PROGRAMMING LANGUAGES PROJECT

RPAL Interpreter

Group 126

Gunasekara S.L. 210194H

Kumarasingha H.P.S.M. 210745B

Contents

1 Project Overview

The project was to create an RPAL interpreter from scratch. Our project involves scanning, parsing, and the running of the CSE machine. In the upcoming pages, we will describe how each module of our interpreter works, including the important functions and inputs and outputs of each module.

1.1 Overall Architecture

The python package consists of 4 folders: lexical_analyser, parser, symbols and engine.

The lexical_analyser folder contains all the logic for the lexical analyser, parser for the parser and engine for the CSE Machine. The symbols folder consists of various classes created for each symbol that could be added to the Control Stack or Data Stack.

1.2 Lexical Analyser

The lexical analyzer takes as an input the name of the file containing the RPAL expression and it outputs a list of tokens.

1.2.1 TokenType.py

All this contains is an enum called TokenType. It specifies the various types of tokens that could be found when scanning an RPAL expression.

1.2.2 Token.py

This is simply a file implementing a Token class which has a simple constructor and a function to return the string representation of a token.

1.2.3 LexicalAnalyser.py

This file contains all the scanning logic. The logic is implemented in a class called LexicalAnaylser. It was a constructor and two important functions: scan() and tokenizeLine(line). The scan() function simply calls tokenizeLine(line) on each line of the input file, and the tokenizeLine function has all the tokenizing logic.

scan() function:

```
def scan(self):
    try:
        p = Path(__file__).parent.parent.parent / self.inputFileName
        input_file = open(p, 'r')
        count = 0
        while True:
            count += 1
            line = input_file.readline()

            if not line:
                break

```

```
            self.tokenizeLine(line)

        input_file.close()

    except IOError:
        print("Could not read file {}".format(self.inputFileName))

    filtered_tokens = [
        token for token in self.tokens if token.type != TokenType.DELETE]

    return filtered_tokens
```

tokenizeLine(line) function, which takes as input a a line and adds the tokens to a running list maintained by the LexicalAnaylser object:

```
def tokenizeLine(self, line):
    digit = r"[0-9]"
    letter = r"[a-zA-Z]"
    operatorSymbol = r"[+\-*<>&.@/:=~|$!#%^_\[\]{}\"`?]"
    punction = r"[();,]"

    identifierPattern = re.compile(
        letter+r"("+letter+r"|"+digit+r"|"+r"_)*")
    integerPattern = re.compile(digit+r"+")
    operatorPattern = re.compile(operatorSymbol+r"+")
    stringPattern = re.compile(
        r"'(\t|\n|\\|\"|"+punction+"| |"+letter+"|"+digit+"|"+operatorSymbol+")*'")
    commentPattern = re.compile(r"//.*")
    spacesPattern = re.compile(r"[ \t\n]+")
    punctuationPattern = re.compile(punction)

    currentIndex = 0

    while (currentIndex < len(line)):
        currentChar = line[currentIndex]

        spaceMatch = re.match(spacesPattern, line[currentIndex:])
        commentMatch = re.match(commentPattern, line[currentIndex:])

        if commentMatch:
            comment = commentMatch.group()
            self.tokens.append(Token(TokenType.DELETE, comment))
            currentIndex += len(comment)
            continue

        if spaceMatch:
            space = spaceMatch.group()
            self.tokens.append(Token(TokenType.DELETE, space))
            currentIndex += len(space)
            continue

        match = re.match(identifierPattern, line[currentIndex:])
        if match:
            identifier = match.group()

```

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

            if identifier in keywords:
                self.tokens.append(Token(TokenType.KEYWORD, identifier))
            else:
                self.tokens.append(Token(TokenType.IDENTIFIER, identifier))

            currentIndex += len(identifier)
            continue

        match = re.match(integerPattern, line[currentIndex:])
        if match:
            integer = match.group()
            self.tokens.append(Token(TokenType.INTEGER, integer))
            currentIndex += len(integer)
            continue

        match = re.match(stringPattern, line[currentIndex:])
        if match:
            string = match.group()
            self.tokens.append(Token(TokenType.STRING, string))
            currentIndex += len(string)
            continue

        match = re.match(operatorPattern, line[currentIndex:])
        if match:
            operator = match.group()
            self.tokens.append(Token(TokenType.OPERATOR, operator))
            currentIndex += len(operator)
            continue

        match = re.fullmatch(punctuationPattern, currentChar)
        if match:
            self.tokens.append(Token(TokenType.PUNCTUATION, currentChar))
            currentIndex += 1
            continue

        print("Unable to tokenize the character: {} at index: {}".format(
            currentChar, currentIndex))
        sys.exit()
```

1.3 Parser

This folder contains all the parsing logic. It takes as input the tokens outputted by the Lexical Analyser, and outputs the Abstract Syntax Tree (AST).

1.3.1 NodeType.py

This contains an enum, listing all the possible types of nodes the AST could have.

1.3.2 Node.py

Simply contains the implementation of the Node class with a constructor and a stringifier.

1.3.3 Parser.py

This contains all the parsing logic encapsulated by a class named Parser. It has several functions. The main ones are parse(), and all of the functions responsible for parsing each production. These functions were written based on the RPAL Phrase Structure Grammar. In addition it has certain helper functions.

parse() function:

```
def parse(self):
    # To know when we have gotten the last token
    self.tokens.append(Token(TokenType.ENDOFTOKENS, ""))
    self.E()

    if (self.tokens[0].type == TokenType.ENDOFTOKENS):
        return self.AST

    else:
        print("Parsing unsuccessful...")
        print("Remaining unparsed tokens:")
        for token in self.tokens:
            print(token)
        return None
```

Functions for each production in the RPAL Phrase Grammar Structure

```
E     -> 'let' D 'in' E                          => 'let'
      -> 'fn'  Vb+ '.' E                         => 'lambda'
      ->  Ew;
Ew    -> T  'where' Dr                           => 'where'
      -> T;
```

Figure 1: Expression Productions

```python
def E(self):
    n = 0
    token = self.tokens[0]

    if token.type == TokenType.KEYWORD and token.value in ["let", "fn"]:
        if token.value == "let":
            self.tokens.pop(0)
            self.D()

            if (self.tokens[0].value != "in"):
                print("Parse error at E : 'in' expected")
                sys.exit()

            self.tokens.pop(0)
            self.E()
            self.AST.append(Node(NodeType.let, "let", 2))

        else:
            self.tokens.pop(0)

            while True:
                self.Vb()
                n += 1

                if ((self.tokens[0].type != TokenType.IDENTIFIER) and (self.tokens[0].value != "(")):
                    break

            if not self.tokens[0].value == ".":
                print("Parse error at E : '.' expected")
                sys.exit()

            self.tokens.pop(0)
            self.E()
            self.AST.append(Node(NodeType.lambda_, "lambda", n+1))

    else:
        self.Ew()

def Ew(self):
    self.T()
    if (self.tokens[0].value == "where"):
        self.tokens.pop(0)  # remove the "where"
        self.Dr()
        self.AST.append(Node(NodeType.where, "where", 2))
```

```
T     -> Ta ( ',' Ta )+                        => 'tau'
      -> Ta ;
Ta    -> Ta 'aug' Tc                           => 'aug'
      -> Tc ;
Tc    -> B '->' Tc '|' Tc                      => '->'
      -> B ;
```

Figure 2: Tuple Expression Productions

```python
def T(self):
    self.Ta()
    n = 1

    while (self.tokens[0].value == ","):
        self.tokens.pop(0)  # remove commas
        self.Ta()
        n += 1

    if (n > 1):
        self.AST.append(Node(NodeType.tau, "tau", n))

def Ta(self):
    self.Tc()
    while (self.tokens[0].value == "aug"):
        self.tokens.pop(0)
        self.Tc()
        self.AST.append(Node(NodeType.aug, "aug", 2))

def Tc(self):
    self.B()
    if (self.tokens[0].value == "->"):
        self.tokens.pop(0)  # Remove the '->'
        self.Tc()

        if not self.tokens[0].value == "|":
            print("Parse error at Tc: conditional '|' expected")
            sys.exit()

        self.tokens.pop(0)
        self.Tc()
        self.AST.append(Node(NodeType.conditional, "->", 3))
```

```
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 ;
```

Figure 3: Boolean Expression Productions

```python
def B(self):
    self.Bt()

    while (self.tokens[0].value == "or"):
        self.tokens.pop(0)  # Remove the 'or'
        self.Bt()
        self.AST.append(Node(NodeType.op_or, "or", 2))

def Bt(self):
    self.Bs()

    while (self.tokens[0].value == "&"):
        self.tokens.pop(0)  # Remove the '&'
        self.Bs()
        self.AST.append(Node(NodeType.op_and, "&", 2))

def Bs(self):
    if (self.tokens[0].value == "not"):
        self.tokens.pop(0)
        self.Bp()
        self.AST.append(Node(NodeType.op_not, "not", 1))

    else:
        self.Bp()

def Bp(self):
    self.A()
    token = self.tokens[0]

    if token.value in [">", ">=", "<", "<=", "gr", "ge", "ls", "le", "eq", "ne"]:
        self.tokens.pop(0)
        self.A()

        match token.value:
            case ">":
                self.AST.append(Node(NodeType.op_compare, "gr", 2))
            case ">=":
                self.AST.append(Node(NodeType.op_compare, "ge", 2))
            case "<":
                self.AST.append(Node(NodeType.op_compare, "ls", 2))
            case ">=":
                self.AST.append(Node(NodeType.op_compare, "le", 2))
            case _:
                self.AST.append(Node(NodeType.op_compare, token.value, 2))
```

```
A     -> A '+' At                                  => '+'
      -> A '-' At                                  => '-'
      ->   '+' At
      ->   '-' At                                  => 'neg'
      -> At ;
At    -> At '*' Af                                 => '*'
      -> At '/' Af                                 => '/'
      -> Af ;
Af    -> Ap '**' Af                                => '**'
      -> Ap ;
Ap    -> Ap '@' '<IDENTIFIER>' R                   => '@'
      -> R ;
```

Figure 4: Arithmetic Expression Productions

```python
def A(self):
    if self.tokens[0].value == "+":
        self.tokens.pop(0)
        self.At()

    elif self.tokens[0].value == "-":
        self.tokens.pop(0)
        self.At()
        self.AST.append(Node(NodeType.op_neg, "neg", 1))

    else:
        self.At()

    while self.tokens[0].value in ["+", "-"]:
        currTok = self.tokens[0]
        self.tokens.pop(0)  # Remove the + or - symbols
        self.At()
        if currTok.value == "+":
            self.AST.append(Node(NodeType.op_plus, "+", 2))
        else:
            self.AST.append(Node(NodeType.op_minus, "-", 2))

def At(self):
    self.Af()
    while (self.tokens[0].value in ["*", "/"]):
        currTok = self.tokens[0]
        self.tokens.pop(0)  # Remove the multiply or divide operator
        self.Af()

        if (currTok.value == "*"):
            self.AST.append(Node(NodeType.op_mul, "*", 2))

        else:
            self.AST.append(Node(NodeType.op_div, "/", 2))

def Af(self):
    self.Ap()

    if (self.tokens[0].value == "**"):
        self.tokens.pop(0)
        self.Af()
        self.AST.append(Node(NodeType.op_pow, "**", 2))
```

```python
def Ap(self):
    self.R()

    while self.tokens[0].value == "@":
        self.tokens.pop(0)

        if self.tokens[0].type != TokenType.IDENTIFIER:
            print("Parsing error at Ap: IDENTIFIER expected")
            sys.exit()

        self.AST.append(Node(NodeType.identifier, self.tokens[0].value, 0))
        self.tokens.pop(0)  # Remove IDENTIFIER

        self.R()
        self.AST.append(Node(NodeType.at, "@", 3))
```

```
R     -> R Rn                                   => 'gamma'
      -> Rn ;
Rn    -> '<IDENTIFIER>'
      -> '<INTEGER>'
      -> '<STRING>'
      -> 'true'                                 => 'true'
      -> 'false'                                => 'false'
      -> 'nil'                                  => 'nil'
      -> '(' E ')'
      -> 'dummy'                                => 'dummy' ;
```

Figure 5: Rator and Rand Productions

```python
def R(self):
    self.Rn()

    while ((self.tokens[0].type in
            [TokenType.IDENTIFIER, TokenType.INTEGER, TokenType.STRING])
           or
           (self.tokens[0].value in
            ["true", "false", "dummy", "nil", "("])):
        self.Rn()
        self.AST.append(Node(NodeType.gamma, "gamma", 2))

def Rn(self):
    match self.tokens[0].type:
        case TokenType.IDENTIFIER:
            self.AST.append(
                Node(NodeType.identifier, self.tokens[0].value, 0))
            self.tokens.pop(0)
        case TokenType.INTEGER:
            self.AST.append(
                Node(NodeType.integer, self.tokens[0].value, 0))
            self.tokens.pop(0)
        case TokenType.STRING:
            self.AST.append(Node(NodeType.string, self.tokens[0].value, 0))
            self.tokens.pop(0)
        case TokenType.KEYWORD:
            match self.tokens[0].value:
                case "true":
```

```
                    self.AST.append(Node(NodeType.true_value, "true", 0))
                    self.tokens.pop(0)
                case "false":
                    self.AST.append(Node(NodeType.false_value, "false", 0))
                    self.tokens.pop(0)
                case "nil":
                    self.AST.append(Node(NodeType.nil, "nil", 0))
                    self.tokens.pop(0)
                case "dummy":
                    self.AST.append(Node(NodeType.dummy, "dummy", 0))
                    self.tokens.pop(0)
                case _:
                    print("Parse error at Rn: Unexpected keyword")
                    sys.exit()

        case TokenType.PUNCTUATION:
            if self.tokens[0].value == "(":
                self.tokens.pop(0)  # Remove the opening bracket
                self.E()

                if (self.tokens[0].value != ")"):
                    print("Parsing error at Rn : Expected a matching ')'")
                    sys.exit()

                self.tokens.pop(0)  # Remove closing bracket
        case _:
            print("Parsing error at Rn: Expected an Rn, but got something else")
            sys.exit()
```

```
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 ')' ;
```

Figure 6: Definition Productions

```
def D(self):
    self.Da()
    if (self.tokens[0].value == "within"):
        self.tokens.pop(0)  # Remove 'within'
        self.D()
        self.AST.append(Node(NodeType.within, "within", 2))

def Da(self):
    self.Dr()
    n = 1

    while self.tokens[0].value == "and":
        self.tokens.pop(0)  # Remove 'and's
        self.Dr()
        n += 1
```

```python

    if (n > 1):
        self.AST.append(Node(NodeType.and_, "and", n))

def Dr(self):
    isRec = False

    if self.tokens[0].value == "rec":
        self.tokens.pop(0)  # Remove 'rec' word
        isRec = True

    self.Db()

    if isRec:
        self.AST.append(Node(NodeType.rec, "rec", 1))

def Db(self):
    if (self.tokens[0].type == TokenType.PUNCTUATION) and (self.tokens[0].value == "("):
        self.tokens.pop(0)  # Remove the opening bracket
        self.D()

        if self.tokens[0].value != ")":
            print("Parsing error at Db: Expected matching ')'")
            sys.exit()

        self.tokens.pop(0)

    elif self.tokens[0].type == TokenType.IDENTIFIER:
        # Hoping to get fcn_form
        if (self.tokens[1].value == "(") or (self.tokens[1].type == TokenType.IDENTIFIER):
            self.AST.append(
                Node(NodeType.identifier, self.tokens[0].value, 0))

            self.tokens.pop(0)  # Remove the ID

            n = 1

            while True:
                self.Vb()
                n += 1

                if (self.tokens[0].type != TokenType.IDENTIFIER and self.tokens[0].value != "("):
                    break

            if (self.tokens[0].value != "="):
                print("Parsing error at Db : Expected an '='")
                sys.exit()

            self.tokens.pop(0)
            self.E()

            self.AST.append(Node(NodeType.fcn_form, "fcn_form", n+1))

        elif self.tokens[1].value == "=":
            self.AST.append(
                Node(NodeType.identifier, self.tokens[0].value, 0))
```

```python
            self.tokens.pop(0)  # Remove ID
            self.tokens.pop(0)  # Remove '='
            self.E()

            self.AST.append(Node(NodeType.equal, "=", 2))

        elif self.tokens[1].value == ",":
            self.Vl()
            if (self.tokems[0] != "="):
                print("Parsing error at Db : Expected an '='")
                sys.exit()

            self.tokens.pop(0)
            self.E()

            self.AST.append(Node(NodeType.equal, "=", 2))
```

```
Vb   -> '<IDENTIFIER>'
     -> '(' Vl ')'
     -> '(' ')'                          => '()';
Vl   -> '<IDENTIFIER>' list ','          => ','?;
```

Figure 7: Variable Productions

```python
def Vb(self):
    if self.tokens[0].type == TokenType.PUNCTUATION and self.tokens[0].value == "(":
        self.tokens.pop(0)  # Remove the opening bracket
        isVl = False

        if self.tokens[0].type == TokenType.IDENTIFIER:
            self.Vl()
            isVl = True

        if self.tokens[0].value != ")":
            print("Parse error at Vb : Unmatched '('")
            sys.exit()

        self.tokens.pop(0)

        if isVl:
            self.AST.append(
                Node(NodeType.identifier, self.tokens[0].value, 0))
            self.tokens.pop(0)

        else:
            self.AST.append(Node(NodeType.empty_params, "()", 0))

    elif self.tokens[0].type == TokenType.IDENTIFIER:
        self.AST.append(Node(NodeType.identifier, self.tokens[0].value, 0))
        self.tokens.pop(0)

def Vl(self):
```

```python
    n = 0

    while True:
        if n > 0:
            self.tokens.pop(0)

        if (self.tokens[0].type != TokenType.IDENTIFIER):
            print("Parse error at V1 : an ID was expected")
            sys.exit()

        self.AST.append(Node(NodeType.identifier, self.tokens[0].value, 0))
        self.tokens.pop(0)
        n += 1

        if (self.tokens[0].value != ","):
            break

    if n > 1:
        self.AST.append(Node(NodeType.comma, ",", n))
```

Helper functions: AstToString() and addStrings(dots, node): AstToString is a function to print the AST in a human readable format to the terminal. addStrings is a function used by AstToString to show the depth of a node in the AST.

```python
# Function with the logic to convert AST to a list of strings to show their depth
# Depth is found here and the addString function uses the dots given from here to add to stringAST
def AstToString(self):
    dots = ""
    stack = []

    while self.AST:
        if not stack:
            if self.AST[-1].children == 0:
                self.addStrings(dots, self.AST.pop())

            else:
                node = self.AST.pop()
                stack.append(node)

        else:
            if self.AST[-1].children > 0:
                node = self.AST.pop()
                stack.append(node)
                dots += "."

            else:
                stack.append(self.AST.pop())
                dots += "."

                while (stack[-1].children == 0):
                    self.addStrings(dots, stack.pop())

                    if not stack:
                        break

```

```python
                    dots = dots[:-1]
                    node = stack.pop()
                    node.children -= 1
                    stack.append(node)

    self.stringAST.reverse()
    return self.stringAST

# Function to prepend the dots to a node, and add it to stringAST list
def addStrings(self, dots, node):
    match node.type:
        case NodeType.identifier:
            self.stringAST.append(dots+"<ID:"+node.value+">")
        case NodeType.integer:
            self.stringAST.append(dots+"<INT:"+node.value+">")
        case NodeType.string:
            self.stringAST.append(dots+"<STR:"+node.value+">")
        case NodeType.true_value, NodeType.false_value, NodeType.nil, NodeType.dummy:
            self.stringAST.append(dots+"<"+node.value+">")
        case NodeType.fcn_form:
            self.stringAST.append(dots+"function_form")
        case _:
            self.stringAST.append(dots+node.value)
```

1.4 Symbols

This is just a folder containing all the files in which classes are implemented for each symbol. This was done while implementing the CSE Machine when we realised that using classes we can easily check what symbol is on the stack using isinstance().

1.5 Engine

This folder contains the logic of the CSE Machine. It takes as input to it, the AST generated from the parser, and outputs the result of the RPAL expression. In this section, for most of the classes we have created Factory classes. This was to abstract out the task of construction to a seperate class. That way, when a factory class returns an instance of a class we want, it has already been initialised to its required state.

1.5.1 NodeAndNodeFactory.py

This contains the logic of getting a node given certain arguments, and standardizing a node.

Logic for getting a node: This was implemented using the standardization rules in the lecture slides.

```
def getNode(*args):
    node = Node()
    node.data = args[0]
    node.depth = args[1]

    if len(args) > 2:
        node.parent = args[2]
        node.children = args[3]
        node.isStandardized = args[4]

    return node
```

Function to standardize a node:

```
def standardize(self):
    if not self.isStandardized:
        for child in self.children:
            child.standardize()

        # In accordance with CSE Rules 6 - 11, we don't standardize "tau", "UOp", "BOp", ",", and "->" nodes

        match self.data:
            # Standardizing "let" node
            case "let":
                E = self.children[0].children[1]
                E.parent = self
                E.depth = self.depth + 1
                P = self.children[1]
                P.parent = self.children[0]
                P.depth = self.depth + 2
                self.children[0].data = "lambda"
                self.children[1] = E
                self.children[0].children[1] = P
                self.data = "gamma"

            # Standardizing "where" node
            case "where":
```

```
                # Here, we can simply convert the "where" node into a "let" node and standardize it as above
                P = self.children[0]
                self.children[0] = self.children[1]
                self.children[1] = P

                self.data = "let"
                self.standardize()

            case "function_form":
                E = self.children[len(self.children)-1]
                lambda_node = NodeFactory.getNode(
                    "lambda", self.depth + 1, self, [], True)

                # We can set isStandardized of this node to True, because since
                # we start standardization from the leaf nodes, E will already
                # be standardized.

                self.children.insert(1, lambda_node)

                while self.children[2] != E:
                    V = self.children.pop(2)
                    V.depth = lambda_node.depth + 1
                    V.parent = lambda_node
                    lambda_node.children.append(V)

                    if len(self.children) > 3:
                        lambda_node = NodeFactory.getNode(
                            "lambda", lambda_node.depth + 1, lambda_node, [], True)
                        lambda_node.parent.children.append(lambda_node)

                lambda_node.children.append(E)

                # Remove E from children of the fcn_form
                self.children.pop(2)
                self.data = "="

            case "lambda":
                degree = len(self.children)

                if degree > 2:
                    E = self.children[degree-1]

                    lambda_node = NodeFactory.getNode(
                        "lambda", self.depth + 1, self, [], True)
                    self.children.insert(1, lambda_node)

                    while self.children[2] != E:
                        V = self.children.pop(2)
                        V.depth = lambda_node.depth + 1
                        V.parent = lambda_node
                        lambda_node.children.append(V)

                        if len(self.children) > 3:
                            lambda_node = NodeFactory.getNode(
                                "lambda", lambda_node.depth + 1, lambda_node, [], True)
                            lambda_node.parent.children.append(lambda_node)
```

```
                    lambda_node.children.append(E)

                    # Remove the E from the starting lambda node
                    self.children.pop(2)

            case "within":
                X1 = self.children[0].children[0]
                E1 = self.children[0].children[1]
                X2 = self.children[1].children[0]
                E2 = self.children[1].children[1]

                gamma = NodeFactory.getNode(
                    "gamma", self.depth + 1, self, [], True)
                lambda_node = NodeFactory.getNode(
                    "lambda", self.depth + 2, gamma, [], True)

                X1.depth += 1
                X1.parent = lambda_node

                X2.depth -= 1
                X2.parent = self

                E1.parent = gamma

                E2.depth += 1
                E2.parent = lambda_node

                lambda_node.children.append(X1)
                lambda_node.children.append(E2)

                gamma.children.append(lambda_node)
                gamma.children.append(E1)

                self.children.clear()
                self.children.append(X2)
                self.children.append(gamma)

                self.data = "="

            case "@":
                E1 = self.children[0]
                N = self.children[1]
                E2 = self.children[2]

                bottom_gamma = NodeFactory.getNode(
                    "gamma", self.depth+1, self, [], True)

                E1.depth += 1
                N.depth += 1

                N.parent = bottom_gamma
                E1.parent = bottom_gamma
                E2.parent = self

                bottom_gamma.children.append(N)
```

```
                bottom_gamma.children.append(E1)

                self.children.pop(0)
                self.children.pop(0)
                self.children.insert(0, bottom_gamma)

                self.data = "gamma"

            case "and":
                Xs = []
                Es = []

                comma = NodeFactory.getNode(
                    ",", self.depth + 1, self, [], True)
                tau = NodeFactory.getNode(
                    "tau", self.depth + 1, self, [], True)

                for equal in self.children:
                    # No need to change depths of E's and X's, but we do need to change their parents
                    equal.children[0].parent = comma
                    equal.children[1].parent = tau

                    Xs.append(equal.children[0])
                    Es.append(equal.children[1])

                comma.children = Xs
                tau.children = Es

                self.children.clear()
                self.children.extend([comma, tau])

                self.data = "="

            case "rec":
                X = self.children[0].children[0]
                E = self.children[0].children[1]

                gamma = NodeFactory.getNode(
                    "gamma", self.depth + 1, self, [], True)
                Ystar = NodeFactory.getNode(
                    "<Y*>", self.depth + 2, gamma, [], True)
                lambda_node = NodeFactory.getNode(
                    "lambda", self.depth + 2, gamma, [], True)
                X2 = NodeFactory.getNode(
                    X.data, self.depth + 3, lambda_node, X.children, True)

                X.depth -= 1
                X.parent = self

                E.depth += 1
                E.parent = lambda_node

                lambda_node.children.extend([X2, E])
                gamma.children.extend([Ystar, lambda_node])

                self.children.clear()
```

```python
                self.children.extend([X, gamma])

                self.data = "="

    self.isStandardized = True
```

1.5.2 ASTAndASTFactory.py

This file contains the details of the AST. The ASTFactory has a static method getAST(data). Here, data is the AST that was outputted from the Parser. The getAST function converts this string representation of the AST into an actual abstract data structure, and returns the root node.

getAST(data) function:

```python
@staticmethod
# A function that takes in the string AST, and returns the root of the AST
# Each node in the AST will have a parent attribute as well as a list containing all its children
def getAST(data):
    root = NodeFactory.getNode(data[0], 0)
    prevNode = root
    depth = 0

    for string in data[1:]:
        d = 0

        while string[d] == '.':
            d += 1

        currNode = NodeFactory.getNode(string[d:], d)

        if depth < d:
            prevNode.children.append(currNode)
            currNode.parent = prevNode

        else:
            while prevNode.depth != d:
                prevNode = prevNode.parent

            prevNode.parent.children.append(currNode)
            currNode.parent = prevNode.parent

        prevNode = currNode
        depth = d

    return AST(root)
```

1.5.3 CSEMachineAndCSEMachineFactory.py

There are two classes implemented here: CSEMachineFactory and CSEMachine. All the background work is being done by the CSEMachineFactory class. And the CSEMachine does what it's supposed to do according to the lecture notes: Modify the Control, Stack, and Environment according to the CSE machine rules.

The functions of CSEMachineFactory are :

- **getSymbol(node)**

  This functions returns an instance of the appropriate symbol (based on the node inputted to the function) out of the symbol classes in the aforementioned Symbols folder.

```python
def getSymbol(self, node):
    match node.data:
        case "not" | "neg":
            return UOp(node.data)

        case "+" | "-" | "*" | "/" | "**" | "&" | "or" | "eq" | "ne" | "ls" | "le" | "gr" | "ge" | "aug":
            return BOp(node.data)

        case "gamma":
            return Gamma()

        case "tau":
            return Tau(len(node.children))

        case "<Y*>":
            return YStar()

        case _:
            if node.data.startswith("<ID:"):
                return Id(node.data[4:len(node.data)-1])
            elif node.data.startswith("<INT:"):
                return Int(node.data[5:len(node.data)-1])
            elif node.data.startswith("<STR:"):
                return Str(node.data[6:len(node.data)-2])
            elif node.data.startswith("nil"):
                return Tup()
            # Since we're using Python, we need to capitalize the boolean or else it will cause problems
            elif node.data.startswith("true"):
                return Bool("True")
            elif node.data.startswith("false"):
                return Bool("False")
            elif node.data.startswith("<dummy>"):
                return Dummy()
            else:
                return Err(f"No symbol found for given node: {node.data}")
```

- **getB(node)**

  A 'B' is the part of the conditional operator that evaluates to a truthvalue. This function is used to create the B symbol and initializing its list of symbols with the pre-order traversal of that node.

```
def getB(self, node):
    b = B()
    b.symbols = self.getPreOrderTraversal(node)

    return b
```

- **getDelta(node)**

  This function creates a Delta symbol and initializes its list of symbols to the pre-order traversal of the delta node.

```
def getDelta(self, node):
    delta = Delta(self.j)
    self.j += 1
    delta.symbols = self.getPreOrderTraversal(node)

    return delta
```

- **getLambda(node)**

  This creates a Lambda symbol. The lambda symbol has an attribute called delta, which is the preorder traversal of the right child of the lambda. This connection is made within this function.

```
def getLambda(self, node):
    lambda_ = Lambda(self.i)
    self.i += 1
    lambda_.delta = self.getDelta(node.children[1])

    if node.children[0].data == ",":
        for identifier in node.children[0].children:
            lambda_.identifiers.append(
                Id(identifier.data[4:len(identifier.data)-1]))
    else:
        bounded_var = node.children[0]
        lambda_.identifiers.append(
            Id(bounded_var.data[4:len(bounded_var.data)-1]))

    return lambda_
```

- **getPreOrderTraversal(node)**

```
def getPreOrderTraversal(self, node):
    symbols = []
    if node.data == "lambda":
        symbols.append(self.getLambda(node))

    elif node.data == "->":
        symbols.append(self.getDelta(node.children[1]))  # delta then
        symbols.append(self.getDelta(node.children[2]))  # delta else
        symbols.append(Beta())
```

```
        symbols.append(self.getB(node.children[0]))

    else:
        symbols.append(self.getSymbol(node))
        for child in node.children:
            symbols.extend(self.getPreOrderTraversal(child))

    return symbols
```

- **getControl(ast)**

  Sets control to its initial state (containing only e0 and delta1) and returns control.

```
def getControl(self, ast):
    control = []
    control.append(self.e0)
    control.append(self.getDelta(ast.root))

    return control
```

- **getStack()**

  Sets stack to its initial state (containing only e0) and returns it.

```
def getStack(self):
    stack = []
    stack.append(self.e0)

    return stack
```

- **getEnv()**

  Returns a list, containing only e0, other envs to be added as the CSE machine executes.

```
def getEnv(self):
    return [self.e0]
```

- **getCSEMachine(ast)**

  Initializes and returns a CSE machine with it's control, stack and env set to the proper initial state.

```
def getCSEMachine(self, ast):
    return CSEMachine(self.getControl(ast), self.getStack(), self.getEnv())
```

The functions of CSEMachine are :

- **execute()**

  This is the main function, and it encapsulates all the logic of the CSE machine. If this functions successfully finishes execution, you will have the control empty and the stack containing the result of the RPAL expression.

```
def execute(self):
    currEnv = self.env[0]
    j = 1

    while self.control:
        # print("\nControl stack: ")
        # self.printControl()

        # print("\nData stack: ")
        # self.printStack()

        # Pop the control
        currentSymbol = self.control.pop()

        # CSE Rule 1
        if isinstance(currentSymbol, Id):
            Ob = currEnv.lookup(currentSymbol)
            self.stack.insert(0, Ob)

        # CSE Rule 2
        elif isinstance(currentSymbol, Lambda):
            lambda_ = currentSymbol
            lambda_.environment = currEnv.index
            self.stack.insert(0, lambda_)

        elif isinstance(currentSymbol, Gamma):
            # Get stack-top
            stackTop = self.stack.pop(0)
            # CSE Rule 4
            if isinstance(stackTop, Lambda):
                lambda_ = stackTop
                e = E(j)
                j += 1

                if (len(lambda_.identifiers) == 1):
                    e.values[lambda_.identifiers[0]] = self.stack.pop(0)

                # CSE Rule 11
                else:
                    tup = self.stack.pop(0)

                    i = 0
                    for id in lambda_.identifiers:
                        e.values[id] = tup.symbols[i]
                        i += 1

                for env in self.env:
                    if env.index == lambda_.environment:
```

```
                        e.parent = env

                currEnv = e
                self.control.append(currEnv)
                self.control.append(lambda_.delta)
                self.stack.insert(0, currEnv)
                self.env.append(currEnv)

            # CSE Rule 10
            elif isinstance(stackTop, Tup):
                tup = stackTop
                index = int(self.stack.pop(0).data)

                # "index - 1" because Python lists are zero-indexed but RPAL lists are 1-indexed
                tupleValue = stackTop.symbols[index-1]
                self.stack.insert(0, tupleValue)

            # CSE Rule 12
            elif isinstance(stackTop, YStar):
                lambda_ = self.stack.pop(0)
                eta = Eta()
                eta.index = lambda_.index
                eta.environment = lambda_.environment
                eta.identifier = lambda_.identifiers[0]
                eta.lambda_ = lambda_

                self.stack.insert(0, eta)

            # CSE Rule 13
            elif isinstance(stackTop, Eta):
                eta = stackTop
                self.control.append(Gamma())
                self.control.append(Gamma())

                lambda_ = eta.lambda_

                self.stack.insert(0, eta)
                self.stack.insert(0, lambda_)

            # Built-in functions
            else:
                builtInFunction = stackTop.data

                match builtInFunction:
                    case "Print":
                        thingToBePrinted = self.stack.pop(0)
                        if not isinstance(thingToBePrinted, Tup):
                            print(thingToBePrinted.data)
                        else:
                            print(self.getStringTuple(thingToBePrinted))

                        self.stack.insert(0, Dummy())

                    case "Stem":
                        stringToBeStemmed: Str = copy.deepcopy(
                            self.stack.pop(0))
```

```python
    stringToBeStemmed.data = stringToBeStemmed.data[0]
    self.stack.insert(0, stringToBeStemmed)

case "Stern":
    stringToBeSterned: Str = copy.deepcopy(
        self.stack.pop(0))

    stringToBeSterned.data = stringToBeSterned.data[1:]
    self.stack.insert(0, stringToBeSterned)

case "Conc":
    # The correct way for Conc A B to happen is
    # Contol                        Stack           Env
    # gamma gamma Conc A B
    # gamma gamma                   Conc A B
    # gamma                         ConcA B
    #                               <result of A+B>

    # Therefore, we need to pop the second gamma from the control too
    self.control.pop()

    str1 = self.stack.pop(0)
    str2 = self.stack.pop(0)

    resultantString = copy.deepcopy(str1)

    resultantString.data += str2.data
    self.stack.insert(0, resultantString)

case "Order":
    tup: Tup = self.stack.pop(0)

    size = len(tup.symbols)
    self.stack.insert(0, Int(str(size)))

case "Null":
    tup: Tup = self.stack.pop(0)
    result = True

    if tup.symbols:
        result = False

    self.stack.insert(0, Bool(str(result)))

case "Isinteger":
    possibleInteger = self.stack.pop(0)

    if isinstance(possibleInteger, Int):
        self.stack.insert(0, Bool("True"))
    else:
        self.stack.insert(0, Bool("False"))

case "Isstring":
    possibleString = self.stack.pop(0)

```

```
                    if isinstance(possibleString, Str):
                        self.stack.insert(0, Bool("True"))
                    else:
                        self.stack.insert(0, Bool("False"))

                case "Istuple":
                    possibleTuple = self.stack.pop(0)

                    if isinstance(possibleTuple, Tup):
                        self.stack.insert(0, Bool("True"))
                    else:
                        self.stack.insert(0, Bool("False"))

                case "Isdummy":
                    possibleDummy = self.stack.pop(0)

                    if isinstance(possibleDummy, Dummy):
                        self.stack.insert(0, Bool("True"))
                    else:
                        self.stack.insert(0, Bool("False"))

                case "Istruthvalue":
                    possibleTruthvalue = self.stack.pop(0)

                    if isinstance(possibleTruthvalue, Bool):
                        self.stack.insert(0, Bool("True"))
                    else:
                        self.stack.insert(0, Bool("False"))

                case "Isfunction":
                    possibleLambda = self.stack.pop(0)

                    if isinstance(possibleLambda, Lambda):
                        self.stack.insert(0, Bool("True"))
                    else:
                        self.stack.insert(0, Bool("False"))

        # CSE Rule 5
        elif isinstance(currentSymbol, E):
            value = self.stack.pop(0)
            env = self.stack.pop(0)

            self.stack.insert(0, value)

            self.env[currentSymbol.index].isRemoved = True

            y = len(self.env)

            # Traverse list of envs in reverse order to find the new current env
            while y > 0:
                if not self.env[y-1].isRemoved:
                    currEnv = self.env[y-1]
                    break
                y -= 1

        elif isinstance(currentSymbol, Rator):
```

```
            rator = currentSymbol
            # CSE Rule 6
            if isinstance(currentSymbol, BOp):
                rand1 = self.stack.pop(0)
                rand2 = self.stack.pop(0)

                result = self.applyBOp(rator, rand1, rand2)

            # CSE Rule 7
            elif isinstance(currentSymbol, UOp):
                rand = self.stack.pop(0)

                result = self.applyUOp(rator, rand)

            self.stack.insert(0, result)

        # CSE Rule 8
        elif isinstance(currentSymbol, Beta):
            boolOnStack = self.stack.pop(0)

            del_else = self.control.pop()
            del_then = self.control.pop()

            if (eval(boolOnStack.data)):
                self.control.append(del_then)

            else:
                self.control.append(del_else)

        # CSE Rule 9
        elif isinstance(currentSymbol, Tau):
            tup = Tup()
            for _ in range(currentSymbol.n):
                tup.symbols.append(self.stack.pop(0))

            self.stack.insert(0, tup)

        # Encountering delta (delta-then or delta-else)
        elif isinstance(currentSymbol, Delta):
            self.control.extend(currentSymbol.symbols)

        elif isinstance(currentSymbol, B):
            self.control.extend(currentSymbol.symbols)

        # Int
        else:
            self.stack.insert(0, currentSymbol)
```

- **applyUOp and applyBOp**

  Functions to apply unary and binary operators on their respective operands. This functions is a helper function for execute()

```
def applyUOp(self, rator, rand):
    if rator.data == "neg":
        return Int(str(-1 * int(rand.data)))

    elif rator.data == "not":
        return Bool(str(not eval(rand.data)))

    else:
        return Err("Unknown unary operator encountered!")

def applyBOp(self, rator, rand1, rand2):
    if rator.data == "+":
        return Int(str(int(rand1.data) + int(rand2.data)))

    elif rator.data == "-":
        return Int(str(int(rand1.data) - int(rand2.data)))

    elif rator.data == "*":
        return Int(str(int(rand1.data) * int(rand2.data)))

    elif rator.data == "/":
        return Int(str(int(rand1.data) / int(rand2.data)))

    elif rator.data == "**":
        return Int(str(int(rand1.data) ** int(rand2.data)))

    elif rator.data == "&":
        return Bool(str(eval(rand1.data) and eval(rand2.data)))

    elif rator.data == "or":
        return Bool(str(eval(rand1.data) or eval(rand2.data)))

    elif rator.data == "eq":
        return Bool(str(rand1.data == rand2.data))

    elif rator.data == "ne":
        return Bool(str(rand1.data != rand2.data))

    elif rator.data == "ls":
        return Bool(str(int(rand1.data) < int(rand2.data)))

    elif rator.data == "le":
        return Bool(str(int(rand1.data) <= int(rand2.data)))

    elif rator.data == "gr":
        return Bool(str(int(rand1.data) > int(rand2.data)))

    elif rator.data == "ge":
        return Bool(str(int(rand1.data) >= int(rand2.data)))

    elif rator.data == "aug":
```

```python
        if not isinstance(rand1, Tup):
            return Err("'aug' operator expects either tuple or nil")

        if isinstance(rand2, Tup):
            rand1.symbols.extend(rand2.symbols)
        else:
            rand1.symbols.append(rand2)

        return rand1

    else:
        return Err("Unknown binary operator encountered!")
```

- **getStringTuple(tup)**

  This functions returns the string representation of a tuple. This is done in case the RPAL program requires a tuple to be printed, or the value of the RPAL program is a tuple.

```python
def getStringTuple(self, tup: Tup):
    result = "("

    for symbol in tup.symbols:
        if isinstance(symbol, Tup):
            result += self.getStringTuple(symbol) + ", "

        else:
            # We need to do the following because in RPAL, truthvalues are in lowercase, but in Python, the first
            # letter is capitalized.
            data = symbol.data.lower() if isinstance(symbol, Bool) else symbol.data
            result += data + ", "

    # Remove the ', ' from the last tuple element
    result = result[0:len(result)-2] + ")"
    return result
```

- **getResult()**

  This function runs the execute method and returns the answer (stack-top after execution).

```python
def getResult(self):
    self.execute()
    answer = self.stack.pop(0)

    if (isinstance(answer, Tup)):
        return self.getStringTuple(answer)

    # We need to do the following because in RPAL, truthvalues are in lowercase, but in Python, the first
    # letter is capitalized.

    return answer.data.lower() if isinstance(answer, Bool) else answer.data
```

2 How to run

2.1 Prerequisites

1. You need to have Python3 installed on your machine.

2.2 Steps

1. Extract the contents of the zipped folder onto your machine.

2. Either copy the input file containing the RPAL expression into the folder, or enter the program into the "input_file.txt" file. The file containing the RPAL program must be directly under the 210194H210745B folder.

3. You can either run the python file directly from the command line, or you can use a make command.

   (a) To use the terminal :

      i. Open a terminal in the 210194H210745B directory, and run the following command :

      ```
      python ./myrpal.py <name of file containing RPAL expression>
      ```

      This will run the RPAL program. If a Print is called in the RPAL program, the printed value will be displayed on the console.

      ii. If you want to print the AST (Abstract Syntax Tree), you need to add a '-ast' flag to the above command. Keep in mind that this will only print the AST for the program, and the program will not actually execute. The new terminal command will look like :

      ```
      python ./myrpal.py <name of file containing RPAL expression> -ast
      ```

   (b) To use the make command :

      i. Open a terminal in the 210194H210745B directory, and run the following command :

      ```
      make run filename="<whatever_the_file_name_is>.txt"
      ```

      This will run the RPAL program. If a Print is called in the RPAL program, the printed value will be displayed on the console.

      ii. If you want to print the AST (Abstract Syntax Tree), you need to use a different target: ast The new terminal command will look like :

      ```
      make ast filename="<whatever_the_file_name_is>.txt"
      ```