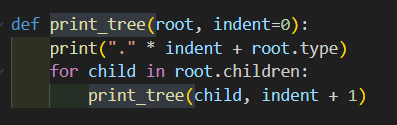






Altogether we standardized 8 node types as ‘let’, ‘where’, ‘fcn\_form’, ‘within’, ‘@’, ‘lambda’, ‘and’ and ‘rec’. All the other node types will be handled by the cse machine as given by the rules.

There is also the print\_tree(root, indent=0) function which would output the standardized AST if the -st switch is provided.



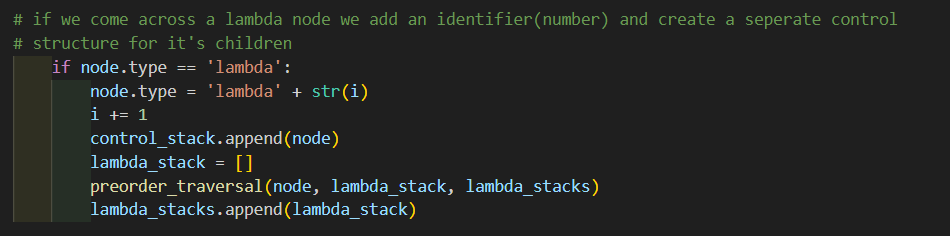
**CSE Machine : ControlStructure.py, CSEM.py**

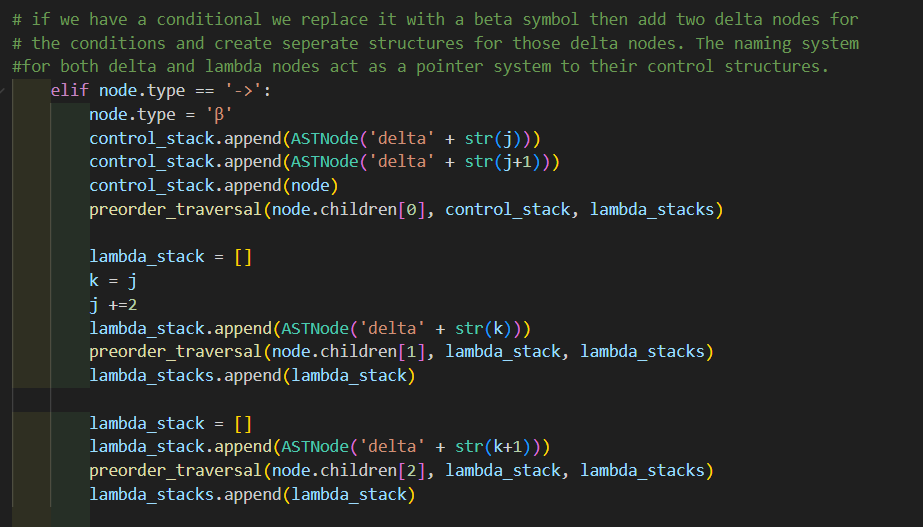
We are using two python scripts for the CSE machine process. ControlStructure.py generates the control structures for the standardized AST and the CSEM.py evaluates the control structures to generate the result.

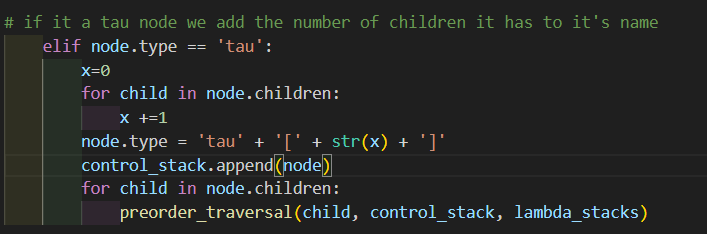
**Control Structure Generation : ControlStructure.py**

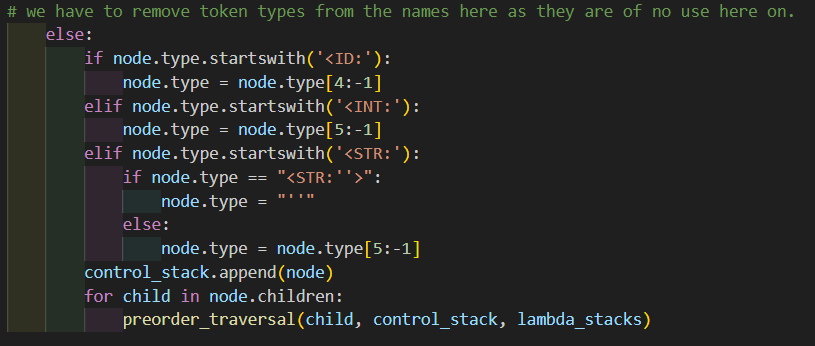
The standardized tree created in the ST.py is first given as input to the function given below. preorder\_traversal(node, control\_stack, lambda\_stacks)

This function flattens the tree into a list containing the control structures. control\_stack contains the starting structure while the rest of structures are in lambda\_stacks. When flattening some nodes need to be handled explicitly as shown below.

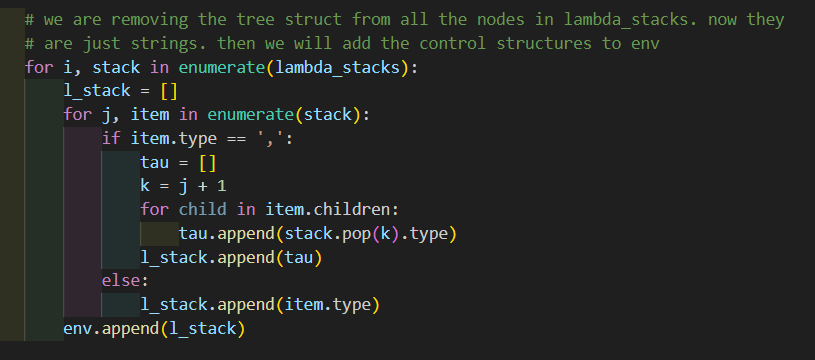
*Lambda node*

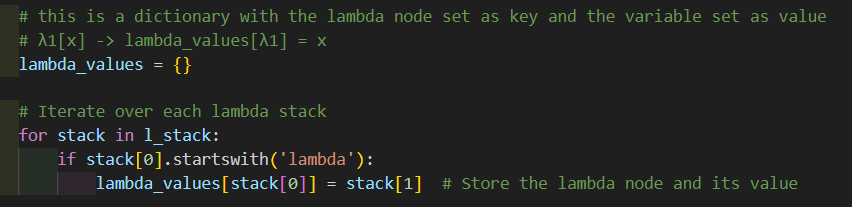
*Beta node*

*Tau node*

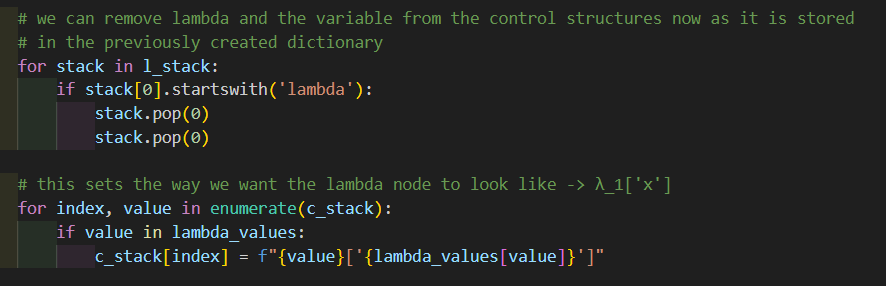
*Token type*

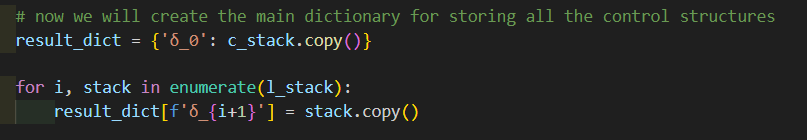
The above function is called inside the main function create\_control\_structure(ast\_list). This function first runs the flattening function and gets the lists containing the control structures and further modify them to get the exact nature of the required control structures. First we will remove the tree structure from all the elements for ease of use.

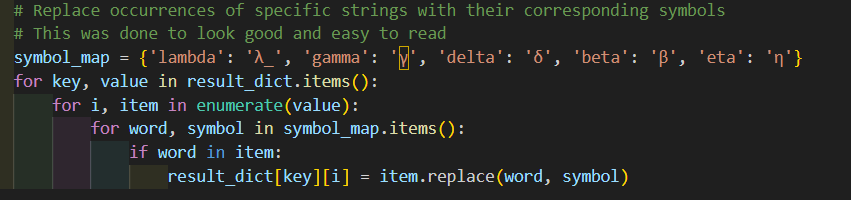


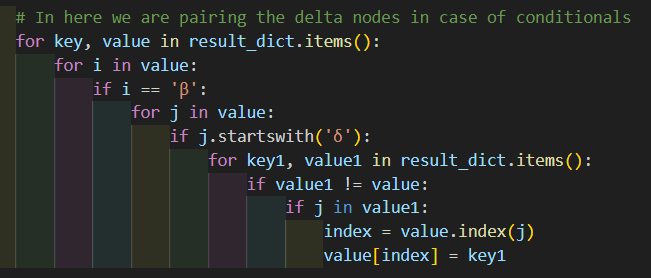
Next, we will use a dictionary to capture the relationship between the lambda nodes and their respective variables.

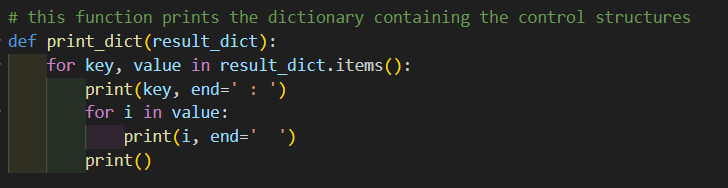
Since the lambda nodes are stored in a separate dictionary, we can remove them, and the variables associated with them from the lists we are working on. Also, we can set the exact nature we want the lambda nodes to look in the control structure.



Now we can build the dictionary with the control structures used in the CSE machine as all the necessary modifications have been made.

Also made some changes to the Greek symbols used in the control structure for readability.

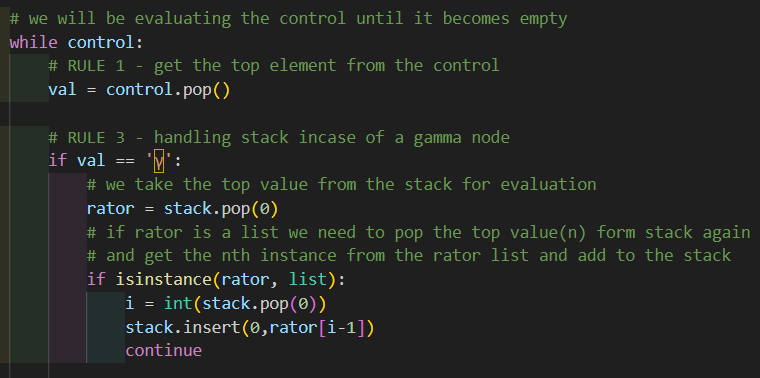
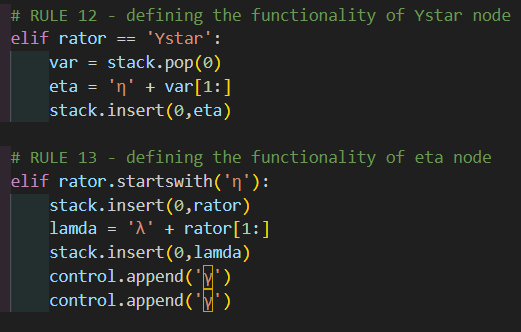
Finally, we had to handle the case of delta nodes. We need to point to the correct control structures when handling conditionals.

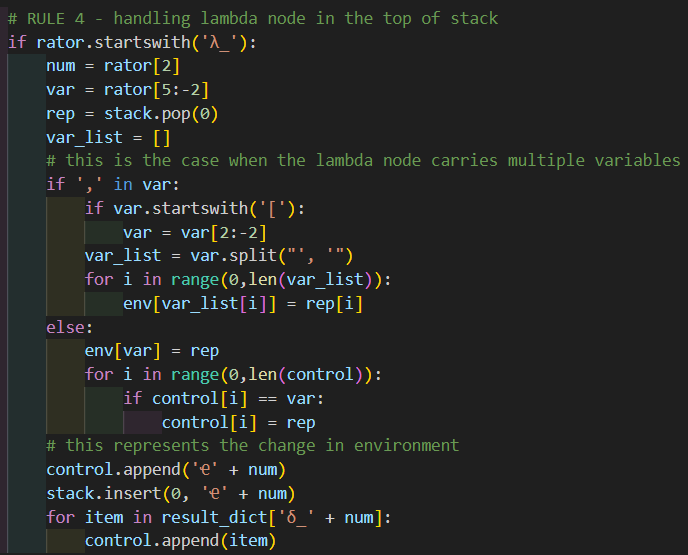
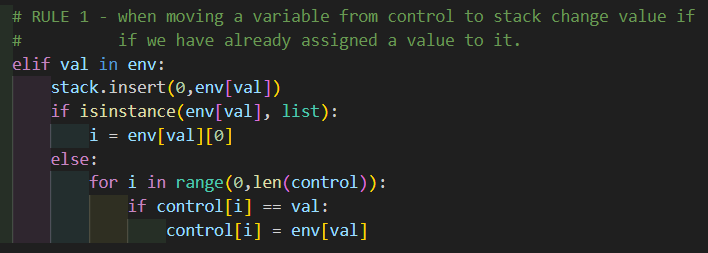
There is also another function print\_dict(result\_dict) which outputs the control structures created by the above function.

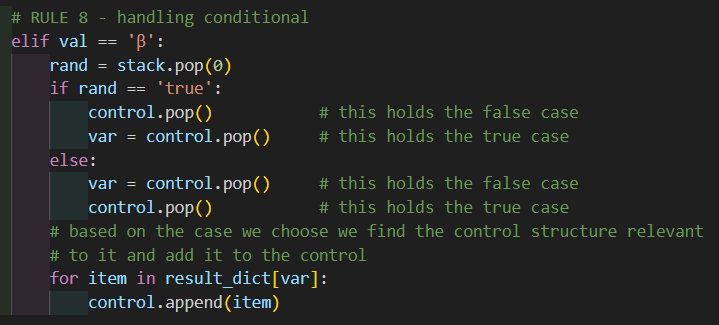
**Control Structure Generation : CSEM.py**

The dictionary containing control structures is evaluated in here by the function given here evaluate(control, stack, result\_dict, env, out=True)

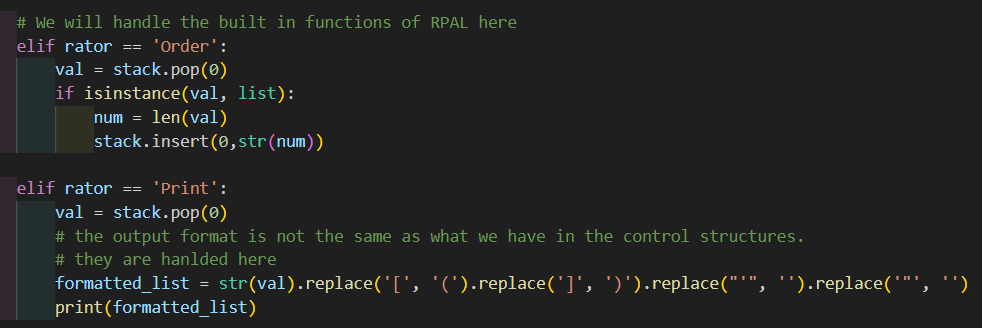
We have to input both the control and stack used in the evaluation as well. The ‘out’ parameter is used to determine whether the evaluation process needs to be shown as output. If it is set to true, we output the evaluation of control and stack.

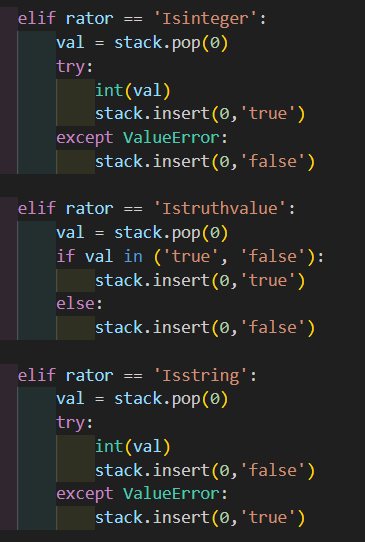
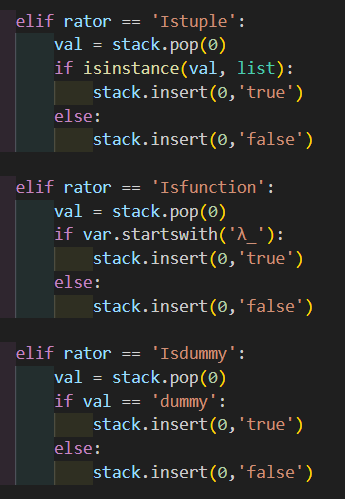
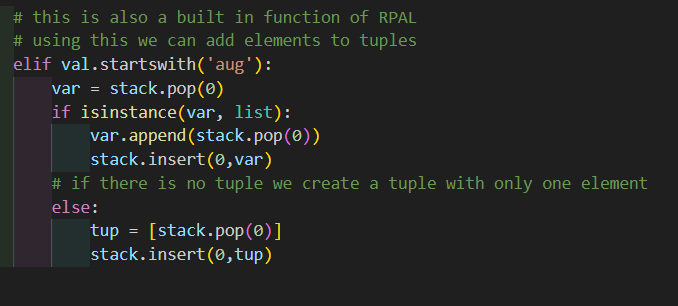
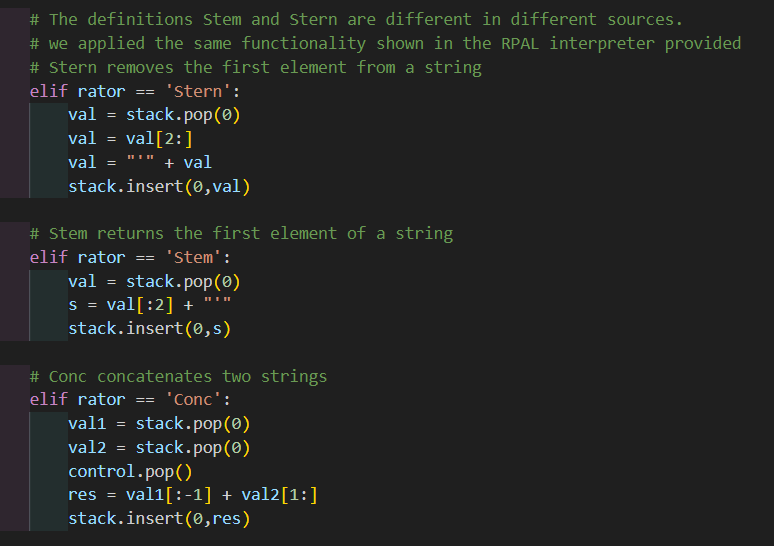
In this function we will be handling all the CSE rules as mentioned in the notes. Given below are some of the implementations for the rules.



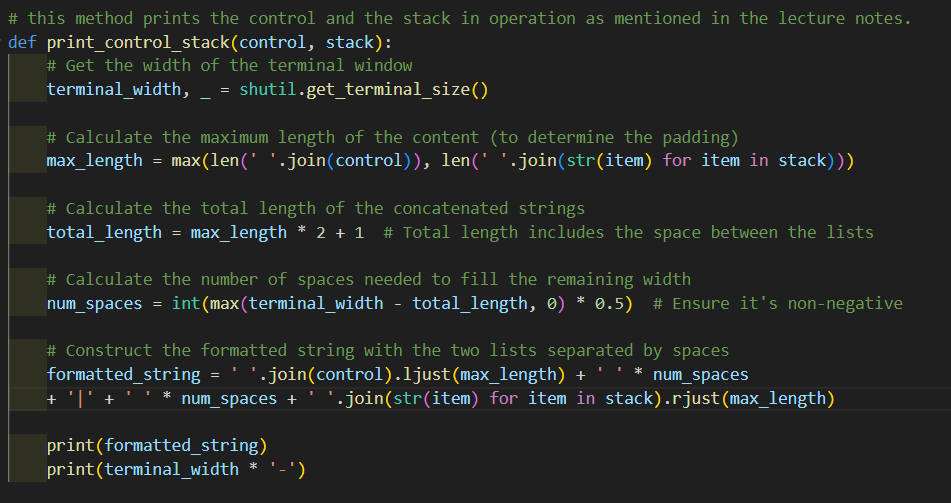


We also had to handle the built-in functions of RPAL language. Finding the definitions of some of these functions were difficult as they had varying definitions in different sources. So, we used the implementation as shown by the RPAL interpreter given to us.



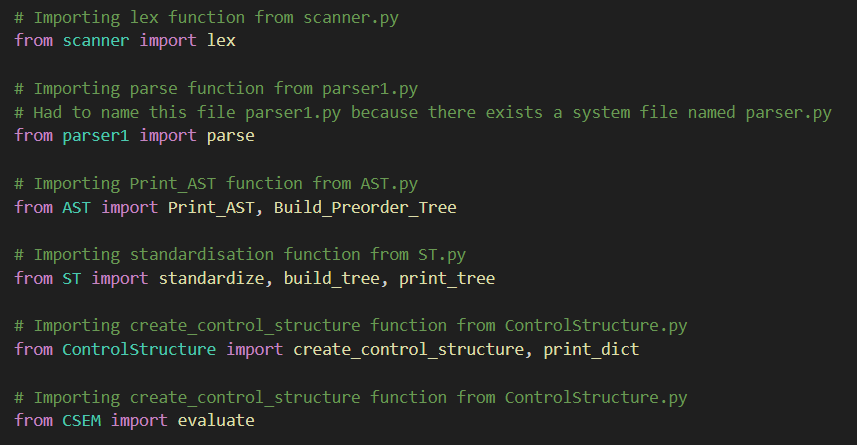


Finally, we have the function print\_control\_stack(control, stack) which output the current state control and the stack. This function is called inside the evaluate function for each iteration to capture the whole evaluation process.



**Main Program : myrpal.py**

This is the base program which executes all the other programs. We first import all the necessary functions from the respective files.



Next we will simply execute these functions one by one according to the flow that was stated earlier

def main():

    try:

        if len(sys.argv) not in [2,3]:

            print("Usage: python scanner.py <filename> [-ast]")

            return

        # Read filename from command-line argument

        filename = sys.argv[1]

        # Read tokens from the file using the lex function

        tokens = lex(filename)

        # Call the parse function to generate the Parse tree

        Parse\_tree = parse(tokens)

        # Next we will build the AST using the parse tree

        AST = Build\_Preorder\_Tree(Parse\_tree)

        # We will standardize the generated AST using this function

        ST = standardize(build\_tree(AST))

        # This will generate the control structures for CSE machine evaluation

        CS = create\_control\_structure(AST)

        control = []

        stack = []

        control.append('Ҽ0')

        stack.append('Ҽ0')

        for val in CS['δ\_0']:

            control.append(val)

        env = {}

        if len(sys.argv) == 3:

            # if -ast switch is given we will print the AST

            if sys.argv[2] == "-ast":

                Print\_AST(AST)

            # if -st switch is given we will print the standardized AST

            if sys.argv[2] == "-st":

                print\_tree(ST)

            # if -cs switch is given we will print the control structures for the program

            if sys.argv[2] == "-cs":

                print\_dict(CS)

            # if -cse switch is given we will print the CSE machine evaluation of the control structures

            if sys.argv[2] == "-cse":

                evaluate(control,stack, CS, env)

        # if no switch is given we will just output the result of the program

        else:

            evaluate(control,stack, CS, env, False)

    except:

        print()