In21-S4-CS3513– Programming Languages

Programming Project 01

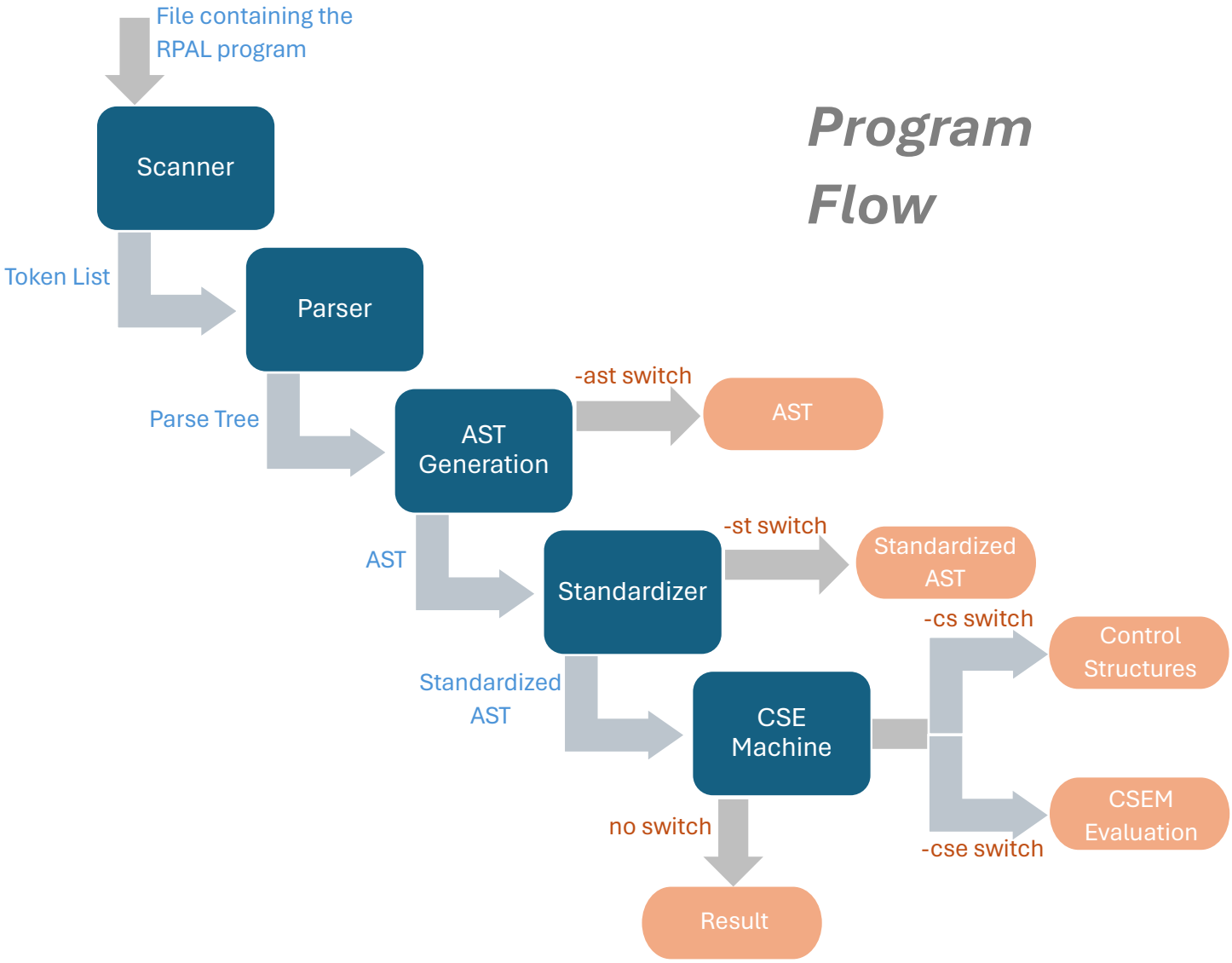
GROUP 50

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.



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.

```
# Token types
IDENTIFIER = '<IDENTIFIER>'
INTEGER = '<INTEGER>'
OPERATOR = '<OPERATOR>'
STRING = '<STRING>'
PUNCTUATION = ['(', ')', ';', ',']
DELETE = '<DELETE>'

# Special keywords based the grammer provided
SPECIAL = ['let', 'in', 'fn', 'where', 'aug', 'or',
           'and', 'not', 'eq', 'ne', 'rec', 'within',
           'true', 'false', 'nil', 'dummy', 'gr', 'ge', 'ls', 'le']

# Regular expressions for lexicon rules
identifier_pattern = r'[a-zA-Z_][a-zA-Z0-9_]*'
integer_pattern = r'\d+'
operator_pattern = r'[+\-*<>&.@/:=~|$\#!%^_[\]{}"?]'
string_pattern = r"'(?:\\[tn\\';,() ]|\\['"?]|[a-zA-Z0-9_+\-*<>&.@/:=~|$\#!%^_[\]{}"])+'"
spaces_pattern = r'[ \t]+'
comment_pattern = r'//.*'
newline_pattern = r'\n'

# Special patterns given in the grammer
special_pattern = r'->'
special_pattern1 = r'\*\*'
```

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.

```
# Matching patterns
for pattern, token_type in [
    (identifier_pattern, IDENTIFIER),
    (integer_pattern, INTEGER),
    (comment_pattern, DELETE),          # comment is given higher precedence than Operator for it to be deleted
    (special_pattern, OPERATOR),        # Precendence of special_pattern and special_pattern1 is higher than operator_pattern
    (special_pattern1, OPERATOR),       # Same as above is done here. Otherwise the two characters will be read as 2 tokens
    (operator_pattern, OPERATOR),
    (string_pattern, STRING),
    (spaces_pattern, DELETE),
    (newline_pattern, DELETE),
]:
```

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.

```
            pos += 1
        elif char == '\n':
            tokens.append(('NEWLINE', '\n'))
            pos += 1
        else:
            raise SyntaxError(f"Invalid character '{char}' at position {pos}")
else:
```

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

```
# Main function reads the input file and tokenize and return the token list
def lex(filename):
    try:
        with open(filename, 'r') as file:
            program = file.read()
    except FileNotFoundError:
        print(f"File '{filename}' not found.")
        return []

    # Tokenize the program
    tokens = tokenize(program)

    return tokens
```

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.

```
# Main function parses the token list and returns the parsed tree
def parse(TokenList):
    global tokens
    tokens = TokenList
    Read("")
    E()

    return Parse_tree
```

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.

```
if Token == Next_Token_Type:
    # If the expected token is an identifier, integer, or string, update the current token.
    # It is track the value of identifier, integer, or string
    if Token in ["<IDENTIFIER>", "<INTEGER>", "<STRING>"]:
        Curr_Token = Next_Token
    # Get the next token from the token list
    Next_Token_Type, Next_Token = tokens.pop(0)
else:
    # Error handling
    if Next_Token_Type in ["<IDENTIFIER>", "<INTEGER>", "<STRING>"]:
        # Check for specific error cases based on the type of expected token
        if Token not in ["<IDENTIFIER>", "<INTEGER>", "<STRING>"]:
            print(
                "Error: Error occurs near ",
                Token,
                ". IDENTIFIER or INTEGER or STRING expected",
            )
        elif Next_Token_Type == "<IDENTIFIER>":
            print("Type Error: IDENTIFIER expected but got ", Token)
        elif Next_Token_Type == "<INTEGER>":
            print("Type Error: INTEGER expected but got ", Token)
        elif Next_Token_Type == "<STRING>":
            print("Type Error: STRING expected but got ", Token)
        else:
            print("Error: Unknown error occurs near ", Token)
    else:
        # Print a generic error message if the expected token type is not an identifier, integer, or string
        print(
            "Error: Missing",
            Token,
            "near",
            Next_Token,
            "or got some unknown Character",
        )
```

The Build_Tree(node_type, num_children) function is responsible for building the parser tree in postorder and appending it to a list.

```
def Build_Tree(node_type, num_children):
    # Check if the node type is an identifier, integer, or string
    if node_type in ["<IDENTIFIER>", "<INTEGER>", "<STRING>"]:
        # If the node type is an identifier, construct the corresponding token
        if node_type == "<IDENTIFIER>":
            Token = "<ID:" + Curr_Token + ">"
        # If the node type is an integer, convert the current token to string and construct the token
        if node_type == "<INTEGER>":
            Token = "<INT:" + str(Curr_Token) + ">"
        # If the node type is a string, construct the corresponding token
        if node_type == "<STRING>":
            Token = "<STR:" + Curr_Token + ">"
        # Append the constructed token to the Parse tree along with the number of children
        Parse_tree.append((Token, num_children))
    else:
        # If the node type is not an identifier, integer, or string, append it to the Parse tree along with the number of children
        Parse_tree.append((node_type, num_children))
```

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.

```
def E():
    # E -> 'let' D 'in' E => 'let'
    if Next_Token_Type == "let":
        Read("let")
        D()
        Read("in")
        E()
        Build_Tree("let", 2)
    # E -> 'fn' Vb+ '.' E => 'lambda'
    elif Next_Token_Type == "fn":
        Read("fn")
        Vb()
        n = 1
        while Next_Token_Type in ["<IDENTIFIER>", "("]:
            Vb()
            n += 1
        Read(".")
        E()
        Build_Tree("lambda", n + 1)
    # E -> Ew
    else:
        Ew()
```

```
def R():
    Rn()  # R -> Rn
    # R ->R Rn => 'gamma'
    while Next_Token_Type in [
        "<IDENTIFIER>",
        "<INTEGER>",
        "<STRING>",
        "true",
        "false",
        "nil",
        "(",
        "dummy",
    ]:
        Rn()
        Build_Tree("gamma", 2)
```

```
def Vl():
    # Vl -> '<IDENTIFIER>' list ',' => ','?
    if Next_Token_Type == "<IDENTIFIER>":
        Read("<IDENTIFIER>")
        Build_Tree("<IDENTIFIER>", 0)
        n = 0
        while Next_Token_Type == ",":
            Read(",")
            Read("<IDENTIFIER>")
            Build_Tree("<IDENTIFIER>", 0)
            n += 1
        if n > 0:
            Build_Tree(",", n + 1)
    else:
        # handle error
        print("Error:error occurs near ", Next_Token, " .IDENTIFIER expected.")
```

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.

```
# Traverse the postorder AST to build the preorder AST
for node in Postorder_AST:
    num_child = node[1]
    node_val = node[0]

    # Add the node value with appropriate indentation to the preorder AST
    Preorder_AST[i] = "." * depth + node_val
    depth += 1
    # Insert None values for the children of the current node
    for j in range(num_child):
        i += 1
        Preorder_AST.insert(i, None)

    # Adjust depth and index if the current node has no children
    if num_child == 0:
        depth -= 1
        i -= 1
        while Preorder_AST[i] != None and i > -1:
            depth -= 1
            i -= 1
return Preorder_AST
```

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.

```
# This function prints the AST using the parse tree
def Print_AST(Preorder_AST):
    for node in Preorder_AST:
        print(node)
```

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.

```python
class ASTNode:
    def __init__(self, type, parent=None):
        self.type = type
        self.children = []
        self.parent = parent

    def add_child(self, child):
        child.parent = self
        self.children.append(child)

    def remove_last_child(self):
        if self.children:
            return self.children.pop()
        else:
            return None
```

```python
def build_tree(ast_list):
    root = ASTNode(ast_list[0])  # Assuming the first element is the root node
    stack = [root]  # Stack to keep track of parent nodes

    for item in ast_list[1:]:
        level = item.count('.')
        node_type = item[level:]

        node = ASTNode(node_type)

        # Pop nodes from stack until the proper parent is found
        while level <= len(stack) - 1:
            stack.pop()

        parent = stack[-1]
        parent.add_child(node)  # Use the add_child method to set the parent of the node
        stack.append(node)

    return root
```

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.

```python
if node.type == 'let':
    #       let              gamma
    #     /     \           /     \
    #    =       P   ->   lambda   E
    #  /   \             /    \
    # X     E           X      P
    equal_node = node.children[0]
    e = equal_node.children[1]
    equal_node.children[1] = node.children[1]
    node.children[1] = e
    equal_node.type = 'lambda'
    node.type = 'gamma'
```

```python
elif node.type == 'where':
    #       where              gamma
    #       /   \             /     \
    #      P     =     ->   lambda   E
    #          /   \        /    \
    #         X     E      X      P
    equal_node = node.children[1]
    node.children[1] = node.children[0]
    node.children[0] = equal_node
    e = equal_node.children[1]
    equal_node.children[1] = node.children[1]
    node.children[1] = e
    equal_node.type = 'lambda'
    node.type = 'gamma'
```

```python
elif node.type == 'function_form':
    #       fcn_form                =
    #       /   |   \              /   \
    #      P    V+   E     ->     P    +lambda
    #                                  /     \
    #                                 V      .E
    equal_node = ASTNode('=')
    equal_node.children.append(node.children[0])
    equal_node.children.append(ASTNode('lambda'))
    node.children.pop(0)
    last_child = node.remove_last_child()
    curr_node = equal_node.children[1]
    while node.children:
        curr_node.add_child(node.children[0])
        node.children.pop(0)
        if node.children:
            curr_node.add_child(ASTNode('lambda'))
        else:
            curr_node.add_child(last_child)
        curr_node = curr_node.children[1]
    node.type = equal_node.type
    node.children = equal_node.children
```

```python
elif node.type == 'within':
    #          within                =
    #          /    \               /    \
    #         =      =     ->     X2     gamma
    #       /   \   /   \                /    \
    #      X1   E1 X2   E2             lambda  E1
    #                                  /    \
    #                                 X1    E2
    e = node.children[0].children[1]
    node.children[0].children[1] = node.children[1].children[1]
    equal_node = ASTNode('=')
    equal_node.add_child(node.children[1].children[0])
    equal_node.add_child(ASTNode('gamma'))
    equal_node.children[1].add_child(node.children[0])
    equal_node.children[1].add_child(e)
    equal_node.children[1].children[0].type = 'lambda'
    node.type = equal_node.type
    node.children = equal_node.children
```

```
elif node.type == '@':
    #         @               gamma
    #       / | \     ->      /   \
    #      E1 N E2          gamma   E2
    #                       /   \
    #                      N     E1
    gamma_node = ASTNode('gamma')
    gamma_node.add_child(ASTNode('gamma'))
    gamma_node.add_child(node.children[2])
    gamma_node.children[0].add_child(node.children[1])
    gamma_node.children[0].add_child(node.children[0])
    node.type = gamma_node.type
    node.children = gamma_node.children
```

```
elif node.type == 'lambda':
    #     lambda        ++lambda
    #      /   \   ->   /    \
    #     V++   E      V     .E
    lambda_node = ASTNode('lambda')
    last_child = node.remove_last_child()
    curr_node = lambda_node
    while node.children:
        curr_node.add_child(node.children[0])
        node.children.pop(0)
        if node.children:
            curr_node.add_child(ASTNode('lambda'))
        else:
            curr_node.add_child(last_child)
        curr_node = curr_node.children[1]
    node.type = lambda_node.type
    node.children = lambda_node.children
```

```
elif node.type == 'and':
    #            and                =
    #             |               /   \
    #            =++      ->      ,   tau
    #           /   \             |    |
    #          X     E           X++  E++
    equal_node = ASTNode('=')
    equal_node.add_child(ASTNode(','))
    equal_node.add_child(ASTNode('tau'))
    for child in node.children:
        equal_node.children[0].add_child(child.children[0])
        equal_node.children[1].add_child(child.children[1])
    node.type = equal_node.type
    node.children = equal_node.children
```

```
elif node.type == 'rec':
    #        rec                =
    #         |               /   \
    #         =      ->       X    gamma
    #        /   \                 /   \
    #       X     E             Ystar  lambda
    #                                  /     \
    #                                 X       E
    equal_node = ASTNode('=')
    equal_node.add_child(node.children[0].children[0])
    equal_node.add_child(ASTNode('gamma'))
    equal_node.children[1].add_child(ASTNode('Ystar'))
    equal_node.children[1].add_child(node.children[0])
    equal_node.children[1].children[1].type = 'lambda'
    node.type = equal_node.type
    node.children = equal_node.children
```

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.

```
def print_tree(root, indent=0):
    print("." * indent + root.type)
    for child in root.children:
        print_tree(child, indent + 1)
```

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

```
# if we come across a lambda node we add an identifier(number) and create a seperate control
# structure for it's children
    if node.type == 'lambda':
        node.type = 'lambda' + str(i)
        i += 1
        control_stack.append(node)
        lambda_stack = []
        preorder_traversal(node, lambda_stack, lambda_stacks)
        lambda_stacks.append(lambda_stack)
```

Beta node

```
# if we have a conditional we replace it with a beta symbol then add two delta nodes for
# the conditions and create seperate structures for those delta nodes. The naming system
#for both delta and lambda nodes act as a pointer system to their control structures.
    elif node.type == '->':
        node.type = 'β'
        control_stack.append(ASTNode('delta' + str(j)))
        control_stack.append(ASTNode('delta' + str(j+1)))
        control_stack.append(node)
        preorder_traversal(node.children[0], control_stack, lambda_stacks)

        lambda_stack = []
        k = j
        j +=2
        lambda_stack.append(ASTNode('delta' + str(k)))
        preorder_traversal(node.children[1], lambda_stack, lambda_stacks)
        lambda_stacks.append(lambda_stack)

        lambda_stack = []
        lambda_stack.append(ASTNode('delta' + str(k+1)))
        preorder_traversal(node.children[2], lambda_stack, lambda_stacks)
        lambda_stacks.append(lambda_stack)
```

Tau node

```
# if it a tau node we add the number of children it has to it's name
    elif node.type == 'tau':
        x=0
        for child in node.children:
            x +=1
        node.type = 'tau' + '[' + str(x) + ']'
        control_stack.append(node)
        for child in node.children:
            preorder_traversal(child, control_stack, lambda_stacks)
```

Token type

```
# we have to remove token types from the names here as they are of no use here on.
    else:
        if node.type.startswith('<ID:'):
            node.type = node.type[4:-1]
        elif node.type.startswith('<INT:'):
            node.type = node.type[5:-1]
        elif node.type.startswith('<STR:'):
            if node.type == "<STR:''>":
                node.type = "''"
            else:
                node.type = node.type[5:-1]
        control_stack.append(node)
        for child in node.children:
            preorder_traversal(child, control_stack, lambda_stacks)
```

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.

```
# we are removing the tree struct from all the nodes in lambda_stacks. now they
# are just strings. then we will add the control structures to env
for i, stack in enumerate(lambda_stacks):
    l_stack = []
    for j, item in enumerate(stack):
        if item.type == ',':
            tau = []
            k = j + 1
            for child in item.children:
                tau.append(stack.pop(k).type)
            l_stack.append(tau)
        else:
            l_stack.append(item.type)
    env.append(l_stack)
```

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

```
# this is a dictionary with the lambda node set as key and the variable set as value
# λ1[x] -> lambda_values[λ1] = x
lambda_values = {}

# Iterate over each lambda stack
for stack in l_stack:
    if stack[0].startswith('lambda'):
        lambda_values[stack[0]] = stack[1]  # Store the lambda node and its value
```

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.

```
# we can remove lambda and the variable from the control structures now as it is stored
# in the previously created dictionary
for stack in l_stack:
    if stack[0].startswith('lambda'):
        stack.pop(0)
        stack.pop(0)

# this sets the way we want the lambda node to look like -> λ_1['x']
for index, value in enumerate(c_stack):
    if value in lambda_values:
        c_stack[index] = f"{value}['{lambda_values[value]}']"
```

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

```
# now we will create the main dictionary for storing all the control structures
result_dict = {'δ_0': c_stack.copy()}

for i, stack in enumerate(l_stack):
    result_dict[f'δ_{i+1}'] = stack.copy()
```

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

```
# Replace occurrences of specific strings with their corresponding symbols
# This was done to look good and easy to read
symbol_map = {'lambda': 'λ_', 'gamma': 'γ', 'delta': 'δ', 'beta': 'β', 'eta': 'η'}
for key, value in result_dict.items():
    for i, item in enumerate(value):
        for word, symbol in symbol_map.items():
            if word in item:
                result_dict[key][i] = item.replace(word, symbol)
```

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

```
# In here we are pairing the delta nodes in case of conditionals
for key, value in result_dict.items():
    for i in value:
        if i == 'β':
            for j in value:
                if j.startswith('δ'):
                    for key1, value1 in result_dict.items():
                        if value1 != value:
                            if j in value1:
                                index = value.index(j)
                                value[index] = key1
```

There is also another function `print_dict(result_dict)` which outputs the control structures created by the above function.

```
# this function prints the dictionary containing the control structures
def print_dict(result_dict):
    for key, value in result_dict.items():
        print(key, end=' : ')
        for i in value:
            print(i, end='  ')
        print()
```

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 will be evaluating the control until it becomes empty
while control:
    # RULE 1 - get the top element from the control
    val = control.pop()

    # RULE 3 - handling stack incase of a gamma node
    if val == 'γ':
        # we take the top value from the stack for evaluation
        rator = stack.pop(0)
        # if rator is a list we need to pop the top value(n) form stack again
        # and get the nth instance from the rator list and add to the stack
        if isinstance(rator, list):
            i = int(stack.pop(0))
            stack.insert(0,rator[i-1])
            continue
```

```
# RULE 12 - defining the functionality of Ystar node
elif rator == 'Ystar':
    var = stack.pop(0)
    eta = 'η' + var[1:]
    stack.insert(0,eta)

# RULE 13 - defining the functionality of eta node
elif rator.startswith('η'):
    stack.insert(0,rator)
    lamda = 'λ' + rator[1:]
    stack.insert(0,lamda)
    control.append('γ')
    control.append('γ')
```

```
# RULE 4 - handling lambda node in the top of stack
if rator.startswith('λ_'):
    num = rator[2]
    var = rator[5:-2]
    rep = stack.pop(0)
    var_list = []
    # this is the case when the lambda node carries multiple variables
    if ',' in var:
        if var.startswith('['):
            var = var[2:-2]
        var_list = var.split("', '")
        for i in range(0,len(var_list)):
            env[var_list[i]] = rep[i]
    else:
        env[var] = rep
        for i in range(0,len(control)):
            if control[i] == var:
                control[i] = rep
    # this represents the change in environment
    control.append('e' + num)
    stack.insert(0, 'e' + num)
    for item in result_dict['δ_' + num]:
        control.append(item)
```

```
# RULE 1 - when moving a variable from control to stack change value if
#          if we have already assigned a value to it.
elif val in env:
    stack.insert(0,env[val])
    if isinstance(env[val], list):
        i = env[val][0]
    else:
        for i in range(0,len(control)):
            if control[i] == val:
                control[i] = env[val]
```

```
# RULE 8 - handling conditional
elif val == 'β':
    rand = stack.pop(0)
    if rand == 'true':
        control.pop()           # this holds the false case
        var = control.pop()     # this holds the true case
    else:
        var = control.pop()     # this holds the false case
        control.pop()           # this holds the true case
    # based on the case we choose we find the control structure relevant
    # to it and add it to the control
    for item in result_dict[var]:
        control.append(item)
```

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.

```
# We will handle the built in functions of RPAL here
elif rator == 'Order':
    val = stack.pop(0)
    if isinstance(val, list):
        num = len(val)
        stack.insert(0,str(num))

elif rator == 'Print':
    val = stack.pop(0)
    # the output format is not the same as what we have in the control structures.
    # they are hanlded here
    formatted_list = str(val).replace('[', '(').replace(']', ')').replace("'", '').replace('"', '')
    print(formatted_list)
```

```
elif rator == 'Isinteger':
    val = stack.pop(0)
    try:
        int(val)
        stack.insert(0,'true')
    except ValueError:
        stack.insert(0,'false')

elif rator == 'Istruthvalue':
    val = stack.pop(0)
    if val in ('true', 'false'):
        stack.insert(0,'true')
    else:
        stack.insert(0,'false')

elif rator == 'Isstring':
    val = stack.pop(0)
    try:
        int(val)
        stack.insert(0,'false')
    except ValueError:
        stack.insert(0,'true')

elif rator == 'Istuple':
    val = stack.pop(0)
    if isinstance(val, list):
        stack.insert(0,'true')
    else:
        stack.insert(0,'false')

elif rator == 'Isfunction':
    val = stack.pop(0)
    if var.startswith('λ_'):
        stack.insert(0,'true')
    else:
        stack.insert(0,'false')

elif rator == 'Isdummy':
    val = stack.pop(0)
    if val == 'dummy':
        stack.insert(0,'true')
    else:
        stack.insert(0,'false')
```

```
# The definitions Stem and Stern are different in different sources.
# we applied the same functionality shown in the RPAL interpreter provided
# Stern removes the first element from a string
elif rator == 'Stern':
    val = stack.pop(0)
    val = val[2:]
    val = "'" + val
    stack.insert(0,val)

# Stem returns the first element of a string
elif rator == 'Stem':
    val = stack.pop(0)
    s = val[:2] + "'"
    stack.insert(0,s)

# Conc concatenates two strings
elif rator == 'Conc':
    val1 = stack.pop(0)
    val2 = stack.pop(0)
    control.pop()
    res = val1[:-1] + val2[1:]
    stack.insert(0,res)
```

```
# this is also a built in function of RPAL
# using this we can add elements to tuples
elif val.startswith('aug'):
    var = stack.pop(0)
    if isinstance(var, list):
        var.append(stack.pop(0))
        stack.insert(0,var)
    # if there is no tuple we create a tuple with only one element
    else:
        tup = [stack.pop(0)]
        stack.insert(0,tup)
```

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.

```
# this method prints the control and the stack in operation as mentioned in the lecture notes.
def print_control_stack(control, stack):
    # Get the width of the terminal window
    terminal_width, _ = shutil.get_terminal_size()

    # Calculate the maximum length of the content (to determine the padding)
    max_length = max(len(' '.join(control)), len(' '.join(str(item) for item in stack)))

    # Calculate the total length of the concatenated strings
    total_length = max_length * 2 + 1  # Total length includes the space between the lists

    # Calculate the number of spaces needed to fill the remaining width
    num_spaces = int(max(terminal_width - total_length, 0) * 0.5)  # Ensure it's non-negative

    # Construct the formatted string with the two lists separated by spaces
    formatted_string = ' '.join(control).ljust(max_length) + ' ' * num_spaces
    + '|' + ' ' * num_spaces + ' '.join(str(item) for item in stack).rjust(max_length)

    print(formatted_string)
    print(terminal_width * '-')
```

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.

```
# Importing lex function from scanner.py
from scanner import lex

# Importing parse function from parser1.py
# Had to name this file parser1.py because there exists a system file named parser.py
from parser1 import parse

# Importing Print_AST function from AST.py
from AST import Print_AST, Build_Preorder_Tree

# Importing standardisation function from ST.py
from ST import standardize, build_tree, print_tree

# Importing create_control_structure function from ControlStructure.py
from ControlStructure import create_control_structure, print_dict

# Importing create_control_structure function from ControlStructure.py
from CSEM import evaluate
```

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('ε0')
        stack.append('ε0')
        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()
```