|  |  |  |  |
| --- | --- | --- | --- |
| فاز اول پروژه ی درس کامپایلر  استاد درس: دکتر رزازی  زهرا نژادیان 9433105- سپیده ملانوروزی 9431072 | |  |  |
| **تاریخ تحویل : دوشنبه 20 آبان – موعد تحویل : دوشنبه 20 ابان** | | | |

**Input :**

class Program{

int my\_v\_ar, integerId = 4;

static void \_main(){

my\_v\_ar = -23 - 22;

print("N

Sanity.");

for(integerId in 0 to 100 steps 10){

my\_v\_ar = integerId << 2;

}

}

// 2nd function

bool \_TorF(){

real var\_2 = 0.14;

if(my\_v\_ar < 8 && 32 - my\_v\_ar > var\_2){

my\_v\_ar = my\_v\_ar | (integerId - 2);

my\_v\_ar = my\_v\_ar & 0b10010010;

}else if(var\_2 != 0.0){

while(true){

var = 3;

break;

}

return true;

}

return false;

}

real \_func(int i, int j){

return i % j \* 3.52;

}

}

**Output :**

Lexeme: class Token: CLASS Attribute: -

Lexeme: Program Token: ID Attribute: 0

Lexeme: { Token: LCB Attribute: -

Lexeme: int Token: INT\_TYPE Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: , Token: COMMA Attribute: -

Lexeme: integerId Token: ID Attribute: 2

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: 4 Token: INTEGER Attribute: 4

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: static Token: SATATIC Attribute: -

Lexeme: void Token: VOID Attribute: -

Lexeme: \_main Token: ID Attribute: 3

Lexeme: ( Token: LP Attribute: -

Lexeme: ) Token: RP Attribute: -

Lexeme: { Token: LCB Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: - Token: SUBTRACTION Attribute: -

Lexeme: 23 Token: INTEGER Attribute: 23

Lexeme: - Token: SUBTRACTION Attribute: -

Lexeme: 22 Token: INTEGER Attribute: 22

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: print Token: PRINT Attribute: -

Lexeme: ( Token: LP Attribute: -

Lexeme: "N

Sanity." Token: STRING Attribute: "N

Sanity."

Lexeme: ) Token: RP Attribute: -

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: for Token: FOR Attribute: -

Lexeme: ( Token: LP Attribute: -

Lexeme: integerId Token: ID Attribute: 2

Lexeme: in Token: IN Attribute: -

Lexeme: 0 Token: INTEGER Attribute: 0

Lexeme: to Token: TO Attribute: -

THIS IS SYMBOL TABLE

Program 0

my\_v\_ar 1

integerId 2

\_main 3

\_TorF 4

var\_2 5

var 6

\_func 7

i 8

j 9

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: | Token: BITWISE\_OR Attribute: -

Lexeme: ( Token: LP Attribute: -

Lexeme: integerId Token: ID Attribute: 2

Lexeme: - Token: SUBTRACTION Attribute: -

Lexeme: 2 Token: INTEGER Attribute: 2

Lexeme: ) Token: RP Attribute: -

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: & Token: BITWISE\_AND Attribute: -

Lexeme: 0b10010010 Token: INTEGER Attribute: 146

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: } Token: RCB Attribute: -

Lexeme: else Token: ELSE Attribute: -

Lexeme: if Token: IF Attribute: -

Lexeme: ( Token: LP Attribute: -

Lexeme: var\_2 Token: ID Attribute: 5

Lexeme: ! Token: NOT Attribute: -

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: 0.0 Token: REAL Attribute: -

Lexeme: ) Token: RP Attribute: -

Lexeme: { Token: LCB Attribute: -

Lexeme: while Token: WHILE Attribute: -

Lexeme: ( Token: LP Attribute: -

Lexeme: true Token: TRUE Attribute: -

Lexeme: ) Token: RP Attribute: -

Lexeme: { Token: LCB Attribute: -

Lexeme: var Token: ID Attribute: 6

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: 3 Token: INTEGER Attribute: 3

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: break Token: BREAK Attribute: -

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: } Token: RCB Attribute: -

Lexeme: return Token: RETURN Attribute: -

Lexeme: 100 Token: INTEGER Attribute: 100

Lexeme: steps Token: STEPS Attribute: -

Lexeme: 10 Token: INTEGER Attribute: 10

Lexeme: ) Token: RP Attribute: -

Lexeme: { Token: LCB Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: integerId Token: ID Attribute: 2

Lexeme: << Token: SHIFT\_LEFT Attribute: -

Lexeme: 2 Token: INTEGER Attribute: 2

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: } Token: RCB Attribute: -

Lexeme: } Token: RCB Attribute: -

\*\*\*COMMENT\*\*\* Lexeme: // 2nd function Token: - Attribute: -

Lexeme: bool Token: BOOL\_TYPE Attribute: -

Lexeme: \_TorF Token: ID Attribute: 4

Lexeme: ( Token: LP Attribute: -

Lexeme: ) Token: RP Attribute: -

Lexeme: { Token: LCB Attribute: -

Lexeme: real Token: REAL\_TYPE Attribute: -

Lexeme: var\_2 Token: ID Attribute: 5

Lexeme: = Token: ASSIGNMENT Attribute: -

Lexeme: 0.14 Token: REAL Attribute: -

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: if Token: IF Attribute: -

Lexeme: ( Token: LP Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: < Token: LT Attribute: -

Lexeme: 8 Token: INTEGER Attribute: 8

Lexeme: && Token: AND Attribute: -

Lexeme: 32 Token: INTEGER Attribute: 32

Lexeme: - Token: SUBTRACTION Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: > Token: GT Attribute: -

Lexeme: var\_2 Token: ID Attribute: 5

Lexeme: ) Token: RP Attribute: -

Lexeme: { Token: LCB Attribute: -

Lexeme: my\_v\_ar Token: ID Attribute: 1

Lexeme: true Token: TRUE Attribute: -

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: } Token: RCB Attribute: -

Lexeme: return Token: RETURN Attribute: -

Lexeme: false Token: FALSE Attribute: -

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: } Token: RCB Attribute: -

Lexeme: real Token: REAL\_TYPE Attribute: -

Lexeme: \_func Token: ID Attribute: 7

Lexeme: ( Token: LP Attribute: -

Lexeme: int Token: INT\_TYPE Attribute: -

Lexeme: i Token: ID Attribute: 8

Lexeme: , Token: COMMA Attribute: -

Lexeme: int Token: INT\_TYPE Attribute: -

Lexeme: j Token: ID Attribute: 9

Lexeme: ) Token: RP Attribute: -

Lexeme: { Token: LCB Attribute: -

Lexeme: return Token: RETURN Attribute: -

Lexeme: i Token: ID Attribute: 8

Lexeme: % Token: MODULO Attribute: -

Lexeme: j Token: ID Attribute: 9

Lexeme: \* Token: MULTIPLICATION Attribute: -

Lexeme: 3.52 Token: REAL Attribute: -

Lexeme: ; Token: SEMICOLON Attribute: -

Lexeme: } Token: RCB Attribute: -

Lexeme: } Token: RCB Attribute: -

**Code :**

import ply.lex as lex

import re

fL = open("output.txt", "w")

symbol\_table = {}

tokens = [ 'NUMERROR','WHITESPACE', 'INTEGER', 'ID','REAL', 'STRING', 'COMMENT', 'CLASS', 'REFERENCE', 'SATATIC',

'INT\_TYPE', 'REAL\_TYPE', 'BOOL\_TYPE', 'STRING\_TYPE', 'VOID', 'TRUE', 'FALSE', 'PRINT', 'RETURN', 'BREAK', 'CONTINUE', 'IF', 'ELSE',

'ELSeIF','WHILE', 'FOR', 'TO', 'IN', 'STEPS', 'BITWISE\_AND', 'AND', 'BITWISE\_OR', 'OR', 'NOT', 'BITWISE\_NOT', 'SHIFT\_RIGHT',

'SHIFT\_LEFT', 'ASSIGNMENT', 'ADDITION', 'SUBTRACTION', 'MULTIPLICATION', 'DIVISION', 'MODULO', 'POWER', 'GT', 'GE', 'LT',

'LE', 'EQ', 'NE', 'LCB' , 'RCB', 'LP', 'RP', 'DOT', 'SEMICOLON', 'COMMA', 'TOKENERROR']

def t\_WHITESPACE(t):

r"""\s+"""

def t\_NUMERROR(t):

r"""([0-9]+[ac-wyzAC-WYZ][a-zA-Z]\*)|(0[0-9]+\.[0-9]\*[1-9])|([1-9][0-9]\*\.[0-9]+0)|(0+x+0+[0-9a-fA-F]+)|(0+b+0+[01]+)|(00+[0-9]\*)"""

txt ="\*\*\*ERROR\*\*\* "+ "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "ERROR" +"\t"+ " Attribute: "+"\t"+ "-" + "\n\n"

fL.write(txt)

def t\_REAL(t):

r"""([1-9][0-9]\*\.[0-9]\*[1-9])|(0\.[0-9]\*[1-9])|([1-9][0-9]\*\.0)|(0\.0)"""

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "REAL" +"\t"+ " Attribute: " +"\t"+ "-" + "\n\n"

fL.write(txt)

return t

def t\_INTEGER(t):

r"""(0x[1-9a-fA-F][0-9a-fA-F]\*)|(0x0)|(0b1[01]\*)|(0b0)|([1-9][0-9]\*)|(0)"""

if '0x' in t.value:

dec = int(t.value, 16)

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "INTEGER" +"\t"+ " Attribute: " +"\t"+ str(dec) + "\n\n"

fL.write(txt)

elif '0b' in t.value:

dec = int(t.value, 2)

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "INTEGER" +"\t"+ " Attribute: " +"\t"+ str(dec) + "\n\n"

fL.write(txt)

else:

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "INTEGER" +"\t"+ " Attribute: " +"\t"+ t.value + "\n\n"

fL.write(txt)

return t

def t\_COMMENT(t):

r'(\/\/[^\n]\*)|(/\\*([^\*]|[\r\n]|(\\*+([^\*/]|[\r\n])))\*\\*+/)'

txt = "\*\*\*COMMENT\*\*\* "+ "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "-" +"\t"+ " Attribute: " +"\t"+ "-" + "\n\n"

fL.write(txt)

# print("comment: "+t.value)

def t\_CLASS(t):

r'class[ ]'

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "CLASS" +"\t"+ " Attribute: " +"\t"+ "-" + "\n\n"

fL.write(txt)

# print("this is class type : "+t.value)

return t

def t\_STRING(t):

r"""(([\"](.\*?)[\"]\s\*\+\s\*)\*(\s\*[\"](.\*?)[\"]))|([\"](.\*?)[\"])|([\"](.\*?)\s\*(.\*?)[\"])"""

list\_of\_tokens = t.value.split('"')

new\_list = []

indexes = []

for a in list\_of\_tokens:

if '+' in a :

indexes.append(list\_of\_tokens.index(a))

replaced = re.sub('[ \t\n\r\f\v]', '', a)

new\_list.append(replaced)

list\_operands = []

flag = 0

for i in range(0, len(new\_list)):

if new\_list[i] == '+':

flag = 1

index = indexes[i]

if index - 1 == 0 or index+1 == len(list\_of\_tokens)-1:

flag = 0

break

list\_operands.append(list\_of\_tokens[index-1])

if i == len(new\_list)-1:

list\_operands.append(list\_of\_tokens[index+1])

if flag == 1:

s = ''.join(list\_operands)

else:

s = t.value

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "STRING" +"\t"+ " Attribute: " +"\t"+ s + "\n\n"

fL.write(txt)

return t

def t\_REFERENCE(t):

r'reference'

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "REFERENCE" +"\t"+ " Attribute: " +"\t"+ "-" + "\n\n"

fL.write(txt)

# print("this is reference type : "+t.value)

return t

def t\_SATATIC(t):

r'static[ ]'

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "SATATIC" +"\t"+ " Attribute: " +"\t"+ "-" + "\n\n"

fL.write(txt)

return t

def t\_INT\_TYPE(t):

r'int[ ]'

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "INT\_TYPE" +"\t"+ " Attribute: "+"\t" + "-" + "\n\n"

fL.write(txt)

return t

def t\_REAL\_TYPE(t):

r'real[ ]'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "REAL\_TYPE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_BOOL\_TYPE(t):

r'bool[ ]'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "BOOL\_TYPE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_VOID(t):

r'void'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "VOID" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_TRUE(t):

r'true'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "TRUE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_FALSE(t):

r'false'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "FALSE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_PRINT(t):

r'print'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "PRINT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_RETURN(t):

r'return[ ]'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "RETURN" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_STRING\_TYPE(t):

r'string[ ]'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "STRING\_TYPE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_BREAK(t):

r'break'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "BREAK" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_CONTINUE(t):

r'continue'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "CONTINUE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_IF(t):

r'if'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "IF" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_ELSE(t):

r'else'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "ELSE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_ELSEIF(t):

r'elseif'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "ELSEIF" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_WHILE(t):

r'while'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "WHILE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_FOR(t):

r'for'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "FOR" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_TO(t):

r'to[ ]'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "TO" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_IN(t):

r'in[ ]'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "IN" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_STEPS(t):

r'steps[ ]'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "STEPS" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_AND(t):

r'&&'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "AND" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_ID(t):

r"""([a-zA-Z]\w\*)|([a-zA-Z]\w\*\\_)|(\\_\w\*)|([a-zA-Z]\w\*[\\_\w]+\\_\w\*)|([\\_\w]+\\_\w+)"""

l = len(t.value)

if l%2 == 0:

txt = "\*\*\*ERROR\*\*\* "+ "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "ERROR" +"\t"+ " Attribute: " +"\t"+ "-" + "\n\n"

fL.write(txt)

else:

replaced = re.sub('[ \t\n\r\f\v]', '', t.value)

if replaced in symbol\_table:

attribute = symbol\_table[replaced]

else:

num = len(symbol\_table)

symbol\_table[replaced] = num

attribute = num

txt = "Lexeme: "+"\t" + t.value +"\t"+ " Token: " +"\t"+ "ID" +"\t"+ " Attribute: " +"\t"+ str(attribute) + "\n\n"

fL.write(txt)

return t

def t\_BITWISE\_AND(t):

r'&'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "BITWISE\_AND" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_OR(t):

r'\|\|'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "OR" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_NOT(t):

r'!'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "NOT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_BITWISE\_OR(t):

r'\|'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "BITWISE\_OR" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n

fL.write(txt)

return t

def t\_BITWISE\_NOT(t):

r'~'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "BITWISE\_NOT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_SHIFT\_RIGHT(t):

r'>>'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "SHIFT\_RIGHT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_SHIFT\_LEFT(t):

r'<<'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "SHIFT\_LEFT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_ADDITION(t):

r'\+'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "ADDITION" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_SUBTRACTION(t):

r'-'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "SUBTRACTION" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_MULTIPLICATION(t):

r'\\*'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "MULTIPLICATION" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_DIVISION(t):

r'\/'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "DIVISION" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_MODULO(t):

r'%'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "MODULO" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_POWER(t):

r'\^'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "POWER" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_GE(t):

r'>='

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "GE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_LE(t):

r'<='

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "LE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_GT(t):

r'>'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "GT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_LT(t):

r'<'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "LT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_EQ(t):

r'=='

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "EQ" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_NE(t):

r'!='

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "NE" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_ASSIGNMENT(t):

r'='

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "ASSIGNMENT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_LCB(t):

r'{'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "LCB" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_RCB(t):

r'}'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "RCB" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_LP(t):

r'\('

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "LP" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_RP(t):

r'\)'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "RP" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_DOT(t):

r'\.'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "DOT" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_SEMICOLON(t):

r';'

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "SEMICOLON" +"\t"+ " Attribute: "+"\t"+ "-"+"\n\n"

fL.write(txt)

return t

def t\_COMMA(t):

r','

txt = "Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "COMMA" +"\t"+ " Attribute: "+"\t"+ "-" +"\n\n"

fL.write(txt)

return t

def t\_TOKENERROR(t):

r"""(\s\*(.+?)+\s\*)|(\s\*[^\"]+\s\*)"""

txt = "\*\*\*ERROR\*\*\* "+"Lexeme: "+"\t"+ t.value +"\t"+" Token: "+"\t"+ "-" +"\t"+ " Attribute: "+"\t"+ "-" +"\n\n"

fL.write(txt)

lexer = lex.lex()

path = "mainInput.txt"

f = open(path, 'r')

text = f.read()

f.close()

lexer.input(text)

while True:

tok = lex.token()

if not tok:

txt = "THIS IS SYMBOL TABLE \n\n"

for a in symbol\_table:

txt += a + "\t" + str(symbol\_table[a]) + "\n\n"

fL.write(txt)

fL.close()

break