**CS 3513 - Programming Languages Programming**

**Project**

**By Group 130**

**SAJEENTHIRAN P. 210553J**

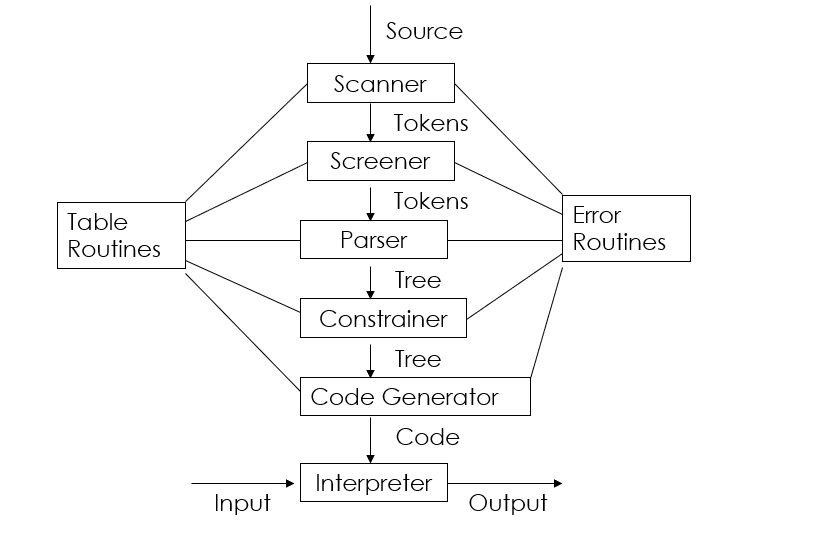
[**DISSANAYAKE W.A.A. 210149C**](https://online.uom.lk/user/view.php?id=26010&course=22900)

**RPAL Interpreter Analysis Report**

**1.Problem Description**

It is required 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 rules. Output of the parser should be the Abstract Syntax Tree (AST) for the given input program. Then implement an algorithm to convert the Abstract Syntax Tree (AST) to a Standardised Tree (ST) and implement a CSE machine. The program should be able to read an input file which contains a RPAL program. Output of your program should match the output of “rpal.exe“ for the relevant program.

**2.Solution**



* Tokenizer
  + Implement a Lexical analyzer using a hard coded scanner .
  + The Tokenizer Screens the input file and generates tokens .
  + It also merges the required tokens.
  + Identifies reserved keywords
  + Removes Comments and white spaces
  + Returns an array of tokens
* AST parser
  + Iterates through the token sequence and builds a Abstract syntax tree
  + Uses recursive decent parsing while building the AST Tree
* Standardizer
  + Performs pre-order traversal of the AST and standardizers each node
* Control Structure Generation
  + Performs a pre-order traversal of the AST Node while maintaining a FIFO Queue and generates control structures
* Control Structure Environment evaluation
  + Maintains a Control Structure array and a stack .
  + Pops each element in the control structure array and executes a rule based on the stack top and the environment

**File Structure**

The RPAL interpreter project includes several key components, each playing a crucial role in the process of interpreting RPAL programs:

***.git***: Version control directory.

***ASTNode.py***: Handles the creation ,manipulation and Standarizationof nodes within the Abstract Syntax Tree (AST).

***controlStructure.py***: generates and Manages the control structures within the interpreter

***cseMachine.py***: Implements the Control structure environment machine, crucial for executing the Standardized Tree.

***Environment.py***: Manages environment settings such as variable bindings, essential for the execution phase.

***myrpal.py***: the main executable script for the interpreter which contains the parser as well.

***output\_files***: Directory intended for storing output, used for debugging and testing outputs.

***README.md***: Provides documentation for setup, usage, and functionalities of the interpreter.

***test\_cases***: Contains test cases for validating the interpreter’s functionality across different scenarios.

***Tokenizer.py***: Handles the tokenization of the input RPAL program into meaningful units (tokens).

**Test.py** : runnig test cases

**Detailed Functionality Analysis**

***Tokenization Process:***

The process starts in the myrapl.py .

File : myrpal.py

Function : The program gets the command line arguments .Then it initiaes a token list and calls the get\_next\_token() method in the Tokernizer.py iteratively and append the tokens in the token list.

File: Tokenizer.py

**scanner**

Breaks down the RPAL program text into tokens using a Lexical analyzer by hard coding , categorizing each token by type (e.g., keyword, identifier).

The TokenType Enum is used to identify the token type.

**Screening**

The Screener class in the Tokernizer.py is reponsible for the scrreing the toekns .

The merge\_tokens() method merges tokens such as \*\* , -> , => ,=<

The remove\_comments() methd removes comments

The screen\_reserved\_keywords() method idetifies the reserved keywords

***Parsing and AST Generation:***

Files: myrpal.py

Function: Utilizes a recursive descent parsing methodology to create an AST from the identified tokens. This AST Parser Class is responsible for the parsing . The parsing begins by calling the procE() method in the ASTParser class.

The ASTParsser Object contains the following methods which are part of the recursive desscent parser ,

* procE
* procEw
* procT
* procTa
* procTc
* procB
* procBt
* procBs
* procBp
* procA
* procAt
* procAf
* procAp
* procR
* procRn
* procD
* procDa
* procDr
* procDb
* procVb
* procVL

These methods are implemented by referring to the RPAL\_Grammer..pdf uploaded in <https://online.uom.lk/mod/resource/view.php?id=379221> .

***AST to Standardized Tree Transformation:***

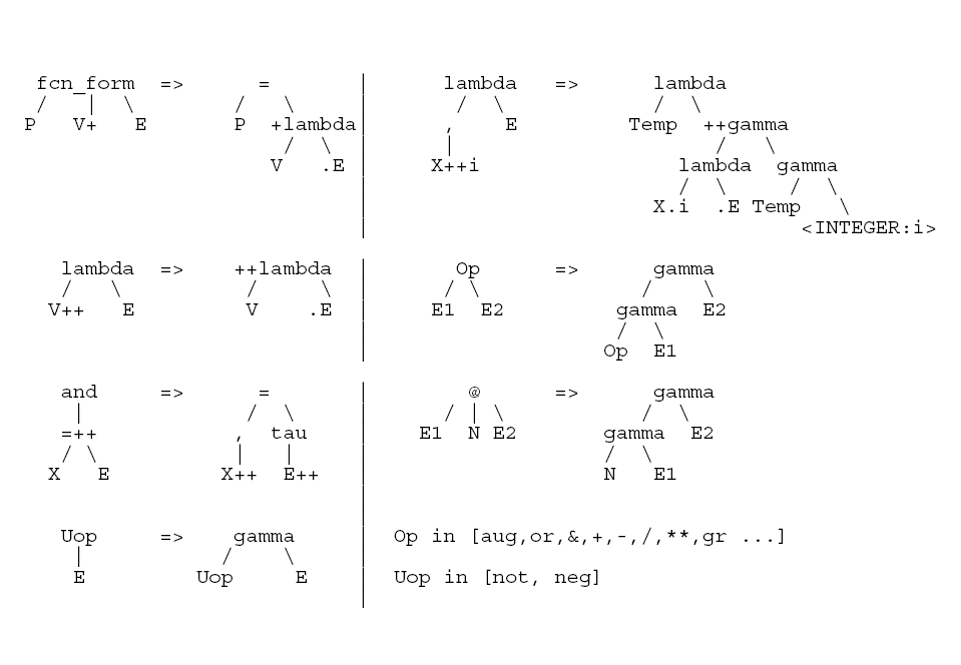
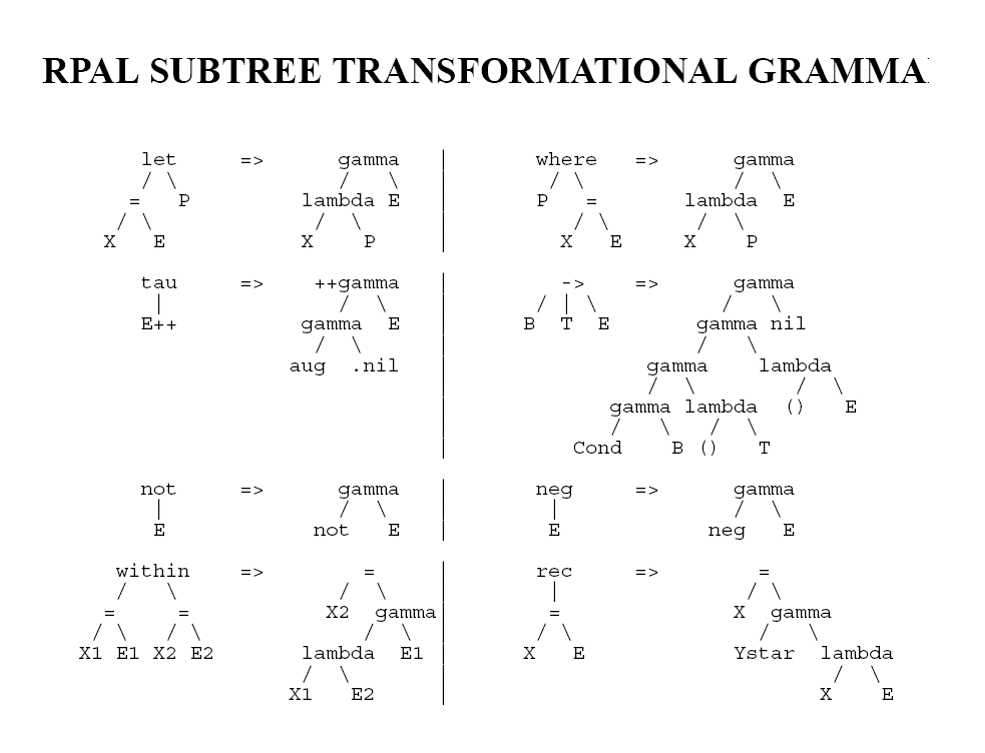
File: ASTNode.py

The standarize() method in the ASTNode class is responsible for standarizing the AST.

We standarized the followinng nodes ,

* let
* where
* within
* function\_form
* and
* rec
* @

Using the following reference ,



***Control Structure generation:***

File: controlStructure.py

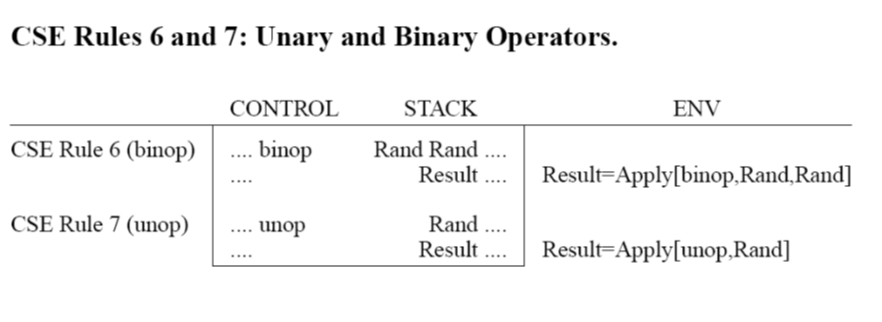
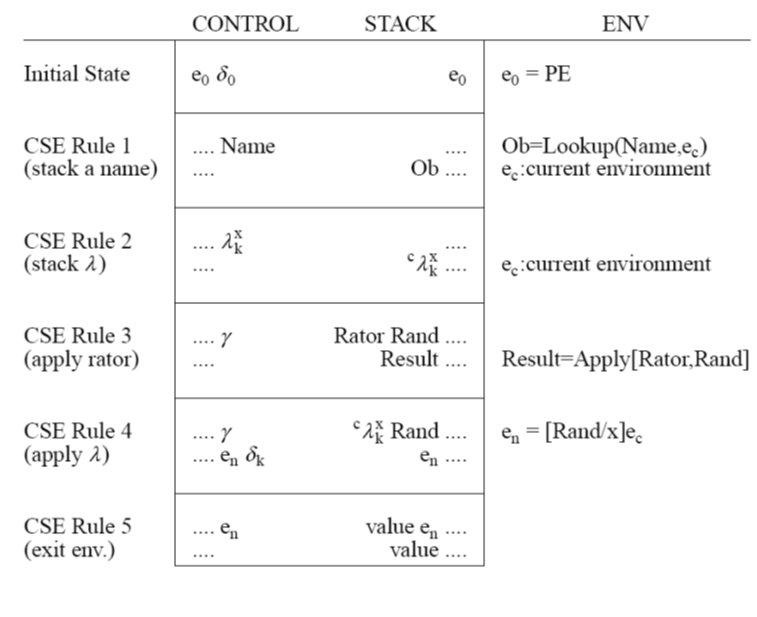
Uses two data strucutes . One is the binary AST Node and the other is the FIFO Queue.

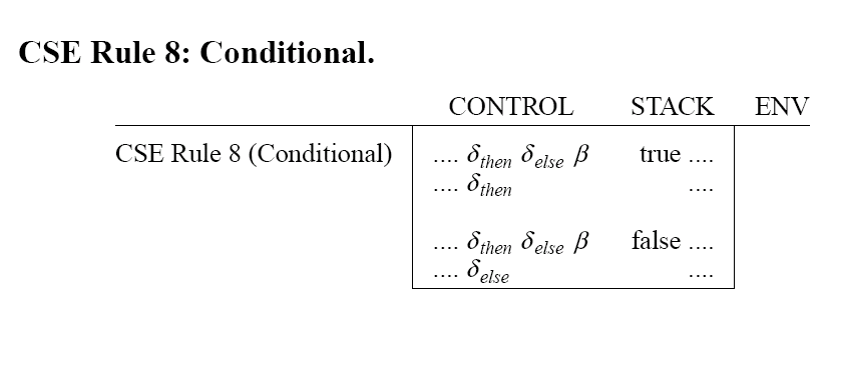
The ControlStrucutreGenerator class is responsible for generation of control structures. The starting point is the generate\_control\_strucutres() method .It initially traverses the Standardized tree in pre-Order fashion while considering lambda nodes as leaves and pushing the right child of the lambda to the FIFO queue. After the first traversal, the algorithm iterates the queue and traverses the nodes and generates control controls and sets it to the map\_ctrl\_structs dictionary with the control structure id as the key and the list of nodes as values for that control structure .

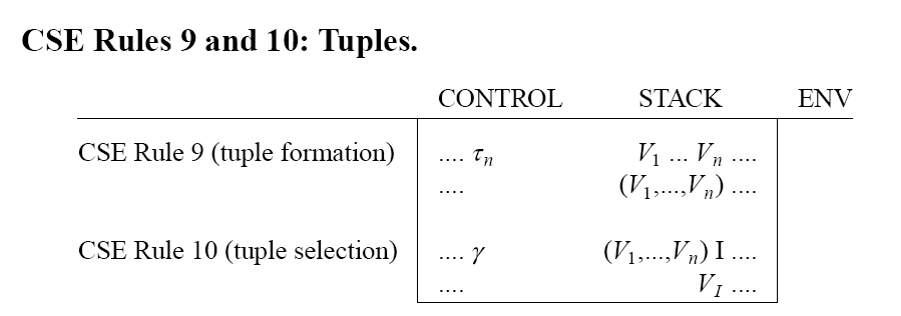
***CSE Machine Execution:***

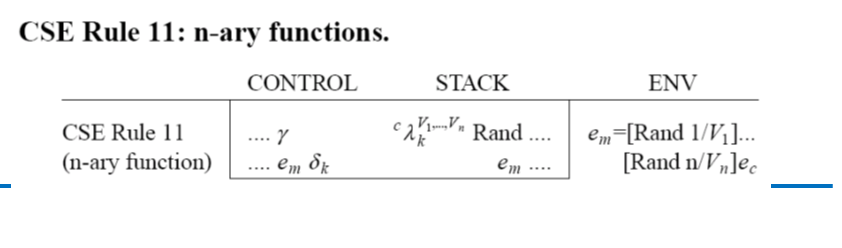
File: cseMachine.py

The CSEmachine class is responsible for evaluating the control structures . The process starts in the execute() method . It first pushes the e0 environment variable with empty key values to both the control structure list and the stack .Then Pushes the contents of 0 th control structure to the control strucutre list. And then based on the top of the control strucutre and the stack top , the CSEMachine performs a operation based on the following CSEMachine Rules ,









***Utility Classes :***

**Environment :**

Location : Environment.py

Purpose : Store the environment variables as a liked list . Can traverse the LinkedList from the tail until the specific variable is found

**Beta:**

Location : controlStrucutre.py

Purpose : used to represent the beta node in the control strucutre.

**Eta :**

Location : CSEMachine.py

Purpose : Used to represent the Eta node in the control stucture which is used in recursive functions

**Tau :**

Location : controlStrucutre.py

Purpose : Used to represent the n-ary elements in the control strucutres

**Testing and Documentation**

Test Cases: Found in the test\_cases directory, these are essential for ensuring the interpreter handles all specified cases correctly, helping to identify bugs and edge cases.

Run the test.py to run all the test cases .

**Runnig the script :**

In order to get the output only ,

python .\myrpal.py file\_name

In order to get the AST tree printed in the command line

python .\myrpal.py file\_name -ast

**Conclusion**

The RPAL interpreter is a comprehensive tool designed to parse and execute programs written in the RPAL language. Its modular design, documented in the various Python scripts and support files, aligns closely with the reference project. This alignment includes robust tokenization, parsing, AST management, and execution mechanisms, making it a well-rounded tool for educational and developmental purposes in the context of programming language theory and compiler construction.