Batch: A2	**Roll No.: 1911027**
Experiment No. 09	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Design and implement 2 pass assembler

AIM: To understand designing and implementation of two pass assembler.

Expected Outcome of Experiment:

CO 5. To understand the designing and implementation of system software like Assembler, Macro preprocessor and linker loader

Books/ Journals/ Websites referred:

1. John J. Donovan : Systems programming, Tata McGraw Hill.

Pre Lab/ Prior Concepts:

An assembler is a translator that translates an assembler program into a conventional machine language program. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are.

Consider an assembler instruction like the following

```
JMP  LATER
        ...
        ...
LATER:
```

This is known as a forward reference. If the assembler is processing the file one line at a time, then it doesn't know where LATER is when it first encounters the jump instruction. So, it doesn't know if the jump is a short jump, a near jump or a far jump.

So, Assembler scan the code twice. The first time, just count how long the machine code instructions will be, just to find out the addresses of all the labels. Also, create a table that has a list of all the addresses and where they will be in the program. This table is known as the symbol table. On the second scan, generate the machine code, and use the symbol table to determine how far away jump labels are, and to generate the most efficient instruction.

This is known as a two-pass assembler. Each pass scans the program, the first pass generates the symbol table and the second pass generates the machine code. I have created a listing of an assembler program that has the machine code listed, and the symbol table listed.

Algorithm for pass 1:

```
Step 1: Start
Step 2: Initialize location counter to zero.
Step 3: Read opcode field of next instruction.
Step 4: Search opcode in pseudo opcode table.
Step 5: If opcode is found in Pseudo Opcode Table
        Step 5.1: If it is ‘DS’ or ‘DC’
        Adjust location counter to proper alignment.
        Assign length of data field to ‘L’
        Go to step 9
        Step 5.2: If it is ‘EQU’
        Evaluate operand field
        Assign value to symbol in label field
        Go to step 3
        Step 5.3: If it is ‘USING’ or ‘DROP’ Go to step 3
        Step 5.4: If it is ‘END’
        Assign storage locations to literals
        Go to step 12
Step 6: Search opcode in Machine Opcode Table.
Step 7: Assign it’s length to ‘L’.
Step 8: Process any literals and enter them into Literal Table.
Step 9: If symbol is there in label field assign current value of Location Counter to symbol.
Step 10: Location Counter = Location Counter + L.
Step 11: Go to step 3
Step 12: Stop.
```

Algorithm for Pass 2:

```
Step 1: Start
Step 2: Initialize location counter to zero.
Step 3: Read opcode field of next instruction.
Step 4: Search opcode in pseudo opcode table.
Step 5: If opcode is found in Pseudo Opcode Table
        Step 5.1: If it is ‘DS’ or ‘DC’
        Adjust location counter to proper alignment.
        If it is ‘DC’ opcode form constant and insert in assembled program
        Assign length of data field to ‘L’
        Go to step 6.4
        Step 5.2: If it is ‘EQU’ or ‘START’ ignore it.Go to step 3
        Step 5.3: If it is ‘USING’
        Evaluate operand and enter base reg no. and value into base table
        Go to step 3
        Step 5.3: If it is ‘DROP’
        Indicate base reg no. available in base table Go to step 3
        Step 5.4: If it is ‘END’
        Generate literals for entries in Literal table
        Go to step 12
Step 6: Search opcode in Machine Opcode Table.
Step 7: Get opcode byte and format code.
Step 8: Assign it’s length to ‘L’.
Step 9: Check type of instruction.
Step 10: If it is of ‘RR’ type
```

```
        Step 10.1: Evaluate both register expressions and insert into second byte.
        Step 10.2: Assemble instruction.
        Step 10.3: Location Counter = Location Counter + L.
        Step 10.4: Go to step 3
Step 11: If it is of 'RX' type
        Step 11.1: Evaluate register and index expressions and insert into second byte.
        Step 11.2: Calculate effective address of operand.
        Step 11.3: Determine appropriate displacement and base register
        Step 11.2: Put base and displacement into bytes 3 and 4
        Step 11.3: Location Counter = Location Counter + L.
        Step 11.4: Go to step 10.2
Step 12: Stop.
```

Stepwise-Procedure:

1) Read a Source Program as input.
2) Consider Mnemonic Opcode Table (MOT), Pseudo-op table (POT), Location Counter (LC).
3) Create symbols Table (SYMTAB) and literal Table (LITAB), and intermediate code(IC).
4) Update the Location Counter (LC).

Sample input and output:

Example : 1

Input file:

```
PRGAM      START        0
            USING        * , 15
            LA          15, SETUP
            SR          TOTAL, TOTAL
AC          EQU         2
INDEX       EQU         3
TOTAL       EQU         4
DATABASE    EQU         13
SETUP       EQU         *
            USING       SETUP, 15
            L           DATABASE, =A(DATA1)
```

```
                USING        DATAAREA, DATABASE
                SR           INDEX, INDEX
LOOP            L            AC, DATA1(INDEX)
                AR           TOTAL, AC
                A            AC, =F'5'
                ST           AC, SAVE(INDEX)
                A            INDEX, =F'4'
                C            INDEX, =F'8000'
                BNE          LOOP
                LR           1, TOTAL
                BR           14
                LTORG

SAVE            DS           2000F
DATAAREA        EQU          *
DATA1           DC           F'25, 26, 27'
END
```

Sample Output:

Symbol Table created by Pass-1

```
Symbol       Value   Length      R/A
------------------------------------------------------------
PRGAM        0          1          R
AC           2          1          A
INDEX        3          1          A
TOTAL        4          1          A
DATABASE     13         1          A
SETUP        6          1          R
LOOP         12         4          R
SAVE         64         4          R
DATAAREA     8064       1          R
```

DATA1	8064	4	R

Literal Table created by Pass-1

Literal	Value	Length	R/A
A(DATA1)	48	4	R
F'5'	52	4	R
F'4'	56	4	R
F'8000'	60	4	R

Object code generated by Pass-2

Loc	**Mnemonic**	**Operand**	
0	LA	15,	6 (0,15)
4	SR	4,	4
6	L	13,	42 (0,15)
10	SR	3,	3
12	L	2,	0 (3,13)
16	AR	4,	2
18	A	2,	46 (0,15)
22	ST	2,	58 (3,15)
26	A	3,	50(0,15)
30	C	3,	54(0,15)
34	BC	7,	6 (0,15)
38	LR	1,	4
40	BCR	15,	14
48	8064		
52	X’00000005’		
56	X’00000004’		
60	8000		
64	-------- 2000 entries.		
8064			

<u>Sample input and output:</u>

<u>Example:</u>

Input file:

```
PRGAM1      START       0
            USING        *, 15
            SR          4, INDEX
            SR          5, INDEX
             L          3,=F’5’
INDEX        EQU        3

LOOP         L          2, SETUP
            A           2,=F’49’
           ST           2, DATA
           A            4,=F'4'
           LA           1, TOTAL
           BNE          LOOP
           BR           14
           LTORG
SETUP      DC           F’34’
TOTAL      DC           F’8’
DATA        DS          10F
END
```

```
ID     Symbol      Value       Length      R/A
-------------------------------------------------------------------------
1      PRGAM1      0           1           R
2      INDEX       3           1           A
3      LOOP        8           4           R
4      SETUP       52          4           R
5      DATA        60          4           R
6      TOTAL       56          4           R
```

Literal Table created by Pass-1

LIT-No	Literal	Value	Length	R/A
1	F'5'	40	4	R
2	F'49'	44	4	R
3	F'4'	48	4	R

Intermediate code generated by Pass 1

```
            START     0
            USING      *, 15
      0     SR        4, ID#2
      2     SR        5, ID#2
      4     L         3, LT#1
      8      L        2, ID#4
     12      A        2, LT#2
    16       ST       2, ID#5
     20     A         4, LT#3
     24     LA        1, ID#6

     28     BNE       ID#3

     32     BR        14

      34    ----------

     52    DC         F’34’
     56    DC         F’8’
     60    DS         10F
    100     END
```

Code generated by Pass 2

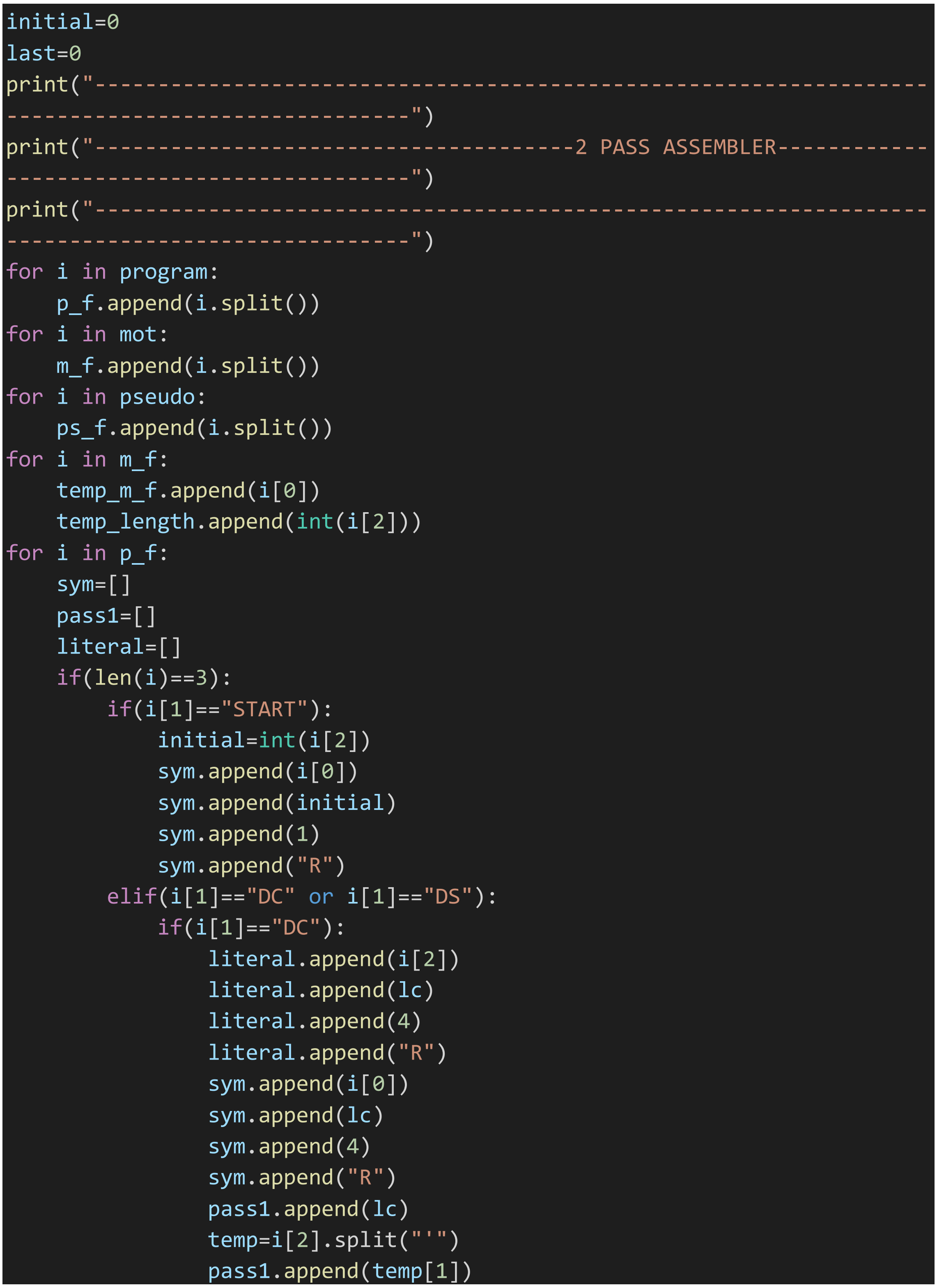
```
      0     SR/ 1B    4,  3
      2     SR/1B     5,  3
      4     L         3,  40(0,15)
```

8	**L**	**2, 52(0,15)**
12	**A**	**2, 44(0,15)**
16	**ST**	**2, 60(0,15)**
20	**A**	**4, 48(0,15)**
24	**LA**	**1, 56(0,15)**
28	**BC**	**7, 8(0,15)**
32	**BR**	**14**
34	**‘ ‘**	
40	**X’00000005’**	
44	**X’00000031’**	
48	**X’00000004’**	
52	**X’00000022’**	
56	**X’00000008’**	
60	**X’0000000A’**	
100		

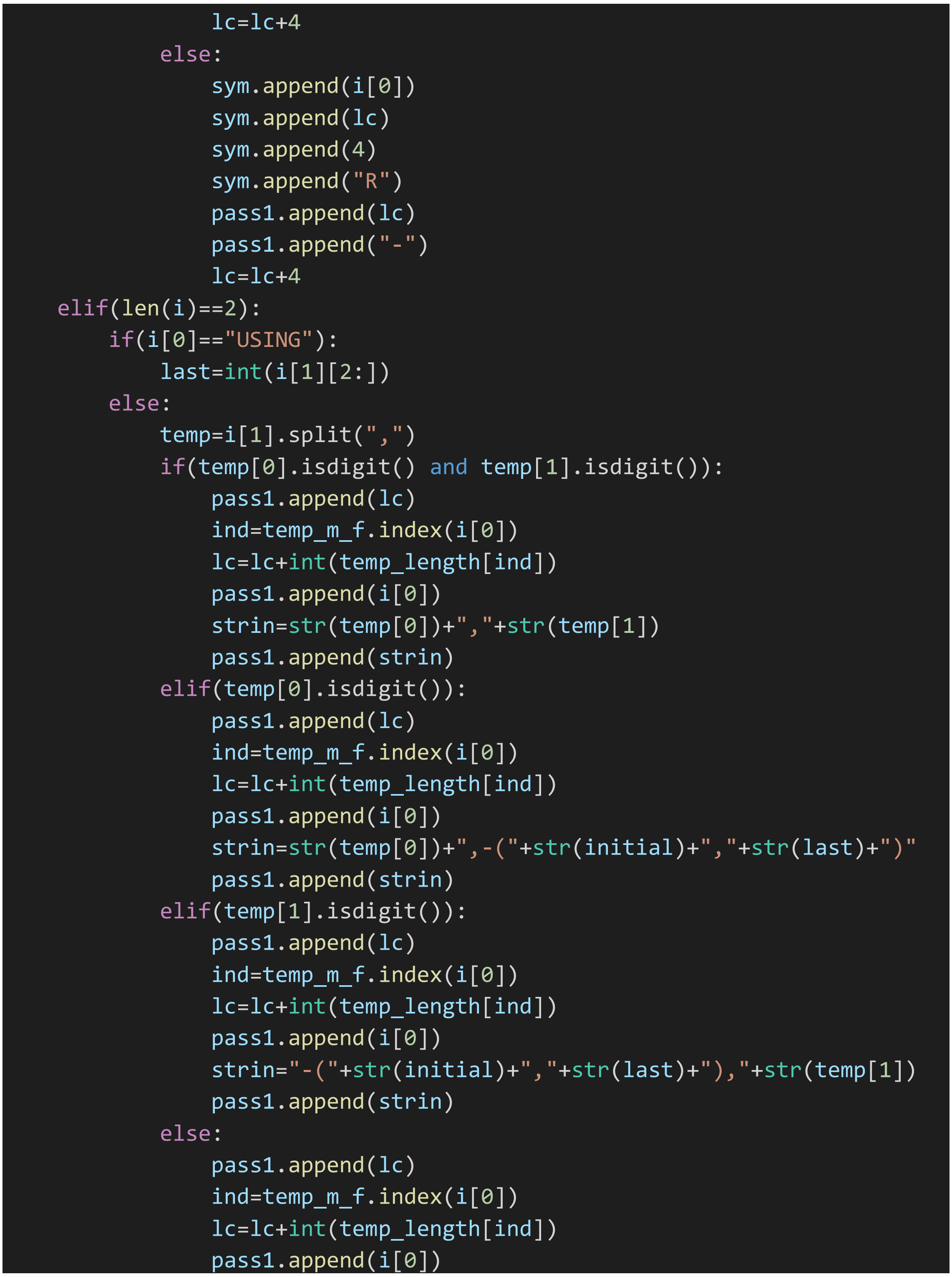
Implementation details:

Code :

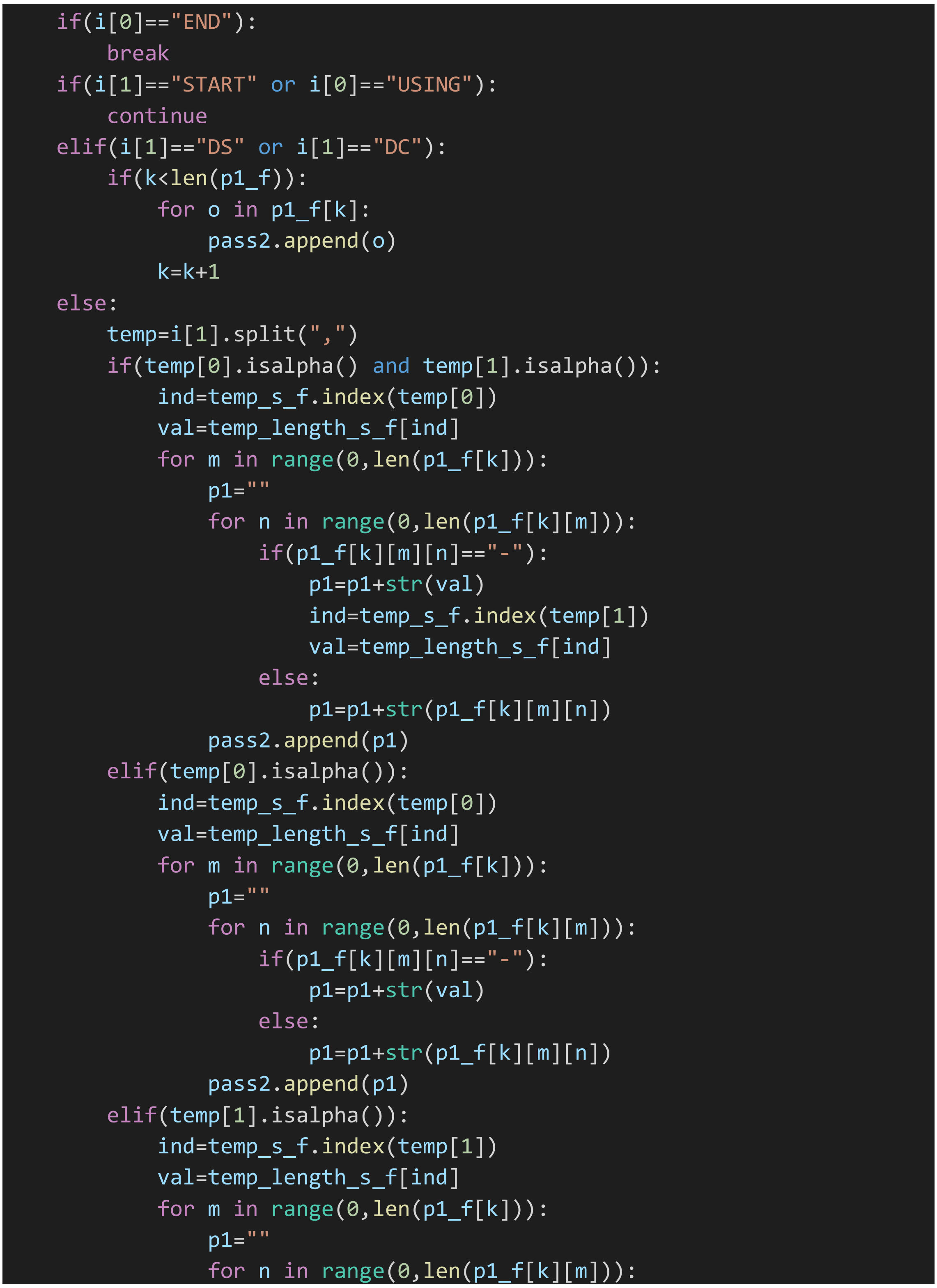
```
from prettytable import PrettyTable
program=open('program.txt','r')
mot=open('mot.txt','r')
pseudo=open('pseudo.txt','r')
pass1txt=open('pass1.txt','a')
symboltxt=open('symbol.txt','a')
literaltxt=open('literal.txt','a')
p_f=[]
m_f=[]
temp_m_f=[]
temp_length=[]
ps_f=[]
lc=0
```



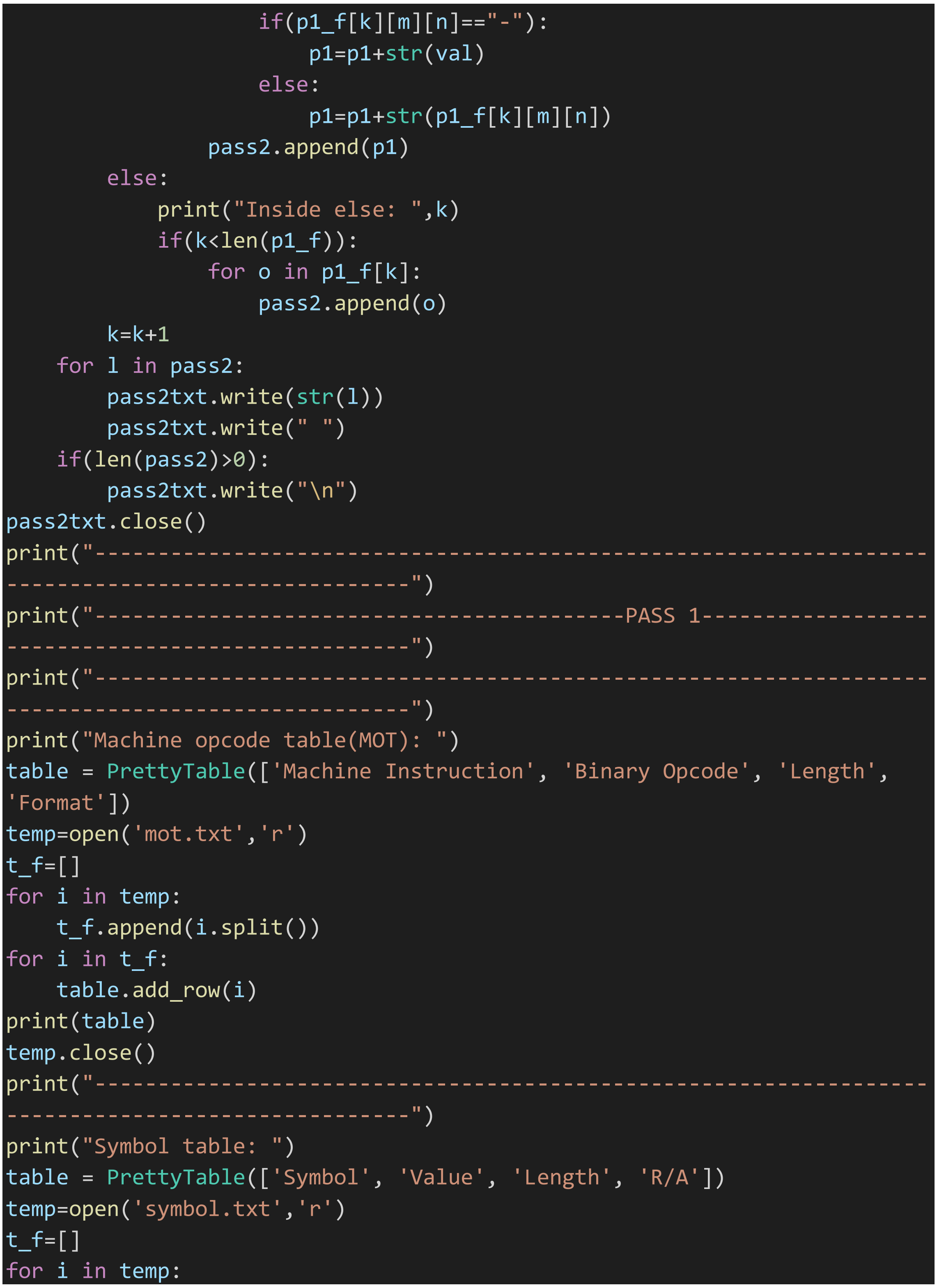
```
initial=0
last=0
print("------------------------------------------------------------------
--------------------------------")
print("--------------------------------------2 PASS ASSEMBLER------------
--------------------------------")
print("------------------------------------------------------------------
--------------------------------")
for i in program:
    p_f.append(i.split())
for i in mot:
    m_f.append(i.split())
for i in pseudo:
    ps_f.append(i.split())
for i in m_f:
    temp_m_f.append(i[0])
    temp_length.append(int(i[2]))
for i in p_f:
    sym=[]
    pass1=[]
    literal=[]
    if(len(i)==3):
        if(i[1]=="START"):
            initial=int(i[2])
            sym.append(i[0])
            sym.append(initial)
            sym.append(1)
            sym.append("R")
        elif(i[1]=="DC" or i[1]=="DS"):
            if(i[1]=="DC"):
                literal.append(i[2])
                literal.append(lc)
                literal.append(4)
                literal.append("R")
                sym.append(i[0])
                sym.append(lc)
                sym.append(4)
                sym.append("R")
                pass1.append(lc)
                temp=i[2].split("'")
                pass1.append(temp[1])
```



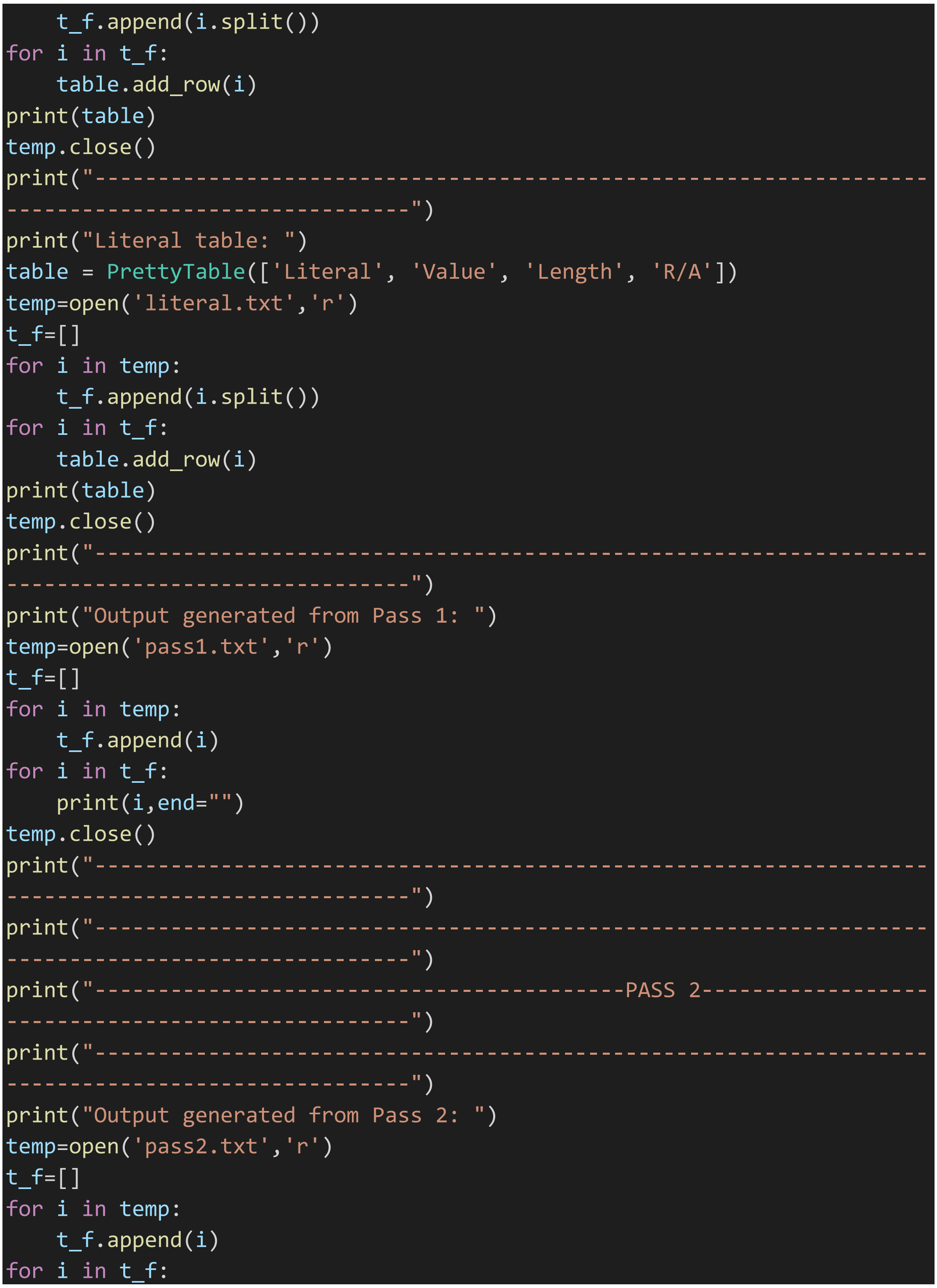
```
                lc=lc+4
            else:
                sym.append(i[0])
                sym.append(lc)
                sym.append(4)
                sym.append("R")
                pass1.append(lc)
                pass1.append("-")
                lc=lc+4
    elif(len(i)==2):
        if(i[0]=="USING"):
            last=int(i[1][2:])
        else:
            temp=i[1].split(",")
            if(temp[0].isdigit() and temp[1].isdigit()):
                pass1.append(lc)
                ind=temp_m_f.index(i[0])
                lc=lc+int(temp_length[ind])
                pass1.append(i[0])
                strin=str(temp[0])+","+str(temp[1])
                pass1.append(strin)
            elif(temp[0].isdigit()):
                pass1.append(lc)
                ind=temp_m_f.index(i[0])
                lc=lc+int(temp_length[ind])
                pass1.append(i[0])
                strin=str(temp[0])+",-("+str(initial)+","+str(last)+")"
                pass1.append(strin)
            elif(temp[1].isdigit()):
                pass1.append(lc)
                ind=temp_m_f.index(i[0])
                lc=lc+int(temp_length[ind])
                pass1.append(i[0])
                strin="-("+str(initial)+","+str(last)+"),"+str(temp[1])
                pass1.append(strin)
            else:
                pass1.append(lc)
                ind=temp_m_f.index(i[0])
                lc=lc+int(temp_length[ind])
                pass1.append(i[0])
                strin="-("+str(initial)+","+str(last)+"),-("+str(initial)+","+str(last)+")"
                pass1.append(strin)
    elif(len(i)==1):
        if(i[0]=="END"):
            break
    for k in pass1:
        pass1txt.write(str(k))
        pass1txt.write(" ")
    if(len(pass1)>0):
        pass1txt.write("\n")
    for k in sym:
        symboltxt.write(str(k))
        symboltxt.write(" ")
    if(len(sym)>0):
        symboltxt.write("\n")
    for k in literal:
        literaltxt.write(str(k))
        literaltxt.write(" ")
    if(len(literal)):
        literaltxt.write("\n")
pass1txt.close()
symboltxt.close()
literaltxt.close()
pass1txt=open('pass1.txt','r')
pass2txt=open('pass2.txt','a')
symboltxt=open('symbol.txt','r')
p1_f=[]
s_f=[]
temp_s_f=[]
temp_length_s_f=[]
for i in pass1txt:
    p1_f.append(i.split())
for i in symboltxt:
    s_f.append(i.split())
for i in s_f:
    temp_s_f.append(i[0])
    temp_length_s_f.append(i[1])
k=0
for i in p_f:
    pass2=[]
```



```
    if(i[0]=="END"):
        break
    if(i[1]=="START" or i[0]=="USING"):
        continue
    elif(i[1]=="DS" or i[1]=="DC"):
        if(k<len(p1_f)):
            for o in p1_f[k]:
                pass2.append(o)
            k=k+1
    else:
        temp=i[1].split(",")
        if(temp[0].isalpha() and temp[1].isalpha()):
            ind=temp_s_f.index(temp[0])
            val=temp_length_s_f[ind]
            for m in range(0,len(p1_f[k])):
                p1=""
                for n in range(0,len(p1_f[k][m])):
                    if(p1_f[k][m][n]=="-"):
                        p1=p1+str(val)
                        ind=temp_s_f.index(temp[1])
                        val=temp_length_s_f[ind]
                    else:
                        p1=p1+str(p1_f[k][m][n])
                pass2.append(p1)
        elif(temp[0].isalpha()):
            ind=temp_s_f.index(temp[0])
            val=temp_length_s_f[ind]
            for m in range(0,len(p1_f[k])):
                p1=""
                for n in range(0,len(p1_f[k][m])):
                    if(p1_f[k][m][n]=="-"):
                        p1=p1+str(val)
                    else:
                        p1=p1+str(p1_f[k][m][n])
                pass2.append(p1)
        elif(temp[1].isalpha()):
            ind=temp_s_f.index(temp[1])
            val=temp_length_s_f[ind]
            for m in range(0,len(p1_f[k])):
                p1=""
                for n in range(0,len(p1_f[k][m])):
```



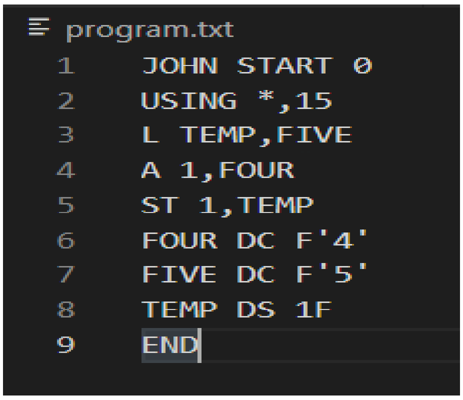
```
                    if(p1_f[k][m][n]=="-"):
                        p1=p1+str(val)
                    else:
                        p1=p1+str(p1_f[k][m][n])
                pass2.append(p1)
        else:
            print("Inside else: ",k)
            if(k<len(p1_f)):
                for o in p1_f[k]:
                    pass2.append(o)
        k=k+1
    for l in pass2:
        pass2txt.write(str(l))
        pass2txt.write(" ")
    if(len(pass2)>0):
        pass2txt.write("\n")
pass2txt.close()
print("------------------------------------------------------------------
--------------------------------")
print("------------------------------------------PASS 1------------------
--------------------------------")
print("------------------------------------------------------------------
--------------------------------")
print("Machine opcode table(MOT): ")
table = PrettyTable(['Machine Instruction', 'Binary Opcode', 'Length',
'Format'])
temp=open('mot.txt','r')
t_f=[]
for i in temp:
    t_f.append(i.split())
for i in t_f:
    table.add_row(i)
print(table)
temp.close()
print("------------------------------------------------------------------
--------------------------------")
print("Symbol table: ")
table = PrettyTable(['Symbol', 'Value', 'Length', 'R/A'])
temp=open('symbol.txt','r')
t_f=[]
for i in temp:
```



```
    t_f.append(i.split())
for i in t_f:
    table.add_row(i)
print(table)
temp.close()
print("------------------------------------------------------------------
--------------------------------")
print("Literal table: ")
table = PrettyTable(['Literal', 'Value', 'Length', 'R/A'])
temp=open('literal.txt','r')
t_f=[]
for i in temp:
    t_f.append(i.split())
for i in t_f:
    table.add_row(i)
print(table)
temp.close()
print("------------------------------------------------------------------
--------------------------------")
print("Output generated from Pass 1: ")
temp=open('pass1.txt','r')
t_f=[]
for i in temp:
    t_f.append(i)
for i in t_f:
    print(i,end="")
temp.close()
print("------------------------------------------------------------------
--------------------------------")
print("------------------------------------------------------------------
--------------------------------")
print("------------------------------------------PASS 2------------------
--------------------------------")
print("------------------------------------------------------------------
--------------------------------")
print("Output generated from Pass 2: ")
temp=open('pass2.txt','r')
t_f=[]
for i in temp:
    t_f.append(i)
for i in t_f:
    print(i,end="")
temp.close()
print("----------------------------------------------------------------
-------------------------------")
```

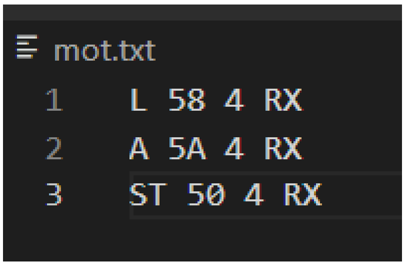
Input files:

Program.txt:



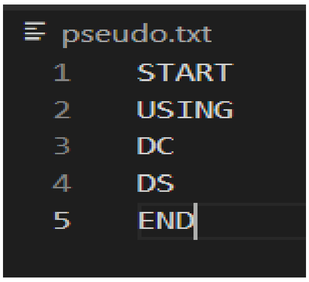
```
JOHN START 0
USING *,15
L TEMP,FIVE
A 1,FOUR
ST 1,TEMP
FOUR DC F'4'
FIVE DC F'5'
TEMP DS 1F
END
```

Mot.txt:



```
L 58 4 RX
A 5A 4 RX
ST 50 4 RX
```

Pseudo.txt:

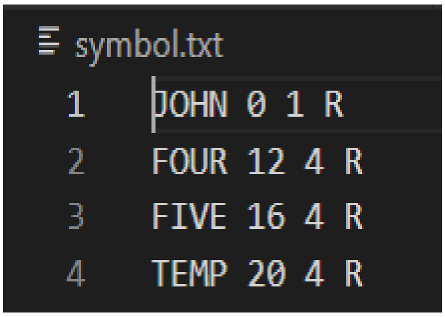


```
START
USING
DC
DS
END
```

Output:

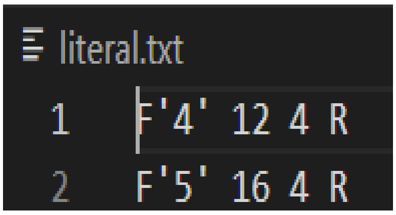
Files generated:

Symbol.txt:



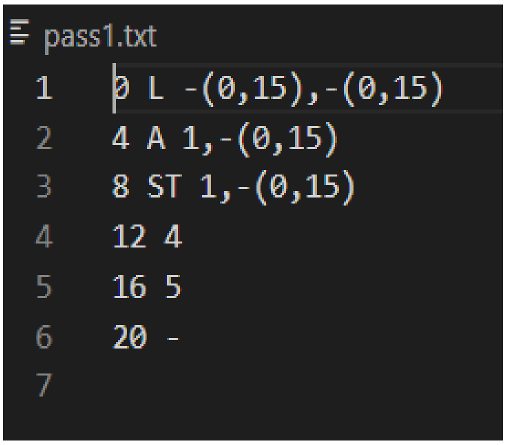
```
symbol.txt
JOHN 0 1 R
FOUR 12 4 R
FIVE 16 4 R
TEMP 20 4 R
```

Literal.txt:



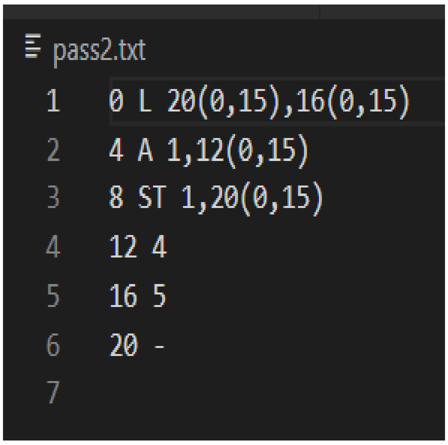
```
literal.txt
F'4' 12 4 R
F'5' 16 4 R
```

Pass1.txt:

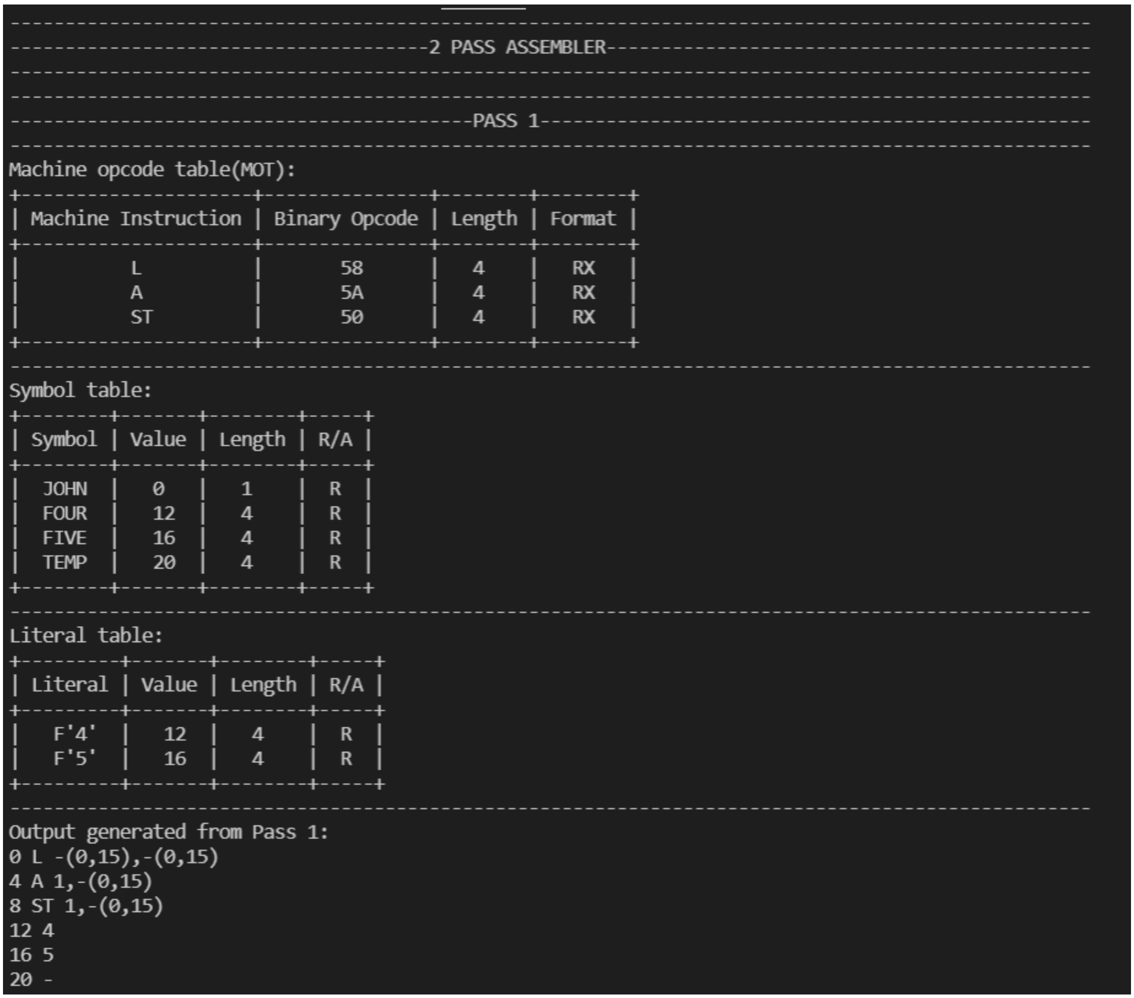


```
pass1.txt
0 L -(0,15),-(0,15)
4 A 1,-(0,15)
8 ST 1,-(0,15)
12 4
16 5
20 -
```

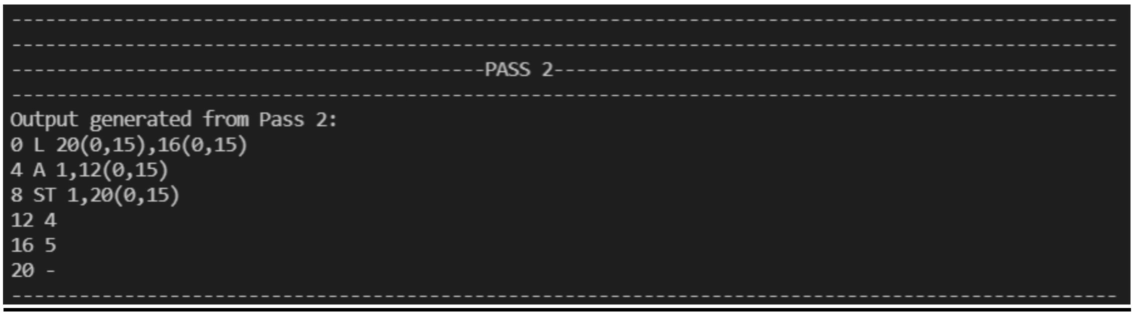
Pass2.txt:



```
pass2.txt
0 L 20(0,15),16(0,15)
4 A 1,12(0,15)
8 ST 1,20(0,15)
12 4
16 5
20 -
```



```
--------------------------------------------------------------------------------------------------
--------------------------------------2 PASS ASSEMBLER--------------------------------------------
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
------------------------------------------PASS 1--------------------------------------------------
--------------------------------------------------------------------------------------------------
Machine opcode table(MOT):
+---------------------+---------------+--------+--------+
| Machine Instruction | Binary Opcode | Length | Format |
+---------------------+---------------+--------+--------+
|          L          |      58       |   4    |   RX   |
|          A          |      5A       |   4    |   RX   |
|          ST         |      50       |   4    |   RX   |
+---------------------+---------------+--------+--------+
--------------------------------------------------------------------------------------------------
Symbol table:
+--------+-------+--------+-----+
| Symbol | Value | Length | R/A |
+--------+-------+--------+-----+
|  JOHN  |   0   |   1    |  R  |
|  FOUR  |  12   |   4    |  R  |
|  FIVE  |  16   |   4    |  R  |
|  TEMP  |  20   |   4    |  R  |
+--------+-------+--------+-----+
--------------------------------------------------------------------------------------------------
Literal table:
+---------+-------+--------+-----+
| Literal | Value | Length | R/A |
+---------+-------+--------+-----+
|   F'4'  |  12   |   4    |  R  |
|   F'5'  |  16   |   4    |  R  |
+---------+-------+--------+-----+
--------------------------------------------------------------------------------------------------
Output generated from Pass 1:
0 L -(0,15),-(0,15)
4 A 1,-(0,15)
8 ST 1,-(0,15)
12 4
16 5
20 -
```



```
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-------------------------------------------PASS 2---------------------------------------------------
----------------------------------------------------------------------------------------------------
Output generated from Pass 2:
0 L 20(0,15),16(0,15)
4 A 1,12(0,15)
8 ST 1,20(0,15)
12 4
16 5
20 -
----------------------------------------------------------------------------------------------------
```

Conclusion: **Understood how the assembly language code is converted to machine language code with the help of assemblers. Learned how 2 pass assembler works and what is the significance and working of each pass. Implemented a simple version of 2 pass assembler in python.**

Post Lab Descriptive Questions

1. What is the purpose of pass1 of assembler?

ANS) 1. Define symbols and literals and remember them in symbol table and literal table respectively.
2. Keep track of location counter.
3. Process pseudo-operations.

2. What is the purpose of pass2 of assembler?

ANS) 1. Generate object code by converting symbolic op-code into respective numeric op-code.
2. Generate data for literals and look for values of symbols.

3. What are the functions of POTGET, MOTGET,LTSTO,BTSTO,LITASS?

ANS) POTGET: Search the POT for a match with the operation field of the current source card.
MOTGET: Search the MOT for a match with the operation of the current source card.
LTSTO: Store the literal into the LT; do not store the same literal twice.
BTSTO: Insert data into appropriate entry of BT.
LITASS: Assign storage location to each literal in the literal table.

4. What is the difference between MOT of pass1 &pass2?

ANS) Pass 1: A table, the Machine-operation Table (MOT), that indicates the symbolic mnemonic, for each instruction and its length (two, four, or six bytes)
Pass 2: A table, the Machine-operation Table (MOT), that indicates for each instruction, symbolic mnemonic, length (two, four, or six bytes), binary machine opcode and format of instruction.

5. What are the examples of fixed tables? Why are they called fixed tables?

ANS) The Machine operation table (MOT) and pseudo-operation table are examples of fixed tables. The contents of these tables are not filled in or altered during the assembly process. The flowchart for Pass 1: The primary function performed by the analysis phase is the building of the symbol table.

6. What are the examples of variable tables? What is the purpose of variable tables?
ANS) The symbol table(ST), literal table(ST) are the variable tables. Purpose of variable tables is to store the data which will be required to calculate intermediate results and also to get final result after 2 passes.

7. What are various operations before transferring control to pass2?
ANS) The primary function performed by the analysis phase is the building of the symbol table. For this purpose it must determine the addresses with which the symbol names used in a program are associated. It is possible to determine some address directly, e.g. the address of the first instruction in the program, however others must be inferred. To implement memory allocation a data structure called location counter (LC) is introduced. The location counter is always made to contain the address of the next memory word in the target program. It is initialized to the constant. Whenever the analysis phase sees a label in an assembly statement, it enters the label and the contents of LC in a new entry of the symbol table. It then finds the number of memory words required by the assembly statement and updates; the LC contents. This ensure: that LC points to the next memory word in the target program even when machine instructions have different lengths and DS/DC statements reserve different amounts of memory. To update the contents of LC, analysis phase needs to know lengths of different instructions. This information simply depends on the assembly language hence the mnemonics table can be extended to include this information in a new field called length. We refer to the processing involved in maintaining the location counter as LC processing.

Post Lab Objective Questions

1. The computer language generally translated to pseudo code
a. Assembler b. Machine
c. Pascal d. FORTRAN
ANS: a. Assembler

2. Simple two pass assembler does which of the functions in the first pass?
a. It allocate space for the literals

b. It computes the total length of the program
c. It build the symbol table for the symbol and their values.
d. All of these

ANS: d. All of these

3. Symbol table implementation is based on the property of locality of reference is
a. Linear list
b. Search tree
c. Hash table
d. self organisation List

ANS: c. Hash table

4. A self-relocating program is one which
a. cannot be made to execute in any area of storage other than the one designated for it at the time of its coding or translation
b. consists of a program and relevant information for its relocation
c. can itself perform the relocation of its address-sensitive portions
d. All of the above

ANS: c. can itself perform the relocation of its address-sensitive portions

5. Assembler is
a. a program that automate the translation of assembly language into machine language
b. a program that accepts a program written in a high level language and produces an object program
c. a program that places programs into memory and prepares them for execution
d. is a program that appears to execute a source program as if it were machine language

ANS: a. a program that automate the translation of assembly language into machine language

Date: 30 / 11 / 2021 **Signature of faculty in-charge**