دانشکده مهندسی کامپیوتر

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

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 :

```
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:   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: -
```

```
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:  =           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:   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
```