Practical-1

Pass-1:

Input file:

```
        START  200
        MOVER        AREG   ='5'
        MOVEM        AREG   A
LOOP    MOVER        AREG   A
        MOVER        CREG   B
        ADD     CREG   ='1'
        MOVER        AREG   A
        MOVER        CREG   B
        MOVER        AREG   A
        MOVER        CREG   B
        MOVER        AREG   A
        BC      ANY    NEXT
        LTORG
        MOVER        AREG   A
NEXT    SUB     AREG   ='1'
        BC      LT     BACK
LAST    STOP
        ORIGIN       LOOP+2
        MULT    CREG   B
        ORIGIN       LAST+1
A       DS      1
BACK    EQU     LOOP
B       DS      1
        END
```

Pass-1:

```
from os import sep, write
class Mnemonics:
    def __init__(self):
        self.AD = {
            "START": 1,
            "END": 2,
            "ORIGIN": 3,
            "EQU": 4,
            "LTORG": 5
        }
        self.RG = {
            "AREG": 1,
            "BREG": 2,
            "CREG": 3,
            "DREG": 4
        }
        self.DL = {
            "DC": 1,
            "DS": 2
        }
        self.IS = {
            "STOP": 0,
            "ADD": 1,
            "SUB": 2,
            "MULT": 3,
            "MOVER": 4,
            "MOVEM": 5,
            "COMP": 6,
            "BC": 7,
            "DIV": 8,
            "READ": 9,
            "PRINT": 10
        }
        self.CC = {
            "LT": 1,
            "LE": 2,
            "EQ": 3,
            "GT": 4,
            "GE": 5,
            "ANY": 6
        }

    def getClassType(self,string):
        if string in self.AD:
            return "AD"
        elif string in self.CC:
            return "CC"
        elif string in self.DL:
            return "DL"
        elif string in self.IS:
            return "IS"
        elif string in self.RG:
            return "RG"
        else:
            return ""
    def getMachineCode(self,string):
        if string in self.AD:
            return self.AD[string]
        elif string in self.CC:
            return self.CC[string]
        elif string in self.DL:
            return self.DL[string]
        elif string in self.IS:
            return self.IS[string]
        elif string in self.RG:
            return self.RG[string]
        else:
            return -1


class pass1:
    def __init__(self):
        self.lookup = Mnemonics()
        self.symbolTable = {}
        self.literalTable = {}
        self.litTableIndex = 0 #Used to append values in literal Table
        self.poolTable = [0]
        self.IC = [] #IC Intermediate Code
        self.location = 0
        self.litTabPtr = 0 #Literal Table Pointer
        self.InputFile = open("input.txt","r")
        self.literalTableFile = open("literalTable.txt","w")
        self.symbolTableFile = open("symbolTable.txt","w")
        self.poolTableFile = open("poolTable.txt","w")
    self.ICFile = open("intermediateCode.txt","w")



def calculateLocation(self,string):
    if "+" in string:
        string = string.split("+")
        return self.symbolTable[string[0]] + int(string[1])
    elif "-" in string:
        string = string.split("-")
        return self.symbolTable[string[0]] - int(string[1])
    else:
        return self.symbolTable[string]


def parseFile(self):
    for line in self.InputFile.readlines():
        self.IC.append([])
        line = line.strip("\n")
        line = line.split("\t")



        # For label
        if line[0] != "":
            if line[0] in self.symbolTable:
                self.symbolTable[line[0]] = self.location #Set location
            else:
                self.symbolTable[line[0]] = self.location #add label,loc


        # For Opcode
        if line[1] == "START":
```

```
        self.location = int(line[2])
        self.IC[-1].append(('AD',1))
        self.IC[-1].append(("C",int(line[2])))
    elif line[1] == "LTORG":
        # literalKeys = list(self.literalTable.keys())
        for i in range(self.poolTable[-1],len(self.literalTable)):
            self.literalTable[i][1] = self.location
            self.IC[-1].append(("DL",1))
            self.IC[-1].append(("C",self.literalTable[i][0]))
            self.IC[-1].append(self.location)
            self.location += 1
            self.litTabPtr += 1
            if i < len(self.literalTable) - 1:
                self.IC.append([])
        self.poolTable.append(self.litTabPtr)
    elif line[1] == "ORIGIN":
        self.location = self.calculateLocation(line[2])
        self.IC[-1].append(("AD",3))
        self.IC[-1].append(("C",self.location))
    elif line[1] == "EQU":
        newlocation =  self.calculateLocation(line[2])
        self.symbolTable[line[0]] = newlocation
        self.IC[-1].append(("AD",4))
        self.IC[-1].append(("C",newlocation))
    elif line[1] == "DC":
        self.IC[-1].append(("DL",1))
        self.IC[-1].append(("C",int(line[2])))
        self.IC[-1].append(self.location)
        self.location += 1
```

```
elif line[1] == "DS":
    self.IC[-1].append(("DL",2))
    self.IC[-1].append(("C",int(line[2])))
    self.IC[-1].append(self.location)
    self.location += int(line[2])
elif line[1] == "STOP":
    self.IC[-1].append(("IS",0))
    self.IC[-1].append(self.location)
    self.location += 1
elif line[1] == "END":
    self.IC[-1].append(("AD",2))
    if self.litTabPtr != len(self.literalTable):
        # literalKeys = list(self.literalTable.keys())
        for i in range(self.poolTable[-1],len(self.literalTable)):
            self.IC.append([])
            self.literalTable[i][1] = self.location
            self.IC[-1].append(("DL",1))
            self.IC[-1].append(("C",self.literalTable[i][0]))
            self.IC[-1].append(self.location)
            self.location += 1
            self.litTabPtr += 1
        self.poolTable.append(self.litTabPtr)
elif line[1] == "PRINT":
    self.IC[-1].append(("IS",10))
    symTabKeys = list(self.symbolTable.keys())
    self.IC[-1].append(("S",symTabKeys.index(line[2])))
    self.IC[-1].append(self.location)
    self.location += 1
elif line[1] == "READ":
    self.IC[-1].append(("IS",9))
    self.symbolTable[line[2]] = None
    symTabKeys = list(self.symbolTable.keys())
    self.IC[-1].append(("S",symTabKeys.index(line[2])))
    self.IC[-1].append(self.location)
    self.location += 1
elif line[1] == "BC":
    self.IC[-1].append(("IS",7))
    classType = self.lookup.getClassType(line[2])
    machineCode = self.lookup.getMachineCode(line[2])
    self.IC[-1].append((classType,machineCode))
    if line[3]  not in self.symbolTable:
        self.symbolTable[line[3]] = None
    symTabKeys = list(self.symbolTable.keys())
    self.IC[-1].append(("S",symTabKeys.index(line[3])))
    self.IC[-1].append(self.location)
    self.location += 1
else:
    #For Opcode
    classType = self.lookup.getClassType(line[1])
    machineCode = self.lookup.getMachineCode(line[1])
    self.IC[-1].append((classType,machineCode))

    #For Operand1
    classType = self.lookup.getClassType(line[2])
    machineCode = self.lookup.getMachineCode(line[2])
    self.IC[-1].append((classType,machineCode))

    #For Operand2
```

```
                if "=" in line[3]:
                    constant = line[3].strip("=")
                    constant = int(constant.strip("'"))
                    self.literalTable[self.litTableIndex] = [constant,None]
                    self.IC[-1].append(("L",self.litTableIndex))
                    self.IC[-1].append(self.location)
                    self.litTableIndex += 1
                else:
                    if line[3] in self.symbolTable:
                        symbolTableKeys = list(self.symbolTable.keys())
                        self.IC[-1].append(("S",symbolTableKeys.index(line[3])))
                        self.IC[-1].append(self.location)
                    else:
                        self.symbolTable[line[3]] = None
                        symbolTableKeys = list(self.symbolTable.keys())
                        self.IC[-1].append(("S",symbolTableKeys.index(line[3])))
                        self.IC[-1].append(self.location)

                self.location += 1

        self.printLiteralTable()
        self.printSymbolTable()
        self.printPoolTable()
        self.printIntermdeiateCode()

    def printLiteralTable(self):
        tab = "\t"
        endline = "\n"
        print("\nLITERAL TABLE:")
        for item in range(len(self.literalTable)):
            line = str(item) + tab + str(self.literalTable[item][0]) + tab
+str(self.literalTable[item][1]) + endline;
            print(line,end="")
            self.literalTableFile.write(line)
        self.literalTableFile.close()
        print("\n")



    def printSymbolTable(self):
        tab = "\t"
        endline = "\n"
        print("\nSYMBOL TABLE:")
        for index,item in enumerate(self.symbolTable):
            line = str(index) + tab + str(item) + tab + str(self.symbolTable[item]) + endline
            print(line,end="")
            self.symbolTableFile.write(line)
        self.symbolTableFile.close()
        print("\n")

    def printPoolTable(self):
        tab = "\t"
        endline = "\n"
        print("\nPOOL TABLE:")
        for item in range(len(self.poolTable)):
            print(self.poolTable[item])
            self.poolTableFile.write(str(self.poolTable[item]) + endline)
        self.poolTableFile.close()

    def printIntermdeiateCode(self):
        tab = "\t"
        endline = "\n"
        print("\nIntermediate Code:")
        for item in self.IC:
            line = ""
            for i in range(len(item)):
                line += str(item[i])
                if i != len(item):
                    line += tab
            line += endline
            print(line,end="")
            self.ICFile.write(line)
        self.ICFile.close()


obj = pass1()
obj.parseFile()
```

```
#Output:
Literal table:
0       5       211
1       1       212
2       1       219

Symbol table:
0       A       217
1       LOOP    202
2       B       218
3       NEXT    214
```

```
4       BACK    202

5       LAST    216


Pool table:

0

2

3


Intermediate code:

('AD', 1)       ('C', 200)

('IS', 4) ('RG', 1)     ('L', 0)  200

('IS', 5) ('RG', 1)     ('S', 0)  201

('IS', 4) ('RG', 1)     ('S', 0)  202

('IS', 4) ('RG', 3)     ('S', 2)  203

('IS', 1) ('RG', 3)     ('L', 1)  204

('IS', 4) ('RG', 1)     ('S', 0)  205

('IS', 4) ('RG', 3)     ('S', 2)  206

('IS', 4) ('RG', 1)     ('S', 0)  207

('IS', 4) ('RG', 3)     ('S', 2)  208

('IS', 4) ('RG', 1)     ('S', 0)  209

('IS', 7) ('CC', 6)     ('S', 3)  210

('DL', 1)       ('C', 5)  211

('DL', 1)       ('C', 1)  212

('IS', 4) ('RG', 1)     ('S', 0)  213

('IS', 2) ('RG', 1)     ('L', 2)  214

('IS', 7) ('CC', 1)     ('S', 4)  215

('IS', 0) 216

('AD', 3)       ('C', 204)

('IS', 3) ('RG', 3)     ('S', 2)  204
```

```
('AD', 3)       ('C', 217)
('DL', 2)       ('C', 1)  217
('AD', 4)       ('C', 202)
('DL', 2)       ('C', 1)  218
('AD', 2)
('DL', 1)       ('C', 1)  219
```

Pass-2:

```
import re
pattern = re.compile(r'\(\'([A-Z]{,2})\',\s(\d+)\)')

tab = "\t"

class pass2:
    def __init__(self):
        self.ICFile = open("intermediateCode.txt",mode='r')
        self.literalTableFile = open("literalTable.txt",mode='r')
        self.symbolTableFile = open("symbolTable.txt",mode='r')
        self.outputFile = open("output.txt",mode='w')
        self.literalTable = {}
        self.symbolTable= {}

    def convertToString(self,string):
        string = str(string)
        if len(string) == 1:
            return "00" + string
        elif len(string) == 2:
            return "0" + string
        elif len(string) == 3:
            return string

    def readSymbolTable(self):
        print("\nSymbol Table:")
        for line in self.symbolTableFile.readlines():
            line = line.split("\t")
            index = int(line[0])
            location = int(line[2])
            self.symbolTable[index] = location
            print(index,location,sep="\t")
        print("\n")

    def readLiteralTable(self):
        print("\nLiteral Table:")
        for line in self.literalTableFile.readlines():
            line = line.split('\t')
            index = int(line[0])
            location = int(line[2])
            self.literalTable[index] = location
            print(index,location,sep="\t")
        print("\n")



    def parseFile(self):
        self.readLiteralTable()
        self.readSymbolTable()
        print("Machine Code:")
        print("LC\tOPCODE\tOP1\tOP2")
```

```
        for line in self.ICFile.readlines():
            line = line.strip("\n")
            line = line.split("\t")
            find = pattern.search(line[0])

            if find.group(1) == "IS" or find.group(1) == "DL":
                lineToParse = ""
                location = line[-2]
                lineToParse += location + tab

                if find.group(1) == "IS":
                    lineToParse += self.convertToString(find.group(2)) + tab

                    if find.group(2) == "10" or find.group(2) == "9":
                        find = pattern.search(line[1])
                        key = int(find.group(2))
                        lineToParse += "000" + tab + self.convertToString(self.symbolTable[key]) +
"\n"
                    elif find.group(2) == "0":
                        lineToParse += "000" + tab + "000" + "\n"
                    else:
                        find = pattern.search(line[1])
                        lineToParse += self.convertToString(find.group(2)) + tab

                        find = pattern.search(line[2])
                        if find.group(1) == "S":
                            key = int(find.group(2))
                            lineToParse += self.convertToString(self.symbolTable[key]) + "\n"
                        elif find.group(1) == "L":
                            key = int(find.group(2))
                            lineToParse += self.convertToString(self.literalTable[key]) + "\n"
                else:
                    if find.group(2) == "1":
                        lineToParse += "000" + tab + "000" + tab
                        find = pattern.search(line[1])
                        lineToParse += self.convertToString(find.group(2)) + "\n"
                    else:
                        lineToParse += "000" + tab + "000" + tab + "000" + "\n"
            else:
                continue
            print(lineToParse,end="")
            self.outputFile.write(lineToParse)
        self.outputFile.close()
        self.literalTableFile.close()
        self.symbolTableFile.close()


obj = pass2()
obj.parseFile()

#Output:
```

Output.txt

```
200     004     001     211
201     005     001     217
202     004     001     217
203     004     003     218
204     001     003     212
```

205	004	001	217
206	004	003	218
207	004	001	217
208	004	003	218
209	004	001	217
210	007	006	214
211	000	000	005
212	000	000	001
213	004	001	217
214	002	001	219
215	007	001	202
216	000	000	000
204	003	003	218
217	000	000	000
218	000	000	000
219	000	000	001

Practical-2:

Input.txt:

```
MACRO
INCR    &MEM_VAL, &INCR_VAL, &REG
MOVER       &REG, &MEM_VAL
ADD     &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
MACRO
INCR_M      &MEM_VAL=, &INCR_VAL=, &REG=
MOVER       &REG, &MEM_VAL
ADD     &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
MACRO
INCR_D      &MEM_VAL=, &INCR_VAL=, &REG=AREG
MOVER       &REG, &MEM_VAL
ADD     &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
START
INCR    A, B, AREG
INCR_M      MEM_VAL=A, INCR_VAL=B, REG=AREG
INCR_M      INCR_VAL=B, REG=AREG, MEM_VAL=A
INCR_D      MEM_VAL=A, INCR_VAL=B
INCR_D      INCR_VAL=B, MEM_VAL=A
INCR_D      INCR_VAL=B, MEM_VAL=A, REG=BREG
END
```

Pass-1:

```
import re

default = re.compile(r'&(\w+)(=)?(\w+)?')

endline = "\n"

tab = "\t"


class macroPass1:

    def __init__(self):

        self.macroNameTable = {} #hash map.key-name of macro, value is an array containing pp,kp,mdtp,kpdtp

        self.macroDefTable = [] #simple array

        self.macroDefTablePointer = 1

        self.keywordParamDefTable = [] #Tuples(name,value) -- KPDTAB

        self.keyParamDefTabPointer = 1 #kpdtp

        self.paramNameTable = {} # key-name of macro,value-an array containing name of params

        self.inputFile = open('input.txt',mode="r")

        self.ICpointer = 0

        self.ICFile = open('intermediateCode.txt',mode='w')

        self.macroDefTableFile = open('macroDefTable.txt',mode='w')

        self.macroNameTableFile = open('macroNameTable.txt',mode='w')

        self.keywordParamDefTableFile = open('keywordParamDefTable.txt',mode='w')

        self.paramNameTableFile = open('paramNameTable.txt',mode='w')


    #input:param_name,current_macro output:(P,index)

    def covertToTuple(self,param,current_macro):

        id = self.paramNameTable[current_macro].index(param) + 1

        return "(P," + str(id) + ")"


    def parseFile(self):
```

```
lines = self.inputFile.readlines()
inMacroDefinition = False
currentMacroName = None
for line in lines:
    line = line.strip('\n')
    line = line.split('\t')
    part_1 = line[0]
    if part_1 == "START":
        break
    self.ICpointer += 1
    part_2 = []
    if len(line) > 1:
        part_2 = line[1].split(', ')

    if part_1 == "MACRO":
        inMacroDefinition = True
        continue
    elif inMacroDefinition == True:
        self.paramNameTable[part_1] = []
        currentMacroName = part_1
        positionalParamCount = 0
        keywordParamCount = 0
        for param in part_2:
            find = default.search(param)
            self.paramNameTable[part_1].append(find.groups()[0])
            if find.group(2) == None:
                positionalParamCount += 1
            else:
                keywordParamCount += 1
```

```
                    if find.group(3) == None:
                        self.keywordParamDefTable.append((find.group(1),"----"))
                    else:
                        self.keywordParamDefTable.append((find.group(1),find.group(3)))
            inMacroDefinition = False
            if keywordParamCount == 0:
                self.macroNameTable[part_1] =
[positionalParamCount,keywordParamCount,self.macroDefTablePointer,0]
            else:
                self.macroNameTable[part_1] =
[positionalParamCount,keywordParamCount,self.macroDefTablePointer,self.keyParamDefT
abPointer]
            self.keyParamDefTabPointer += keywordParamCount
        elif part_1 == "MEND":
            self.macroDefTable.append([])
            self.macroDefTable[-1].append("MEND")
            self.macroDefTablePointer += 1
            currentMacroName = None
        else:
            self.macroDefTable.append([])
            self.macroDefTable[-1].append(part_1)
            for param in part_2:
                find = default.search(param)
                if find != None:
                    self.macroDefTable[-
1].append(self.covertToTuple(find.group(1),currentMacroName))
                else:
                    self.macroDefTable[-1].append(param)


            self.macroDefTablePointer += 1
        #Write the Intermediate Code file
        while self.ICpointer < len(lines):
            line = lines[self.ICpointer]
            self.ICFile.write(line)
            self.ICpointer += 1
        self.ICFile.close()
        self.inputFile.close()

    def writeKeywordParamDefTable(self):
        print("\nKeyword Parameter Default Table:")
        counter = 1
        for value in self.keywordParamDefTable:
            line = value[0] + tab + value[1] + endline
            self.keywordParamDefTableFile.write(line)
            print(counter,line,end="",sep="\t")
            counter += 1
        self.keywordParamDefTableFile.close()

    def writeMacroNameTable(self):
        line = "Name" + tab + "PP" + tab + "KP" + tab + "MDTP" + tab + "KPDTP" + endline
        print("\nMacro Name Table:")
        print(line,end="")
        self.macroNameTableFile.write(line)
        for key,value in self.macroNameTable.items():
            line = key
            for ele in value:
                line += tab + str(ele)
            line += endline
            print(line,end="")
            self.macroNameTableFile.write(line)
        self.macroNameTableFile.close()

    def writeParamNameTable(self):
        print("\nParameter Name Table:")
        for key,value in self.paramNameTable.items():
            line = key
            for param in value:
                line += tab + param
            line += endline
            print(line,end="")
            self.paramNameTableFile.write(line)
        self.paramNameTableFile.close()

    def writeMacroDefTable(self):
        print("\nMacro Definition Table:")
        counter = 1
        for value in self.macroDefTable:
            line = ""
            for item in value:
                line += item + tab
            line += endline
            print(counter,line,end="",sep="\t")
            self.macroDefTableFile.write(line)
            counter += 1
        self.macroDefTableFile.close()
```

#Output:

```
MacroName Table:
Name    PP      KP      MDTP    KPDTP
INCR    3       0       1       0
INCR_M          0       3       5       1
INCR_D          0       3       9       4

MacroDef Table:
MOVER           (P,3)   (P,1)
ADD     (P,3)   (P,2)
MOVEM           (P,3)   (P,1)
MEND
MOVER           (P,3)   (P,1)
ADD     (P,3)   (P,2)
MOVEM           (P,3)   (P,1)
MEND
MOVER           (P,3)   (P,1)
ADD     (P,3)   (P,2)
MOVEM           (P,3)   (P,1)
MEND

ParaName Table:
INCR    MEM_VAL         INCR_VAL        REG
INCR_M          MEM_VAL         INCR_VAL        REG
INCR_D          MEM_VAL         INCR_VAL        REG

ParaDef Table:
MOVER           (P,3)   (P,1)
```

```
ADD    (P,3)    (P,2)
MOVEM           (P,3)    (P,1)
MEND
MOVER           (P,3)    (P,1)
ADD    (P,3)    (P,2)
MOVEM           (P,3)    (P,1)
MEND
MOVER           (P,3)    (P,1)
ADD    (P,3)    (P,2)
MOVEM           (P,3)    (P,1)
MEND
```

Intermediate code:

```
START
INCR    A, B, AREG
INCR_M          MEM_VAL=A, INCR_VAL=B, REG=AREG
INCR_M          INCR_VAL=B, REG=AREG, MEM_VAL=A
INCR_D          MEM_VAL=A, INCR_VAL=B
INCR_D          INCR_VAL=B, MEM_VAL=A
INCR_D          INCR_VAL=B, MEM_VAL=A, REG=BREG
END
```

Pass-2:

```
import re
default = re.compile(r'&?(\w+)(=)?(\w+)?')
parameter = re.compile(r'\(P,(\d+)\)')

class macroPass2:
   def __init__(self):
        self.macroNameTable = {} #hash map. value is an array containing pp,kp,mdtp,kpdtp
        self.macroDefTable = [] #simple array
        self.keywordParamDefTable = {} #Key-Name of macro. Value- an array containing tuples(name,value) -- KPDTAB
        self.paramNameTable = {} # key-name of function,value-an array containing name of params
        self.ICFile = open('intermediateCode.txt',mode='r')
        self.macroDefTableFile = open('macroDefTable.txt',mode='r')
        self.macroNameTableFile = open('macroNameTable.txt',mode='r')
        self.keywordParamDefTableFile = open('keywordParamDefTable.txt',mode='r')
        self.paramNameTableFile = open('paramNameTable.txt',mode='r')
        self.output = []
        self.outputFile = open('output.txt',mode='w')

    def readMacroDefTable(self):
        self.macroDefTable.append([])
        for line in self.macroDefTableFile.readlines():
            line = line.strip('\n')
            line = line.split('\t')
            self.macroDefTable.append(line[:-1])

    def readMacroNameTable(self):
        skipFirstLine = False
        for line in self.macroNameTableFile.readlines():
            line = line.strip('\n')
            line = line.split('\t')
            if skipFirstLine == False:
                skipFirstLine = True
                continue
            else:
```

```
            self.macroNameTable[line[0]] = line[1:]

def readKeywordParamDefTable(self):
    lines = self.keywordParamDefTableFile.readlines()
    for macroName,value in self.macroNameTable.items():
        numOfKeywordParam = int(value[1])
        kpdtp = int(value[3])
        self.keywordParamDefTable[macroName] = {}
        for i in range(kpdtp,kpdtp + numOfKeywordParam):
            line = lines[i-1].strip('\n').split('\t')
            self.keywordParamDefTable[macroName][line[0]] = line[1]

def readParamNameTable(self):
    for line in self.paramNameTableFile.readlines():
        line = line.strip('\n')
        line = line.split('\t')
        self.paramNameTable[line[0]] = line[1:]



def createAPTAB(self,macroName):
    APTAB = []
    APTAB.append(self.paramNameTable[macroName])
    APTAB.append([])
    APTAB.append([])
    keywordParamDict = self.keywordParamDefTable[macroName]
    for param in APTAB[0]:
        if param in keywordParamDict:
            APTAB[1].append(keywordParamDict[param])
            if keywordParamDict[param] == "----":
                APTAB[2].append(None)
            else:
                APTAB[2].append(keywordParamDict[param])
        else:
            APTAB[1].append(None)
            APTAB[2].append(None)
    return APTAB

def printAPTAB(self,APTAB):
    print("APTAB:")
    for i in range(len(APTAB[2])):
        print(i+1,APTAB[2][i],sep="\t")
    print()

def tupleToParam(self,line:list,APTAB:list) -> list:
    result = []
    for part in line:
        find = parameter.search(part)
        if find == None:
            result.append(part)
        else:
            idx = int(find.group(1)) - 1
            result.append(APTAB[2][idx])
    return result



def parseFile(self):
    for line in self.ICFile.readlines():
        line = line.strip('\n')
    line = line.split('\t')
    part_1 = line[0]


    if part_1 not in self.macroNameTable:
        self.output.append(line)
    else:
        #Pass actual parameters
        part_2 = line[1].split(', ')
        APTAB = self.createAPTAB(part_1)
        for param in range(len(part_2)):
            find = default.search(part_2[param])


            if find.group(2) == None:
                APTAB[2][param] = find.group(1)
            else:
                idx = APTAB[0].index(find.group(1))
                APTAB[2][idx] = find.group(3)
        #write macro definition with actual parameters
        mdtp = int(self.macroNameTable[part_1][2])
        print(*line,sep="\t")
        self.printAPTAB(APTAB)
        for macroDefLine in self.macroDefTable[mdtp:]:
            if macroDefLine[0] == "MEND":
                break
            else:
                self.output.append(self.tupleToParam(macroDefLine,APTAB))
                print(*self.tupleToParam(macroDefLine,APTAB),sep="\t")
        print('\n')
self.writeOutputFile()
```

```
        self.keywordParamDefTableFile.close()
        self.macroDefTableFile.close()
        self.macroNameTableFile.close()
        self.paramNameTableFile.close()

    def writeOutputFile(self):
        for value in self.output:
            line = "\t".join(value)
            line += "\n"
            self.outputFile.write(line)
        self.outputFile.close
```

Calling :

```
from macroPass1 import macroPass1
from macroPass2 import macroPass2

pass1 = macroPass1()
pass2 = macroPass2()
print("Pass1:")
pass1.parseFile()
pass1.writeMacroNameTable()
pass1.writeParamNameTable()
pass1.writeMacroDefTable()
pass1.writeKeywordParamDefTable()
print("\nPass2:")
pass2.readMacroDefTable()
pass2.readMacroNameTable()
pass2.readKeywordParamDefTable()
pass2.readParamNameTable()
```

```
pass2.parseFile()
```

#Output:

Output.txt:

```
START
MOVER           AREG    A
ADD     AREG    B
MOVEM           AREG    A
MOVER           AREG    A
ADD     AREG    B
MOVEM           AREG    A
MOVER           AREG    A
ADD     AREG    B
MOVEM           AREG    A
MOVER           AREG    A
ADD     AREG    B
MOVEM           AREG    A
MOVER           AREG    A
ADD     AREG    B
MOVEM           AREG    A
MOVER           BREG    A
ADD     BREG    B
MOVEM           BREG    A
END
```

#Practical-3

Vb (pvg Program)

```
 Public Class Form1
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As  System.EventArgs)
Handles Button1.Click
TextBox3.Text = projectdll.add(Val(TextBox1.Text), Val(TextBox2.Text)) End Sub
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button2.Click
TextBox3.Text = projectdll.sub1(Val(TextBox1.Text), Val(TextBox2.Text)) End Sub
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button3.Click
TextBox3.Text = projectdll.mul(Val(TextBox1.Text), Val(TextBox2.Text)) End Sub
Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button4.Click
TextBox3.Text = projectdll.div(Val(TextBox1.Text), Val(TextBox2.Text)) End Sub
Private Sub Button5_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button5.Click
Me.Close() End Sub
End Class
```

Project DLL File

```
Public Module projectdll
Public Function add(ByVal n1 As Integer, ByVal n2 As Integer)
Dim RESULT As String
RESULT = n1 + n2
Return (RESULT)
End Function

End Function
Public Function mul(ByVal n1 As Integer, ByVal n2 As Integer)
 Dim RESULT As String
RESULT = n1 * n2
Return (RESULT)
End Function
Public Function div(ByVal n1 As Integer, ByVal n2 As Integer)
 Dim RESULT As String
RESULT = n1 / n2
Return (RESULT)
End Function
End Module
```

Practical-5

// program to simulate CPU Scheduling Algorithms: FCFS and SJF: (Using Switch Case):-

```
import java.util.*;
public class Main{
 public static void main(String args[]){

 int wt[],proc[],tat[],bst[],n,i,j,total=0;

 Scanner sc= new Scanner(System.in);

 System.out.println("Page scheduling MENU: ");

 System.out.println(" 1. Using FCFS?");
 System.out.println(" 2. Using SJF?");
 System.out.print("Your Choice==> ");
 int x= sc.nextInt();

 System.out.print("\nNo. of processes: ");
 n=sc.nextInt();
 proc = new int[n];
 wt=new int[n];
 bst=new int[n];
 tat= new int[n];
```

```
switch (x){

case 1:

System.out.println("Enter Cpu time: ");

for(i=0;i<n;i++){

System.out.print(" Process["+(i+1)+"]: ");

bst[i]=sc.nextInt();

proc[i]=i+1;

}

wt[0]=0;

for(i = 1;i<n;i++){

wt[i]=0;

for(j=0;j<i;j++){

wt[i]+=bst[j];

total+=wt[i];

}

}

System.out.println("\nProcess\t\tBT\tWT\tTAT");

System.out.println("-------------------------------------");

for(i=0;i<n;i++){

tat[i]=wt[i]+bst[i];

System.out.println("Proc["+proc[i]+"]\t\t"+bst[i]+"\t"+wt[i]+"\t"+tat[i]);

}


case 2:

System.out.println("Enter Cpu time: ");

for(i=0;i<n;i++){

System.out.print(" Process["+(i+1)+"]: ");

bst[i]=sc.nextInt();

proc[i]=i+1;
}
for(i=0;i<n;i++){
int pp=i;
for(j=i+1;j<n;j++){
if(bst[j]<bst[pp])
pp=j;}

int temp=bst[i];
bst[i]=bst[pp];
bst[pp]=temp;

temp=proc[i];
proc[i]=proc[pp];
proc[pp]=temp;
}
wt[0]=0;
for(i = 1;i<n;i++){
wt[i]=0;
for(j=0;j<i;j++){
wt[i]+=bst[j];
total+=wt[i];
}
}
System.out.println("\nProcess\t\tBT\tWT\tTAT");
System.out.println("-------------------------------------------");
for(i=0;i<n;i++){
tat[i]=wt[i]+bst[i];
System.out.println("Proc["+proc[i]+"]\t\t"+bst[i]+"\t"+wt[i]+"\t"+tat[i]);
}
```

```
 }
 }
}
```

#Output:

```
Page scheduling MENU:
 1. Using FCFS?
 2. Using SJF?
Your Choice==> 1

No. of processes: 4
Enter Cpu time:
 Process[1]: 10
 Process[2]: 20
 Process[3]: 30
 Process[4]: 40

Process                 BT       WT       TAT
------------------------------------
Proc[1]         10      0        10
Proc[2]         20      10       30
Proc[3]         30      30       60
Proc[4]         40      60       100

Page scheduling MENU:
 1. Using FCFS?
 2. Using SJF?
Your Choice==> 2
```

```
No. of processes: 4
Enter Cpu time:
 Process[1]: 10
 Process[2]: 21
 Process[3]: 15
 Process[4]: 5

Process                BT      WT      TAT
------------------------------------
Proc[4]         5       0       5
Proc[1]         10      5       15
Proc[3]         15      15      30
Proc[2]         21      30      51
```

// program to simulate CPU Scheduling Algorithms: Priority and Round Robin:-

// 2.Round Robin:-

```
import java.util.*;
public class Main {
public static void main(String args[]) {
Scanner s = new Scanner(System.in);
int wtime[],btime[],rtime[],num,quantum,total;
wtime = new int[10];
btime = new int[10];
rtime = new int[10];
System.out.print("Enter number of processes(MAX 10): ");
num = s.nextInt();
System.out.print("Enter burst time");

for(int i=0;i<num;i++) { System.out.print("\nP["+(i+1)+"]: "); btime[i] = s.nextInt(); rtime[i] =

btime[i]; wtime[i]=0; } System.out.print("\n\nEnter quantum: "); quantum = s.nextInt(); int rp =

num; int i=0; int time=0; System.out.print("0"); wtime[0]=0; while(rp!=0) { if(rtime[i]>quantum)

{

rtime[i]=rtime[i]-quantum;

System.out.print(" | P["+(i+1)+"] | ");

time+=quantum;

System.out.print(time);

}

else if(rtime[i]<=quantum && rtime[i]>0)

{time+=rtime[i];

rtime[i]=rtime[i]-rtime[i];

System.out.print(" | P["+(i+1)+"] | ");

rp--;

System.out.print(time);

}

i++;

if(i==num)

{

i=0;

}

}

}

}
```

#Output:

```
Enter number of processes(MAX 10): 4
Enter burst time
P[1]: 2


P[2]: 4


P[3]: 6


P[4]: 8



Enter quantum: 2
0 | P[1] | 2
 | P[2] | 4 | P[3] | 6 | P[4] | 8 | P[2] | 10
 | P[3] | 12 | P[4] | 14 | P[3] | 16
 | P[4] | 18 | P[4] | 20
```

Practical-4

```
import java.util.*;
import java.util.concurrent.Semaphore;
public class Main {

static int mutex=1;
static int database=1;
static int Read_Count=0;
static void Reader() throws Exception
{
while(true)
{
mutex=wait(mutex);
Read_Count=Read_Count+1;
if(Read_Count==1){
database=signal(database);
}
mutex=signal(mutex);
System.out.println(Read_Count+ " User Reading the Data.........");
mutex=wait(mutex);
Read_Count=Read_Count-1;
if(Read_Count==0)
{
database=signal(database);
}
mutex=signal(mutex);
System.out.println("Reading Finished!!!!!!");
break;
}
}
static int wait(int mutex)
{
while(mutex<=0)
break ;
mutex=mutex-1;
return mutex;
}
static int signal(int database)
{
database=database+1;
return database;
}
static void Writer() throws Exception
{
while(true)
{
database=wait(database);
System.out.println("Writing on the database......");
database=signal(database);
System.out.println("Writing Finished!!!!!.");
break;
}
}
public static void main(String[] args)throws Exception {
Writer();
Reader();
Reader();
}
}
```

#Output

```
Writing on the database......
Writing Finished!!!!!.
1 User Reading the Data.........
Reading Finished!!!!!!
1 User Reading the Data.........
Reading Finished!!!!!!
```

Practical-7

```
//  Program to simulate Page replacement algorithm: FIFO LRU and Optimal:-

// 1.FIFO And LRU:-

import java.util.*;
public class Main {
 public static void main(String[] args) {
 pr();
 }
 static void pr(){
 Scanner sc = new Scanner(System.in);
 System.out.println("Page Replacement : ");
 System.out.println("Enter 1 for FIFO ");
 System.out.println("Enter 2 for LRU ");
 System.out.printf("Enter Choice : ");
 int x = sc.nextInt();
 System.out.printf("Length of String : " );
 int n = sc.nextInt();

 int fr = 3;
 int ref[] = new int[n];
 for (int i = 0; i < n; i++){
 ref[i] = sc.nextInt();
 }
 // FIFO
 HashMap<Integer,Integer> map = new HashMap<>();
 ArrayList<ArrayList<Integer>> arr = new ArrayList<>();
```

```
for(int i = 0 ; i <= n ; i++){
arr.add(new ArrayList<>());
}

for(int i = 0 ; i < fr ; i++){
arr.get(0).add(-1);

}
int ct = 1;
int hit = 0;
if(x == 1 && n > 0){
int indx= 0;
for(int i = 1 ; i <= n ; i++){
int curr = ref[i-1];
arr.get(i).addAll(arr.get(i-1));
if(!map.containsKey(curr)){
if(indx < fr) arr.get(i).set((indx),ref[i-1]);
else{
int min = Integer.MAX_VALUE;
int temp = 0;
for(int j : map.keySet()){
if(map.get(j) < min){
min = map.get(j);
temp = j;
}
}

for(int j = 0 ; j < fr ; j++){
if(arr.get(i).get(j) == temp){
```

```
arr.get(i).set(j,curr);
break;
}
}
map.remove(temp);
}
map.put(ref[i-1],ct++);
indx++;
}else{
hit++;
}
}
}else if(x == 2 && n > 0 ){

//LRU
int indx= 0;
for(int i = 1 ; i <= n ; i++){
int curr = ref[i-1];
arr.get(i).addAll(arr.get(i-1));
if(!map.containsKey(curr)){
if(indx < fr) arr.get(i).set(indx,ref[i-1]);
else{
int min = Integer.MAX_VALUE;
int temp = 0;
for(int j : map.keySet()){
if(map.get(j) < min){
min = map.get(j);
temp = j;
}
```

```
}
for(int j = 0 ; j < fr ; j++){
if(arr.get(i).get(j) == temp){
arr.get(i).set(j,curr);
break;
}
}
map.remove(temp);
}
indx++;
}else{
hit++;
}
map.put(ref[i-1],ct++);
}
}
// Output
System.out.println();
for(int i = 0 ; i <= n ; i++){
for(int j = 0 ; j < fr ; j++){
System.out.printf(arr.get(i).get(j) + " ");
}
System.out.println();
}
System.out.println("Total Page Fault : " + (n - hit));
System.out.println("Total Page Hit : " + hit);

sc.close();
}
```

```
}
```

#Ouput:

```
Page Replacement :
Enter 1 for FIFO
Enter 2 for LRU
Enter Choice : 1
Length of String : 5
1
2
3
4
5

-1 -1 -1
1 -1 -1
1 2 -1
1 2 3
4 2 3
4 5 3
Total Page Fault : 5
Total Page Hit : 0
```

// 2. Optimal Page Replacement Algorithm:-

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
public class Main {
public static void main(String[] args) throws IOException
{
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
int rl, fr, pt = 0, hit = 0, fault = 0;
boolean isFull = false;
int buffer[];
int reference[];
int mem_layout[][];
System.out.println("\nENTER THE NUMBER OF FRAMES: ");
fr = Integer.parseInt(br.readLine());
System.out.println("\nENTER THE LENGTH OF REFERENCE STRING: ");
rl = Integer.parseInt(br.readLine());
reference = new int[rl];
mem_layout = new int[rl][fr];
buffer = new int[fr];
for(int j = 0; j < fr; j++)
buffer[j] = -1;
System.out.println("\nENTER THE REFERENCE STRING: ");
for(int i = 0; i < rl; i++)
{
reference[i] = Integer.parseInt(br.readLine());
}
System.out.println();
for(int i = 0; i < rl; i++)
{
int search = -1;
for(int j = 0; j < fr; j++)
{
```

```
if(buffer[j] == reference[i])
{
search = j;
hit++;
break;
}
}
if(search == -1)
{
if(isFull)
{
int index[] = new int[fr];
boolean index_flag[] = new boolean[fr];
for(int j = i + 1; j < rl; j++)
{
for(int k = 0; k < fr; k++)
{
if((reference[j] == buffer[k]) && (index_flag[k] == false))
{
index[k] = j;
index_flag[k] = true;
break;
}
}
}
int max = index[0];
pt = 0;
if(max == 0)
max = 200;
for(int j = 0; j < fr; j++)
{
if(index[j] == 0)
index[j] = 200;
if(index[j] > max)
{
max = index[j];
pt = j;
}
}
}
buffer[pt] = reference[i];
fault++;
if(!isFull)
{
pt++;
if(pt == fr)
{
pt = 0;
isFull = true;
}
}
}
for(int j = 0; j < fr; j++)
mem_layout[i][j] = buffer[j];
}
for(int i = 0; i < fr; i++)
{
for(int j = 0; j < rl; j++)
```

```
System.out.printf("%3d ",mem_layout[j][i]);
System.out.println();
}
System.out.println("\nTOTAL NUMBER OF HIT: " + hit);
System.out.println("\nHIT RATIO: " + (float)((float)hit/rl));
System.out.println("\nTOTAL NUMBER OF PAGE FAULT: " + fault);
}
}
```

#Output:

```
ENTER THE NUMBER OF FRAMES:
4

ENTER THE LENGTH OF REFERENCE STRING:
5

ENTER THE REFERENCE STRING:
1
2
3
4
5

  1   1   1   1   5
 -1   2   2   2   2
 -1  -1   3   3   3
 -1  -1  -1   4   4
```

```
TOTAL NUMBER OF HIT: 0

HIT RATIO: 0.0

TOTAL NUMBER OF PAGE FAULT: 5
```

Practical-6

# Write a program to simulate Memory placement strategies – best fit, first fit, next fit and worst fit.

```
class memoryManagement_variable:
    def __init__(self):
        self.memoryBlocks = [int(i) for i in input("Enter sizes of the memory blocks: ").split("
")]
        self.numProcesses = int(input("Enter number of processes: "))
        self.processSizes = [int(i) for i in input("Enter the size of processes: ").split(" ")
        ]
        self.allocatedBlock = [-1] * self.numProcesses
        self.blockSizeRem = [self.memoryBlocks[i] for i in range(self.numProcesses)]


    def printTable(self):
        print('%-10s%-15s%-15s%-15s%-15s' %("Process","Process Size","Block
Number","Block Size","Unused Memory"))
        for i in range(self.numProcesses):
            if self.allocatedBlock[i] != -1:
                print('%-10i%-15i%-15i%-15i%-15i' %(i+1,self.processSizes[i],self.allocatedBlock[i]
+ 1,self.memoryBlocks[self.allocatedBlock[i]],self.blockSizeRem[self.allocatedBlock[i]]))
            else:
                print('%-10i%-15i%-15s%-15s%-15s' %(i+1,self.processSizes[i],"N/A","-","-"))
        print("\n")

class memoryManagement_fixed:
    def __init__(self):
        self.memoryBlocks = [int(i) for i in input("Enter sizes of the memory blocks: ").split(" ")]
        self.numProcesses = int(input("Enter number of processes: "))
        self.processSizes = [int(i) for i in input("Enter the size of processes: ").split(" ")
        ]
        self.allocatedBlock = [-1] * self.numProcesses
        self.memoryAllocated = [False] * self.numProcesses



    def printTable(self):
        print('%-10s%-15s%-15s%-15s' %("Process","Process Size","Block Number","Block Size"))
        for i in range(self.numProcesses):
            if self.allocatedBlock[i] != -1:
                print('%-10i%-15i%-15i%-15i' %(i+1,self.processSizes[i],self.allocatedBlock[i] + 1,self.memoryBlocks[self.allocatedBlock[i]]))
            else:
                print('%-10i%-15i%-15s%-15s' %(i+1,self.processSizes[i],"N/A","-"))
        print("\n")


class bestFitVariable(memoryManagement_variable):
    def __int__(self):
        memoryManagement_variable.__init__(self)


    def execute(self):


        for i in range(self.numProcesses):
            bestBlock = -1
            for block in range(len(self.memoryBlocks)):
                if self.blockSizeRem[block] >= self.processSizes[i]:
                    if bestBlock == -1:
                        bestBlock = block
                    elif self.blockSizeRem[bestBlock] > self.blockSizeRem[block]:
                        bestBlock = block

            if bestBlock != -1:
                self.allocatedBlock[i] = bestBlock
                self.blockSizeRem[bestBlock] -= self.processSizes[i]
        print("\nBEST FIT - Variable Size of Memory Block:")
        self.printTable()

class bestFitFixed(memoryManagement_fixed):
    def __init__(self):
        super().__init__()
    def execute(self):
        for i in range(self.numProcesses):
            bestBlock = -1
            for block in range(len(self.memoryBlocks)):
                if self.memoryBlocks[block] >= self.processSizes[i] and
self.memoryAllocated[block] == False:
                    if bestBlock == -1:
                        bestBlock = block
                    elif self.memoryBlocks[bestBlock] > self.memoryBlocks[block]:
                        bestBlock = block

            if bestBlock != -1:
                self.allocatedBlock[i] = bestBlock
                self.memoryAllocated[bestBlock] = True
        print("\nBEST FIT - Fixed size of memory block:")
        self.printTable()
```

```
class worstFitVariable(memoryManagement_variable):
    def __init__(self):
        super().__init__()

    def execute(self):
        for i in range(self.numProcesses):
            worstBlock = -1
            for block in range(len(self.memoryBlocks)):
                if self.blockSizeRem[block] >= self.processSizes[i]:
                    if worstBlock == -1:
                        worstBlock = block
                    elif self.blockSizeRem[worstBlock] < self.blockSizeRem[block]:
                        worstBlock = block

            if worstBlock != -1:
                self.allocatedBlock[i] = worstBlock
                self.blockSizeRem[worstBlock] -= self.processSizes[i]
        print("\nWORST FIT - Variable Size of Memory Block:")
        self.printTable()

class worstFitFixed(memoryManagement_fixed):
    def __init__(self):
        super().__init__()

    def execute(self):
        for i in range(self.numProcesses):
            worstBlock = -1
```

```
      for block in range(len(self.memoryBlocks)):
        if self.memoryBlocks[block] >= self.processSizes[i] and self.memoryAllocated[block] == False:
          if worstBlock == -1:
            worstBlock = block
          elif self.memoryBlocks[worstBlock] < self.memoryBlocks[block]:
            worstBlock = block

      if worstBlock != -1:
        self.allocatedBlock[i] = worstBlock
        self.memoryAllocated[worstBlock] = True
    print("\nWORST FIT - Fixed Size of Memory Block:")
    self.printTable()

class firstFitVariable(memoryManagement_variable):
  def __init__(self):
    super().__init__()

  def execute(self):
    for i in range(self.numProcesses):
      for block in range(len(self.memoryBlocks)):
        if self.blockSizeRem[block] >= self.processSizes[i]:
          self.allocatedBlock[i] = block
          self.blockSizeRem[block] -= self.processSizes[i]
          break
    print("\nFIRST FIT - Variable Size of Memory Block:")
    self.printTable()

class firstFitFixed(memoryManagement_fixed):
  def __init__(self):
```

```python
        super().__init__()

    def execute(self):
        for i in range(self.numProcesses):
            for block in range(len(self.memoryBlocks)):
                if self.memoryBlocks[block] >= self.processSizes[i] and self.memoryAllocated[block] == False:
                    self.allocatedBlock[i] = block
                    self.memoryAllocated[block] = True
                    break
        print("\nFIRST FIT - Fixed Size of Memory Block:")
        self.printTable()


while True:
    print("1>Best Fit",
          "2>Worst Fit",
          "3>First Fit",
          "4>Next Fit",
          "5>Exit",sep="\n",end="\n\n")
    choice = int(input("Enter a choice: "))
    if choice == 1:
        bf = bestFitVariable()
        bf.execute()
        bf = bestFitFixed()
        bf.execute()
    elif choice == 2:
        wf = worstFitVariable()
        wf.execute()
        wf = worstFitFixed()
        wf.execute()
    elif choice == 3:
        ff = firstFitVariable()
        ff.execute()
        ff = firstFitFixed()
        ff.execute()
    else:
        break
```

```
#Output:


# 1>Best Fit
# 2>Worst Fit
# 3>First Fit
# 4>Exit

# Enter sizes of the memory blocks:
Enter number of processes:
Enter the size of processes:
# BEST FIT -:
# Process   Process Size   Block Number   Block Size
# 1       100          2            170
# 2       50           1            200
# 3       400          3            500




# 1>Best Fit
# 2>Worst Fit
# 3>First Fit
# 4>Exit


# Enter a choice:

Enter sizes of the memory blocks:

Enter number of processes: Enter the size of processes:

# WORST FIT:

# Process   Process Size   Block Number   Block Size     Unused Memory

# 1       100          3            500          350

# 2       50          3            500          350

# 3       400          N/A          -            -



# 1>Best Fit

# 2>Worst Fit

# 3>First Fit

# 4>Exit


# Enter a choice:

Enter sizes of the memory blocks:

Enter number of processes:

Enter the size of processes:

# FIRST FIT :

# Process   Process Size   Block Number   Block Size     Unused Memory

# 1       100          1            200          50

# 2       50          1            200          50

# 3       400          3            500          100
```