# **Assignment 2**

**Aim :**

Design suitable data structures & implement the FIRST pass of a two-pass Macro processor.

**Theory :**

A Macro instruction is the notational convenience for the programmer. For every occurrence of macro, the whole macro body or macro block of statements gets expanded in the main source code. Thus Macro instructions make writing code more convenient.

Salient features of Macro Processor:

* Macro represents a group of commonly used statements in the source programming language.
* Macro Processor replaces each macro instruction with the corresponding group of source language statements. This is known as the expansion of macros.
* Using Macro instructions programmer can leave the mechanical details to be handled by the macro processor.
* Macro Processor designs are not directly related to the computer architecture on which it runs.
* Macro Processor involves definition, invocation, and expansion.

**Code :**

# Pass 1 implementation of 2 pass Macro Processor

# Author - Sumedh A. Kulkarni <sumedh.17u150@viit.ac.in>

import re

class MNTentry:

def \_\_init\_\_(self, name, num\_args, start, end = 0):

self.name = name

self.num\_args = num\_args

self.start\_index = start

self.end\_index = end

def printEntry(self):

print("%s\t\t%s\t\t%s\t\t%s" %

(self.name,

self.num\_args,

self.start\_index,

self.end\_index))

class MacroProcessor:

def \_\_init\_\_(self, inpfile, outfile):

self.input\_file = inpfile

self.output\_file = outfile

self.intermidiate\_file = None

self.lines = []

self.mnt = [] # Macro Name Table

self.mdt =[] # Macro Defination Table

self.mdt\_index = 0

self.error\_flag = False

self.error\_msg = ""

def passOne(self):

line\_num = 0

with open(self.input\_file, 'r') as infile:

with open("intermidiate\_file.asm", 'w') as self.intermidiate\_file:

self.lines = infile.readlines()

while line\_num < len(self.lines):

line = self.lines[line\_num].strip()

cmd = line.split()

if cmd[0] == "END":

break

elif cmd[0] != "MACRO":

self.intermidiate\_file.write(self.lines[line\_num])

line\_num+=1

else:

line\_num = self.processMacro(line\_num)

if self.error\_flag:

break

if not self.error\_flag:

print("SUCCESS!!")

print("Intermidiate file is generated.")

self.printMNT()

self.printMDT()

else:

print("ERROR!!\n", self.error\_msg)

def printMNT(self):

print("===== MNT =====")

print("Macro Name\tNo. Args\tStart Index\tEnd Index")

for entry in self.mnt:

entry.printEntry()

print()

def printMDT(self):

ind = 0

print("===== MDT ======")

for cmd in self.mdt:

print(ind, cmd)

ind+=1

print()

def processMacro(self, line\_num) -> int:

try:

cmd = re.split(' |,', self.lines[line\_num].strip())

m\_name = cmd[1]

n\_args = 0

formal\_t\_positional = {}

if len(cmd) > 2:

n\_args = len(cmd)-2

position = 1

for arg in cmd[2:]:

formal\_t\_positional[arg] = f"#{position}"

position+=1

self.mnt.append(MNTentry(m\_name, n\_args, self.mdt\_index))

line\_num+=1

while line\_num < len(self.lines):

cmd = re.split(' |,', self.lines[line\_num].strip())

if(cmd[0] == "MEND"):

self.mnt[-1].end\_index = self.mdt\_index-1

# self.mdt\_index+=1

line\_num+=1

break

elif(cmd[0] == "MACRO"):

self.processMacro(line\_num)

else:

for i in range(len(cmd)):

if cmd[i] in formal\_t\_positional.keys():

cmd[i] = formal\_t\_positional[cmd[i]]

self.mdt.append(cmd[0]+" "+",".join(cmd[1:]))

self.mdt\_index+=1

line\_num+=1

return line\_num

except Exception as e:

self.error\_flag = True

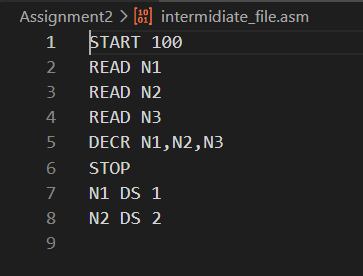
self.error\_msg = e

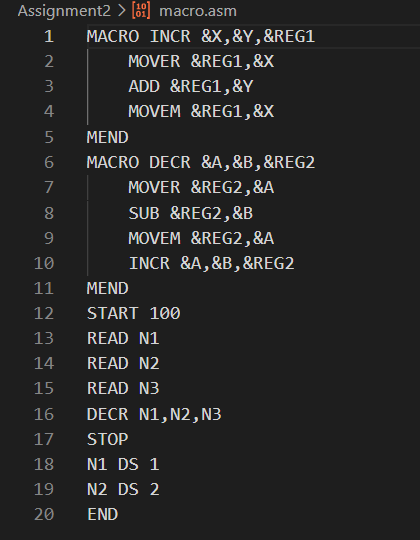
return -1

if \_\_name\_\_ == "\_\_main\_\_":

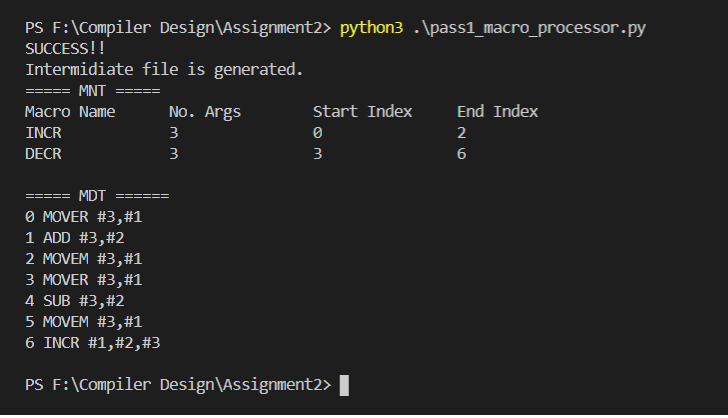
mp = MacroProcessor("macro.asm", "macro\_out.asm")

mp.passOne()

**Input: Intermediate File (output) :**

****

**Output (MNT, MDT) :**



**Conclusion :**

Pass 1 of 2 Pass Macro Processor was implemented which generates all the required data structures like MNT, MDT, and IR file.