System Software Practicals

Assignment 4

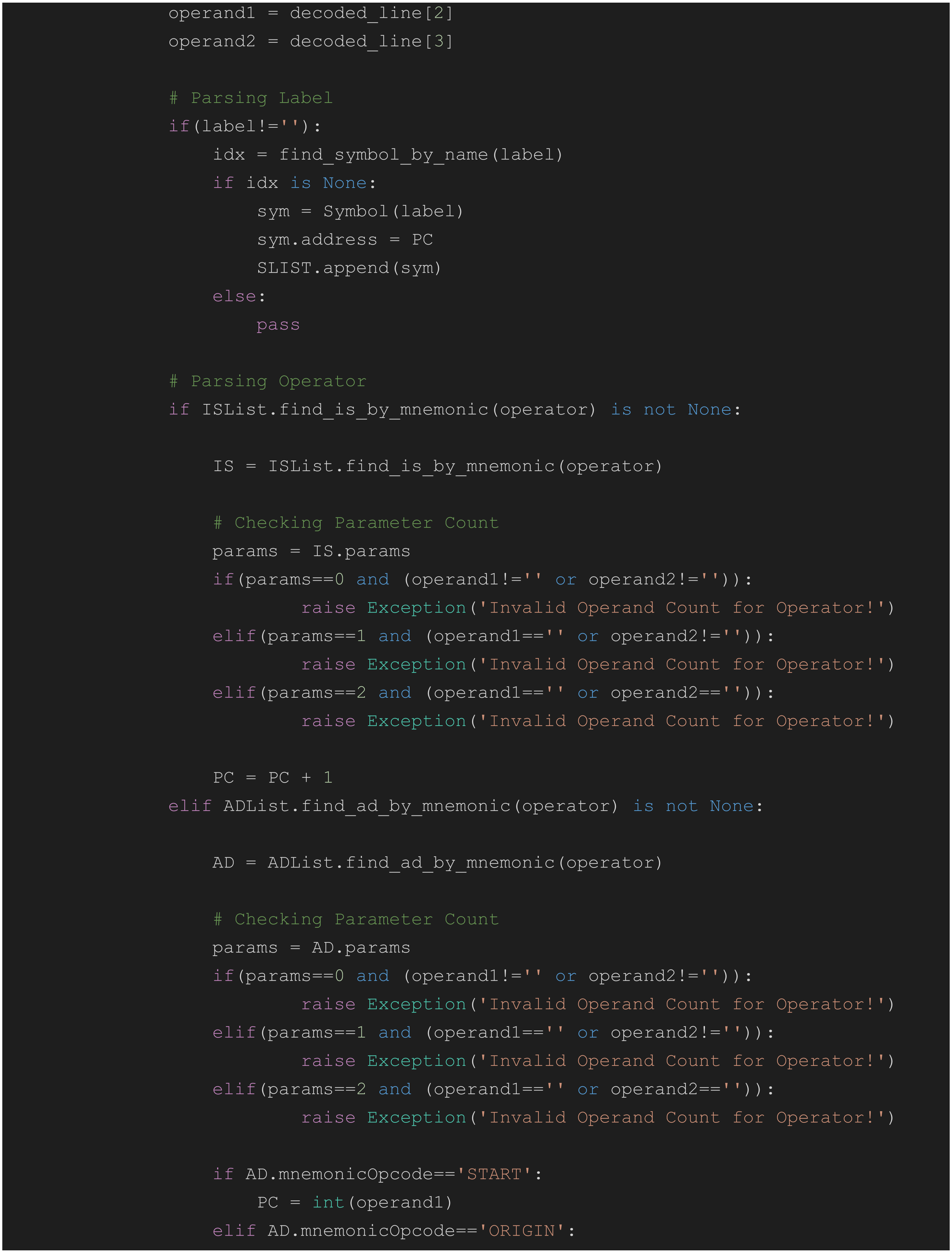
Krunal Rank
U18CO081

1.Generate variant-I and variant-II representation for multiplication of two numbers.

__main__.py:

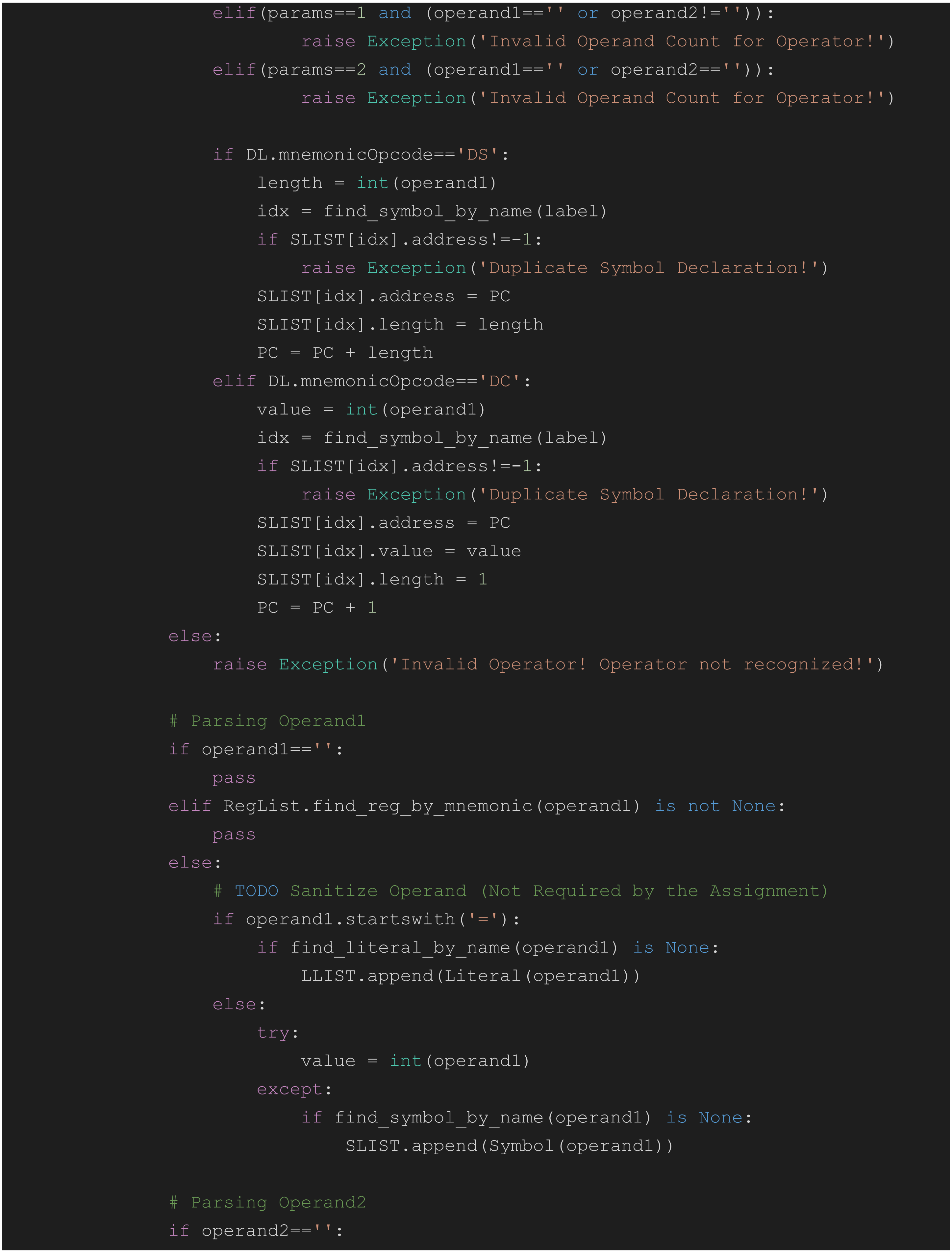
```
# Required libraries
import argparse

# In built Data Structures
from register import RegList
from condition_code import CCList
from assembler_directive import ADList
from declarative_statement import DLList
from imperative_statement import ISList

# Other Data Structures
from symbol import Symbol
from pool import Pool
from literal import Literal


#In built Data Structures
CCList = CCList()
ADList = ADList()
DLList = DLList()
ISList = ISList()
RegList = RegList()

# Program Counter
PC = 0

# Stack for Origin PC
PCLIST = []

# Literal List, Symbol List, Pool List
LLIST = []
SLIST = []
PLIST = []
```

```python
def find_literal_by_name(name):
    for i in range(len(LLIST)):
        if LLIST[i].name==name:
            return i
    return None

def find_symbol_by_name(name):
    for i in range(len(SLIST)):
        if SLIST[i].name==name:
            return i
    return None

# END Command is used
SAFE_END = False

if __name__ == '__main__':

    # Parsing FilePath as Arguments
    parser = argparse.ArgumentParser(description="Generates Target Code for Assembly Source Code")
    parser.add_argument('file_path',metavar='filePath',help='File Path to Assembly Source Code')
    args = parser.parse_args()
    file_path = args.file_path


    # Parsing the File
    try:
        with open(file_path,'r') as f:
            lines = f.readlines() # lines = List of lines in file f
            line_count = 0
            for line in lines:
                line_count = line_count + 1

                line = line.upper().replace('\n','').replace('\t','').replace(' ','').replace(',','')
                decoded_line = line.split('-')
                if len(decoded_line)!=4:
                    raise Exception('4 Parameters required in each assembly statement!')

                print(decoded_line)
                label = decoded_line[0]
                operator = decoded_line[1]
```

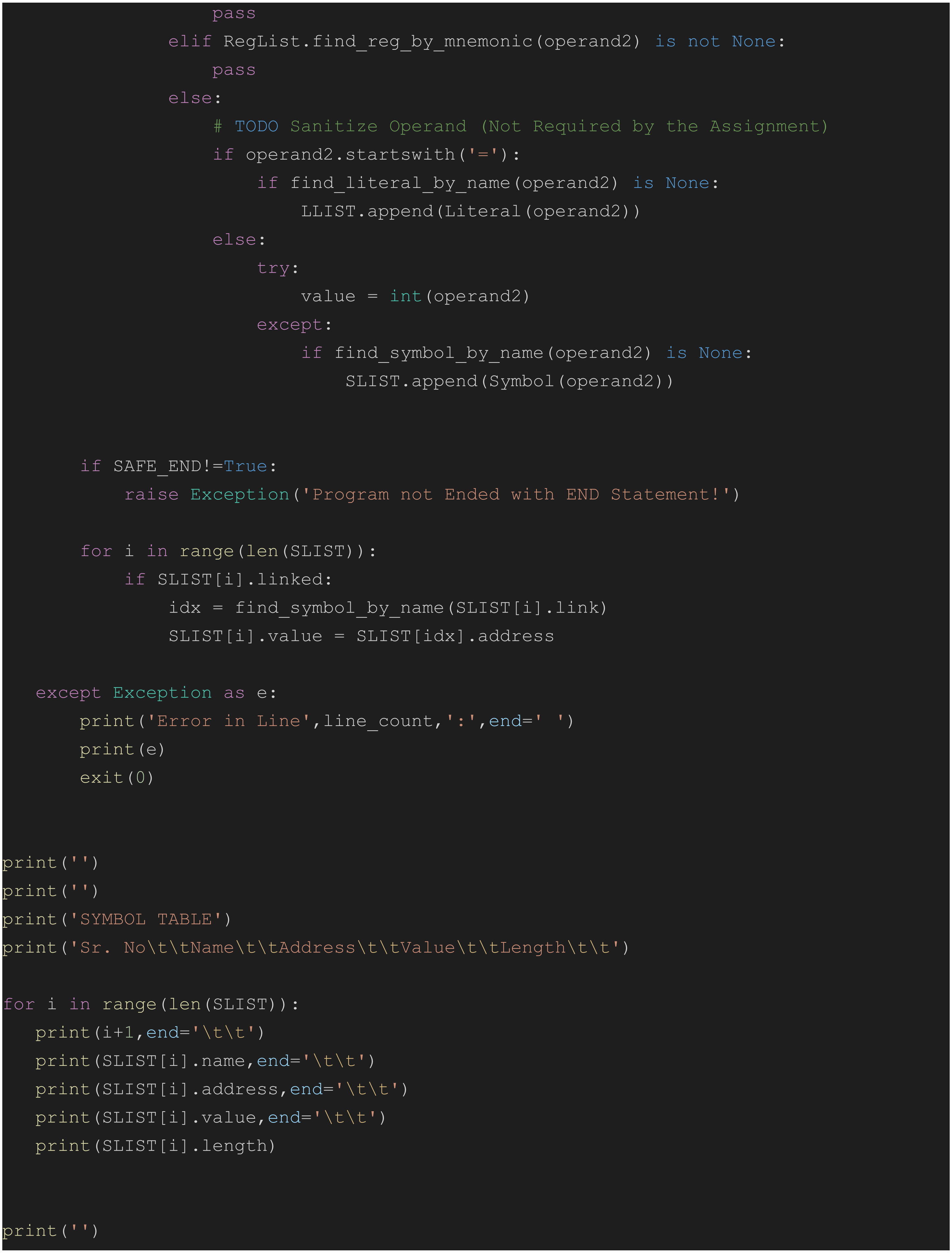


```
operand1 = decoded_line[2]
operand2 = decoded_line[3]

# Parsing Label
if(label!=''):
    idx = find_symbol_by_name(label)
    if idx is None:
        sym = Symbol(label)
        sym.address = PC
        SLIST.append(sym)
    else:
        pass

# Parsing Operator
if ISList.find_is_by_mnemonic(operator) is not None:

    IS = ISList.find_is_by_mnemonic(operator)

    # Checking Parameter Count
    params = IS.params
    if(params==0 and (operand1!='' or operand2!='')):
            raise Exception('Invalid Operand Count for Operator!')
    elif(params==1 and (operand1=='' or operand2!='')):
            raise Exception('Invalid Operand Count for Operator!')
    elif(params==2 and (operand1=='' or operand2=='')):
            raise Exception('Invalid Operand Count for Operator!')

    PC = PC + 1
elif ADList.find_ad_by_mnemonic(operator) is not None:

    AD = ADList.find_ad_by_mnemonic(operator)

    # Checking Parameter Count
    params = AD.params
    if(params==0 and (operand1!='' or operand2!='')):
            raise Exception('Invalid Operand Count for Operator!')
    elif(params==1 and (operand1=='' or operand2!='')):
            raise Exception('Invalid Operand Count for Operator!')
    elif(params==2 and (operand1=='' or operand2=='')):
            raise Exception('Invalid Operand Count for Operator!')

    if AD.mnemonicOpcode=='START':
        PC = int(operand1)
    elif AD.mnemonicOpcode=='ORIGIN':
```

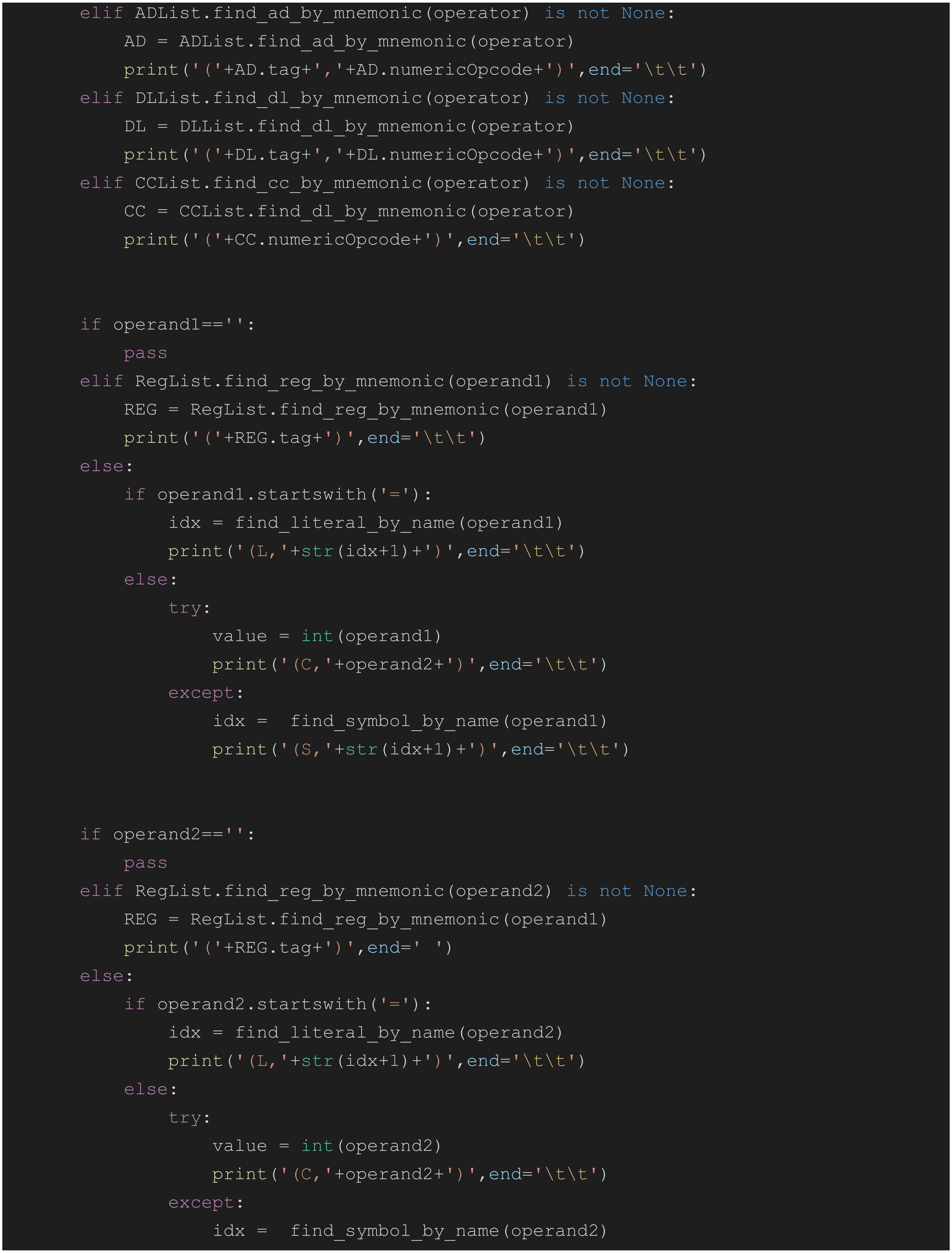
```
                        PCLIST.append(PC)
                        PC = int(operand1)
                    elif AD.mnemonicOpcode=='STOP':
                        if len(PCLIST<=0):
                            raise Exception('Invalid Operator! STOP called before 
ORIGIN!')
                        PC = PCLIST[len(PCLIST)-1]
                        PCLIST.pop(len(PCLIST)-1)
                    elif AD.mnemonicOpcode=='EQU':
                        idx = find_symbol_by_name(label)
                        SLIST[idx].linked = True
                        SLIST[idx].link = operand1
                    elif AD.mnemonicOpcode=='LTORG':
                        literal_pointer = -1

                        for i in range(len(LLIST)):
                            if(LLIST[i].address==-1):
                                if literal_pointer==-1:
                                    literal_pointer = i
                                LLIST[i].address = PC
                                PC = PC + 1

                        if literal_pointer!=-1:
                            PLIST.append(Pool(literal_pointer))
                    elif AD.mnemonicOpcode=='END':
                        SAFE_END = True
                        literal_pointer = -1

                        for i in range(len(LLIST)):
                            if(LLIST[i].address==-1):
                                if literal_pointer==-1:
                                    literal_pointer = i
                                LLIST[i].address = PC
                                PC = PC + 1

                        if literal_pointer!=-1:
                            PLIST.append(Pool(literal_pointer))
                elif DLList.find_dl_by_mnemonic(operator) is not None:

                    DL = DLList.find_dl_by_mnemonic(operator)
                    # Checking Parameter Count
                    params = DL.params
                    if(params==0 and (operand1!='' or operand2!='')):
                            raise Exception('Invalid Operand Count for Operator!')
```



```
            elif(params==1 and (operand1=='' or operand2!='')):
                    raise Exception('Invalid Operand Count for Operator!')
            elif(params==2 and (operand1=='' or operand2=='')):
                    raise Exception('Invalid Operand Count for Operator!')

            if DL.mnemonicOpcode=='DS':
                length = int(operand1)
                idx = find_symbol_by_name(label)
                if SLIST[idx].address!=-1:
                    raise Exception('Duplicate Symbol Declaration!')
                SLIST[idx].address = PC
                SLIST[idx].length = length
                PC = PC + length
            elif DL.mnemonicOpcode=='DC':
                value = int(operand1)
                idx = find_symbol_by_name(label)
                if SLIST[idx].address!=-1:
                    raise Exception('Duplicate Symbol Declaration!')
                SLIST[idx].address = PC
                SLIST[idx].value = value
                SLIST[idx].length = 1
                PC = PC + 1
        else:
            raise Exception('Invalid Operator! Operator not recognized!')

        # Parsing Operand1
        if operand1=='':
            pass
        elif RegList.find_reg_by_mnemonic(operand1) is not None:
            pass
        else:
            # TODO Sanitize Operand (Not Required by the Assignment)
            if operand1.startswith('='):
                if find_literal_by_name(operand1) is None:
                    LLIST.append(Literal(operand1))
            else:
                try:
                    value = int(operand1)
                except:
                    if find_symbol_by_name(operand1) is None:
                        SLIST.append(Symbol(operand1))

        # Parsing Operand2
        if operand2=='':
```

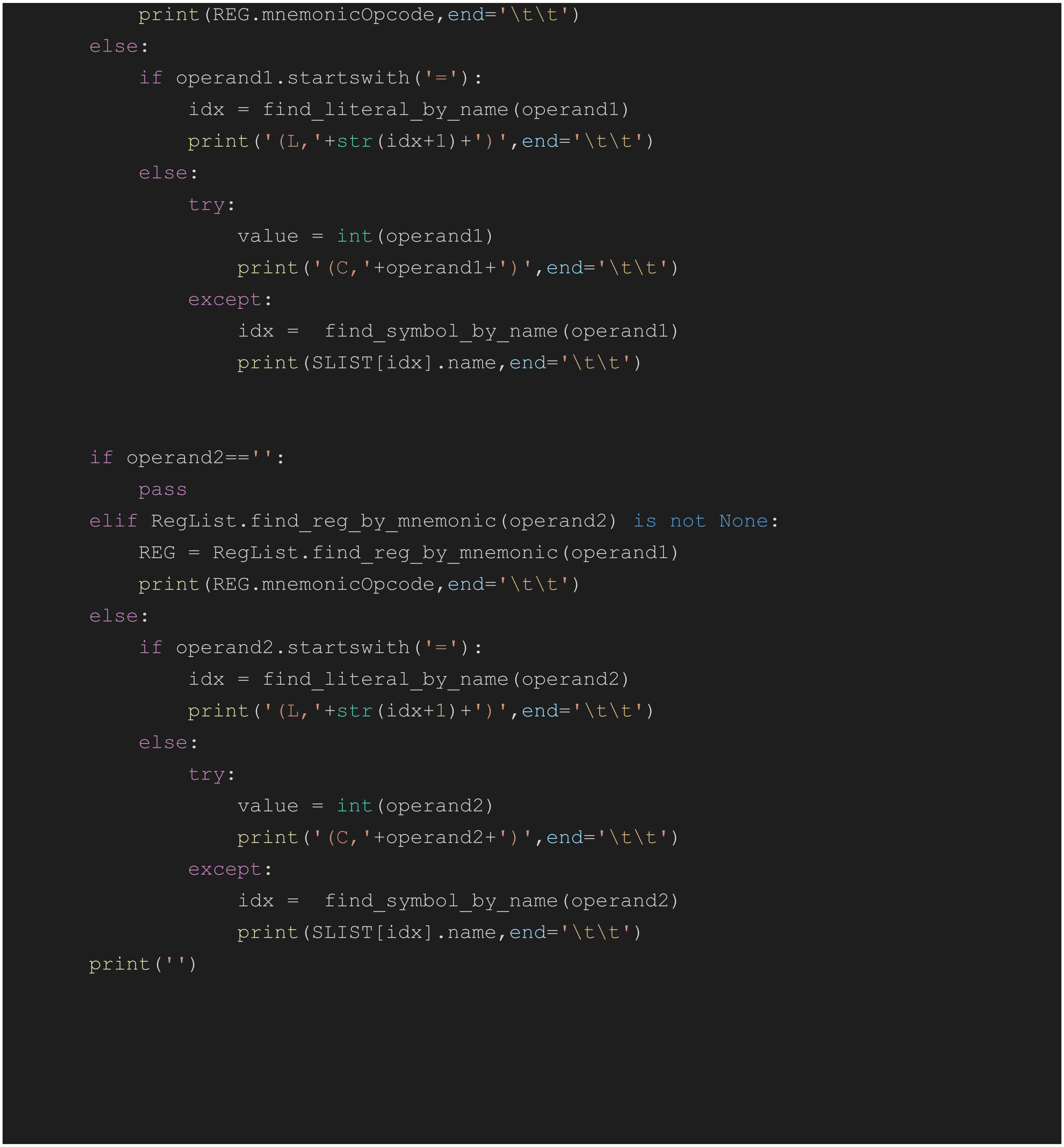


```
                    pass
                elif RegList.find_reg_by_mnemonic(operand2) is not None:
                    pass
                else:
                    # TODO Sanitize Operand (Not Required by the Assignment)
                    if operand2.startswith('='):
                        if find_literal_by_name(operand2) is None:
                            LLIST.append(Literal(operand2))
                    else:
                        try:
                            value = int(operand2)
                        except:
                            if find_symbol_by_name(operand2) is None:
                                SLIST.append(Symbol(operand2))


        if SAFE_END!=True:
            raise Exception('Program not Ended with END Statement!')

        for i in range(len(SLIST)):
            if SLIST[i].linked:
                idx = find_symbol_by_name(SLIST[i].link)
                SLIST[i].value = SLIST[idx].address

    except Exception as e:
        print('Error in Line',line_count,':',end=' ')
        print(e)
        exit(0)


print('')
print('')
print('SYMBOL TABLE')
print('Sr. No\t\tName\t\tAddress\t\tValue\t\tLength\t\t')

for i in range(len(SLIST)):
    print(i+1,end='\t\t')
    print(SLIST[i].name,end='\t\t')
    print(SLIST[i].address,end='\t\t')
    print(SLIST[i].value,end='\t\t')
    print(SLIST[i].length)


print('')
print('')
print('LITERAL TABLE')
print('Sr. No\t\tName\t\tAddress')

for i in range(len(LLIST)):
   print(i+1,end='\t\t')
   print(LLIST[i].name,end='\t\t')
   print(LLIST[i].address)


print('')
print('')
print('POOL TABLE')
print('Sr. No\t\tLiteral Pointer')

for i in range(len(PLIST)):
   print(i+1,end='\t\t')
   print(PLIST[i].literal_pointer+1)


print('')
print('')
print('Variant 1 Intermediate Code')
with open(file_path,'r') as f:
   lines = f.readlines() # lines = List of lines in file f
   line_count = 0
   for line in lines:
       line_count = line_count + 1

       line = line.upper().replace('\n','').replace('\t','').replace('
',''').replace(',','')
       decoded_line = line.split('-')
       if len(decoded_line)!=4:
           raise Exception('4 Parameters required in each assembly statement!')

       label = decoded_line[0]
       operator = decoded_line[1]
       operand1 = decoded_line[2]
       operand2 = decoded_line[3]


       if ISList.find_is_by_mnemonic(operator) is not None:
           IS = ISList.find_is_by_mnemonic(operator)
           print('('+IS.tag+','+IS.numericOpcode+')',end='\t\t')
```



```
    elif ADList.find_ad_by_mnemonic(operator) is not None:
        AD = ADList.find_ad_by_mnemonic(operator)
        print('('+AD.tag+','+AD.numericOpcode+')',end='\t\t')
    elif DLList.find_dl_by_mnemonic(operator) is not None:
        DL = DLList.find_dl_by_mnemonic(operator)
        print('('+DL.tag+','+DL.numericOpcode+')',end='\t\t')
    elif CCList.find_cc_by_mnemonic(operator) is not None:
        CC = CCList.find_dl_by_mnemonic(operator)
        print('('+CC.numericOpcode+')',end='\t\t')


    if operand1=='':
        pass
    elif RegList.find_reg_by_mnemonic(operand1) is not None:
        REG = RegList.find_reg_by_mnemonic(operand1)
        print('('+REG.tag+')',end='\t\t')
    else:
        if operand1.startswith('='):
            idx = find_literal_by_name(operand1)
            print('(L,'+str(idx+1)+')',end='\t\t')
        else:
            try:
                value = int(operand1)
                print('(C,'+operand2+')',end='\t\t')
            except:
                idx =  find_symbol_by_name(operand1)
                print('(S,'+str(idx+1)+')',end='\t\t')


    if operand2=='':
        pass
    elif RegList.find_reg_by_mnemonic(operand2) is not None:
        REG = RegList.find_reg_by_mnemonic(operand1)
        print('('+REG.tag+')',end=' ')
    else:
        if operand2.startswith('='):
            idx = find_literal_by_name(operand2)
            print('(L,'+str(idx+1)+')',end='\t\t')
        else:
            try:
                value = int(operand2)
                print('(C,'+operand2+')',end='\t\t')
            except:
                idx =  find_symbol_by_name(operand2)
```

```
                    print('(S,'+str(idx+1)+')',end='\t\t')
        print('')


print('')
print('')
print('Variant 2 Intermediate Code')
with open(file_path,'r') as f:
    lines = f.readlines() # lines = List of lines in file f
    line_count = 0
    for line in lines:
        line_count = line_count + 1

        line = line.upper().replace('\n','').replace('\t','').replace('
','').replace(',','')
        decoded_line = line.split('-')
        if len(decoded_line)!=4:
            raise Exception('4 Parameters required in each assembly statement!')

        label = decoded_line[0]
        operator = decoded_line[1]
        operand1 = decoded_line[2]
        operand2 = decoded_line[3]


        if ISList.find_is_by_mnemonic(operator) is not None:
            IS = ISList.find_is_by_mnemonic(operator)
            print('('+IS.tag+','+IS.numericOpcode+')',end='\t\t')
        elif ADList.find_ad_by_mnemonic(operator) is not None:
            AD = ADList.find_ad_by_mnemonic(operator)
            print('('+AD.tag+','+AD.numericOpcode+')',end='\t\t')
        elif DLList.find_dl_by_mnemonic(operator) is not None:
            DL = DLList.find_dl_by_mnemonic(operator)
            print('('+DL.tag+','+DL.numericOpcode+')',end='\t\t')
        elif CCList.find_cc_by_mnemonic(operator) is not None:
            CC = CCList.find_dl_by_mnemonic(operator)
            print('('+CC.numericOpcode+')',end='\t\t')



        if operand1=='':
            pass
        elif RegList.find_reg_by_mnemonic(operand1) is not None:
            REG = RegList.find_reg_by_mnemonic(operand1)
```



```
            print(REG.mnemonicOpcode,end='\t\t')
        else:
            if operand1.startswith('='):
                idx = find_literal_by_name(operand1)
                print('(L,'+str(idx+1)+')',end='\t\t')
            else:
                try:
                    value = int(operand1)
                    print('(C,'+operand1+')',end='\t\t')
                except:
                    idx =  find_symbol_by_name(operand1)
                    print(SLIST[idx].name,end='\t\t')


        if operand2=='':
            pass
        elif RegList.find_reg_by_mnemonic(operand2) is not None:
            REG = RegList.find_reg_by_mnemonic(operand1)
            print(REG.mnemonicOpcode,end='\t\t')
        else:
            if operand2.startswith('='):
                idx = find_literal_by_name(operand2)
                print('(L,'+str(idx+1)+')',end='\t\t')
            else:
                try:
                    value = int(operand2)
                    print('(C,'+operand2+')',end='\t\t')
                except:
                    idx =  find_symbol_by_name(operand2)
                    print(SLIST[idx].name,end='\t\t')
        print('')
```

Inbuilt Data Structures:

imperative_statement.py:

```
class ImperativeStatement:
   def __init__(self, mO, nO,p):
       self.mnemonicOpcode = mO
       self.numericOpcode = nO
       self.tag = "IS"
       self.params = p

class ISList:
   def __init__(self):

       self.list = [
           ImperativeStatement("STOP", "00",0),
           ImperativeStatement("ADD", "01",2),
           ImperativeStatement("SUB", "02",2),
           ImperativeStatement("MUL", "03",2),
           ImperativeStatement("MOVER", "04",2),
           ImperativeStatement("MOVEM", "05",2),
           ImperativeStatement("COMP", "06",2),
           ImperativeStatement("BC", "07",2),
           ImperativeStatement("DIV", "08",2),
           ImperativeStatement("READ", "09",1),
           ImperativeStatement("PRINT", "10",1),
       ]

   def find_is_by_mnemonic(self,mO):
       for IS in self.list:
           if IS.mnemonicOpcode==mO:
               return IS
       return None

   def find_is_by_numeric(self,nO):
       for IS in self.list:
           if IS.numericOpcode==nO:
               return IS
       return None
```

assembler_directive.py:

```
class AssemblerDirective:

   def __init__(self,mO,nO,p):
       self.mnemonicOpcode = mO
       self.numericOpcode = nO
       self.tag = "AD"
       self.params = p


class ADList:
   def __init__(self):

       self.list = [
           AssemblerDirective("START", "01",1),
           AssemblerDirective("END", "02",0),
           AssemblerDirective("ORIGIN", "03",1),
           AssemblerDirective("EQU", "04",1),
           AssemblerDirective("LTORG", "05",0),
       ]

   def find_ad_by_mnemonic(self,mO):
       for AD in self.list:
           if AD.mnemonicOpcode==mO:
               return AD
       return None

   def find_ad_by_numeric(self,nO):
       for AD in self.list:
           if AD.numericOpcode==nO:
               return AD
       return None
```

condition_code.py:

```
class ConditionCode:

   def __init__(self,mO,nO):
       self.mnemonicOpcode = mO
       self.numericOpcode = nO



class CCList:
   def __init__(self):

       self.list = [
           ConditionCode("LT", "01"),
           ConditionCode("LE", "02"),
           ConditionCode("EQ", "03"),
           ConditionCode("GT", "04"),
           ConditionCode("GE", "05"),
           ConditionCode("ANY", "06"),
       ]

   def find_cc_by_mnemonic(self,mO):
       for CC in self.list:
           if CC.mnemonicOpcode==mO:
               return CC
       return None

   def find_cc_by_numeric(self,nO):
       for CC in self.list:
           if CC.numericOpcode==nO:
               return CC
       return None
```

declarative_statement.py:

```
class DeclarativeStatement:

    def __init__(self,mO,nO,p):
        self.mnemonicOpcode = mO
        self.numericOpcode = nO
        self.tag = "DL"
        self.params = p


class DLList:
    def __init__(self):

        self.list = [
            DeclarativeStatement("DS", "01",1),
            DeclarativeStatement("DC", "02",1),
        ]

    def find_dl_by_mnemonic(self,mO):
        for DL in self.list:
            if DL.mnemonicOpcode==mO:
                return DL
        return None

    def find_dl_by_numeric(self,nO):
        for DL in self.list:
            if DL.numericOpcode==nO:
                return DL
        return None
```

register.py:

```
class Register:

    def __init__(self,mO,nO):
        self.mnemonicOpcode = mO
        self.numericOpcode = nO
        self.tag = nO


class RegList:
    def __init__(self):

        self.list = [
            Register("AREG", "01"),
            Register("BREG", "02"),
            Register("CREG", "03"),
            Register("DREG", "04"),
        ]

    def find_reg_by_mnemonic(self,mO):
        for REG in self.list:
            if REG.mnemonicOpcode==mO:
                return REG
        return None

    def find_reg_by_numeric(self,nO):
        for REG in self.list:
            if REG.numericOpcode==nO:
                return REG
        return None
```

Other Data Structures:

symbol.py:

```
class Symbol:

    def __init__(self,name):
        self.name = name
        self.address = -1
        self.length = -1
        self.value = -1
        self.linked = False
        self.link = ""

```

literal.py:

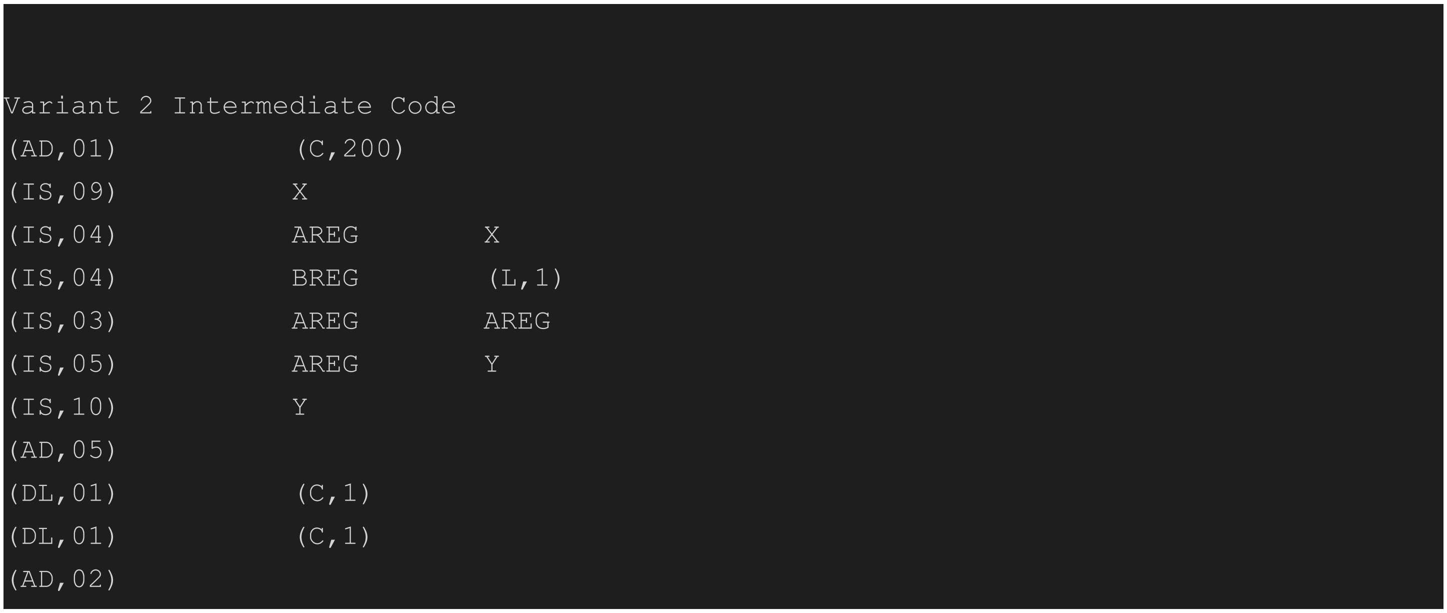
```
class Literal:

    def __init__(self,name):
        self.name = name
        self.address = -1
        self.pool = -1
```

pool.py:

```
class Pool:

    def __init__(self,literal_pointer):
        self.literal_pointer = literal_pointer
```

Output:

```
['', 'START', '200', '']
['', 'READ', 'X', '']
['', 'MOVER', 'AREG', 'X']
['', 'MOVER', 'BREG', '="5"']
['', 'MUL', 'AREG', 'BREG']
['', 'MOVEM', 'AREG', 'Y']
['', 'PRINT', 'Y', '']
['', 'LTORG', '', '']
['X', 'DS', '1', '']
['Y', 'DS', '1', '']
['', 'END', '', '']


SYMBOL TABLE
Sr. No            Name        Address           Value       Length
1           X           207         -1          1
2           Y           208         -1          1


LITERAL TABLE
Sr. No            Name        Address
1           ="5"        206


POOL TABLE
Sr. No            Literal Pointer
1           1


Variant 1 Intermediate Code
(AD,01)           (C,)
(IS,09)           (S,1)
(IS,04)           (01)        (S,1)
(IS,04)           (02)        (L,1)
(IS,03)           (01)        (01)
(IS,05)           (01)        (S,2)
(IS,10)           (S,2)
(AD,05)
(DL,01)           (C,)
(DL,01)           (C,)
(AD,02)
```



```
Variant 2 Intermediate Code
(AD,01)          (C,200)
(IS,09)          X
(IS,04)          AREG       X
(IS,04)          BREG       (L,1)
(IS,03)          AREG       AREG
(IS,05)          AREG       Y
(IS,10)          Y
(AD,05)
(DL,01)          (C,1)
(DL,01)          (C,1)
(AD,02)
```