UNIT 4 : MACRO AND MACRO PROCESSOR

Definition:

A macro is a unit of specification for program generation through expansion
A macro name is an abbreviation, which stands for some related lines of code.
Macros are useful for the following purpose
- To simplify & reduce amount of repetitive coding
- To reduce errors caused by repetitive coding
- To make an assembly program more readable

STRUCTURE OF MACRO

MACRO ← Start of macro definition

<MACRO NAME> <FORMAL PARAMETERS>

-
-
-
-
} Macro Body

MEND ← End of macro definition

A macro consists of name, set of formal parameters and body of code
the use of macro name with set of actual parameters is replaced by some code generated by its body. This is called macro expansion
Macro allows programmer to define pseudo operations, typically operations that are generally desirable, are not implemented as a part of the processor instruction, and can be implemented as sequence of instructions.

MACRO CALL

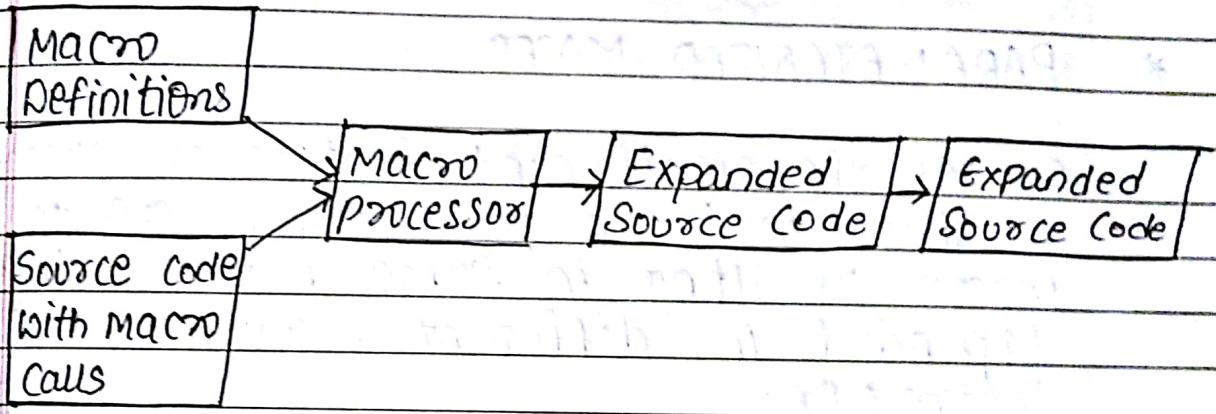
A macro is called by writing the macro name in the mnemonic field of the assembly statement.

<MACRO NAME> [ACTUAL PARAMETERS]

MACRO EXPANSION

A macro call leads to macro expansion. During macro expansion, the macro statement is replaced by sequence of assembly statements.

Fig representation



Example: Source

```
Macro
A    1,data
A    2,data
A    3,data
MEND
      :
      :
INCR
      :
      :
data  DC    F'5'
      :
      :
END
```

Expanded Source

```
A    1,Data
A    2,Data
A    8,Data
data DC   F'5'
```

Every macro begins with MACRO keyword at the beginning and ends with MEND (end of macro). When ever a macro is called the entire code is substituted in the program where it is called So the resultant of macro code is an expanded code.

* PARAMETERIZED MACRO

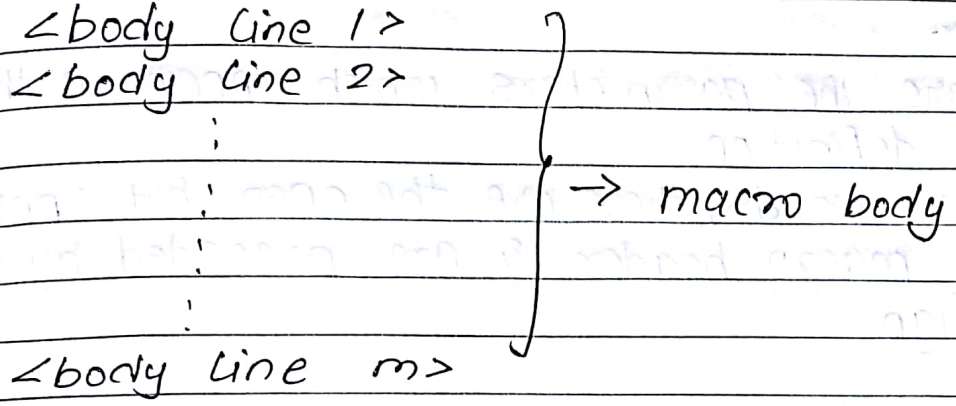
One of the significant feature of macro is that we can pass parameters to a macro resulting in same macro being expanded in different sequence of instruction.

Such type of macros are called "parameterized macros"

Macros may have any number of parameters as long as they fit in one line. Parameter names are local symbol, which are known within the macro only. Outside the macro they have no meaning

SYNTAX

MACRO
<macro name> < parameter 1> ... < parameter n)



MEND

Valid macro arguments are:

i) Arbitrary seq of printable characters, not containing blanks, tabs, commas or semicolons.
ii) Quoted strings (in single / double quotes)
iii) Single printable characters, preceded by "!" as an escape character
iv) Character sequences, enclosed in literal brackets <.....>, which may be arbitrary sequences of valid macro blanks, commas & semicolons
v) Arbitrary sequences of valid macro arguments
vi) Expression preceded by a '%' character

Parameters (or Arguments) are of two types:

a) Formal Parameters

These are the parameters which appear in the macro definition.
In other words, these are the ones that appear in the macro header & are preceded by an '&' sign.

b) Actual Parameters

These are the parameters which appear in macro call
Here, the values/variables are actually passed to the macros during macro call.

Example

Source	Expanded Source
MACRO	:
INCR &arg (Macro name with argument)	:
A 1, &arg	:
A 2, &arg	:
A 3, &arg	:
MEND	A 1, data1
:	A 2, data1
:	A 3, data1
INCR data1 (use of data1 as operand)	:
:	:
:	A 1, data2
INCR data2 (use of data2 as operand)	A 2, data2
:	A 3, data2
data1 DC F'5'	:
data2 DC F'6'	:
:	data1 DC F'5'
END	data2 DC F'6'

PARAMETER PASSING TECHNIQUE

a) By Position

Parameters correspon do their poisitions i-e here poisition is used to map actual parameters to formal parameters.

Example:

```
INCR  A,B,C
```

'A' replaces first dummy argument
'B' " Second " "
'C' " third " "

b) BY KEYWORD

- Here actual parameters correspond to formal parameters by use of keywords
- This allows reference to dummy args by name as well as by position

Eg:

- INCR &arg1 = A , &arg3 = C , &arg2 = 'B'
- INCR &arg1 = &arg2 = A , &arg2 = 'C'

The number of arguments, passed to a macro, can be less (but not greater) than the number of its formal parameters.
If an argument is omitted, the corresponding formal parameter is replaced by an empty string.
If other arguments than the last ones are to be omitted, they can be represented by commas.
Macro Parameters support code reuse, allowing one macro definition to implement multiple algorithms.

NESTED MACROS:

There are two ways of using nested macro facility:

- Nested Macro Call
- Nested Macro Definition

Nested Macro Call

Macro bodies may also contain macro calls, and so may the bodies of those called macros and so forth. If a macro call is seen throughout the expansion of a macro, the assembler starts immediately with the expansion of called macro. For this, its expanded body lines are simply inserted into the expanded macro body of the calling macro, until the called macro is completely expanded.

Then the expansion of the calling macro is continued with the body line following the nested macro call

Example:

```
INSIDE MACRO
SUBB A,B3
ENDM
OUTSIDE MACRO
MOV A,#42
INSIDE
MOV R7,A
ENDM
```

In the body of the macro OUTSIDE, the macro INSIDE is called. If OUTSIDE is called, one gets something

```
MOV A,#42
SUBB A,B3
MOV R7,A
```

NESTED MACRO DEFINITION

A macro body may also contain further macro definitons. However, these nested macros definitions aren't valid until the enclosing macro has been expanded. That means, the enclosing macro must have been called, before the nested macros can be called.

CONDITIONAL MACRO EXPANSION

Conditional macro expansion means different sequence of instruction can be inserted into the macro body using the same macro definition, based on evaluation of certain conditions.

For conditional macro expansion, two macro pseudo opcodes are used -

1) AIF 2) AGO

1) AIF

It is used to evaluate a condition & depending on its result, the flow of control inside the macro body is altered

SYNTAX:

AIF <condition> <label>

2) AGO

It is an unconditional branching statement for the macro processor

SYNTAX:

AGO <label>

RECURSIVE MACROS:

If macros are calling themselves, one speaks of recursive macro calls. In other words, recursive macros are special cases of nested macros calls where we make a call from within a macro to macro itself.

In this case, there must be some stop criterion, to prevent the macro of calling itself over & over until the assembler is out of memory. It uses stacks for maintaining status of macro calls

Conditional macro expansion must be used carefully; if a proper condition is not given to terminate the recursion, the program may go into infinite recursive loop. AS a result all the available stack space may get exhausted right at the time of processing the macros, which is not desired at all.
Example:

Definition of Macro ADD1

```
MACRO
ADD1  &arg
 L    1,&arg
 A    1,=F'1'
 ST   1,&arg
MEND
```

Definition of Macro ADDS

```
MACRO
ADDS &arg1,&arg2
     &arg3
ADD1  &arg1
ADD1  &arg2
ADD1  &arg3
MEND
```

MACRO PROCESSOR

An assembler does not recognize the macro call. So, the macro processor works as a pre processor & expands the macro calls by replacing them with the corresponding sequence of instruction.

FUNCTIONS:

1] RECOGNIZE MACRO DEFINITION:
MACRO & MEND pseudo ops are used to identify the start & end of the macro respectively.

2] SAVE MACRO DEFINITION:
A macro processor uses Macro Name Table (MNT) & Macro Definition Table (MDT) to store definition of macros used in the program.

3] RECOGNIZE CALLS:
The processor recognize macro calls that appears as operation mnemonics (first token of a statement)

4] EXPAND THE MACRO CALLS:

MNT & MDT are used to get the stored definition of macro being called.

DESIGN OF A MACRO PROCESSOR

STEP I

SPECIFICATION OF PROBLEM

The design of a macro processor is also divided into two passes because of forward reference problem

In the first pass, (a macro processor is expected to)

- Examine every op-code will save all macro definition in MDT
- Save a copy of input text minus macro definition on secondary storage (ex magnetic tape) for use in Pass II
- The MNT will also get prepared along with MDT in PASS-I

In the second pass,

- Examines every mnemonic and replace each macro name with the appropriate text from macro definition.

STEP 2:

Specification of DATA STRUCTURES

Some of the basic tables used in the design of a macro processor are as follows:

- Macro Name Table (MNT)
- Macro Definition Table (MDT)
- Argument List Array (ALA)

DATA STRUCTURE FOR PASS I:

- The input source program deck
- The output macro source deck copy for use by Pass II
- The Macro Name Table (MNT): It is used to store macro names with corresponding index to MDT
- The Macro Name Table Counter (MNTC): used to indicate next available entry in MNT.
- The Macro Definition Table (MDT): It is used to store body for the macro definitions.
- The Macro Definition Table Counter (MDTC): used to indicate next available entry in MDT
- The Argument List Array (ALA) is used to substitute index markers for dummy arguments before storing a macro definition.

DATA STRUCTURE FOR PASS 2:

- The copy of the input macro source program.
- The output expanded source program to be used as input to the assembler
- The MDT created in Pass I
- The Macro Definition Table Pointer (MDTP): used to indicate next line of text to be used during macro expansion
- The MNT created in Pass - I
- The ALA used to substitute the macro call argument for the index markers in the stored macro definition.

STEP 3

SPECIFY THE FORMAT OF DATA STRUCTURE.

The ALA is used to determine both pass-I & pass - II but for somewhat reverse functions Consider the example with macro INR, the stored definition would be:

MDT

```
        :
&LAB    INCR    &ARG1,&ARG2,&ARG3
#0      A       1,#1
        A       2,#2
        A       3,#3
        MEND
        :
```

ALA (8 bytes per entry)

Index	Argument	
0	"Loop1bbb"	
1	"Data1bbb"	
2	"Data2bbb"	
3	"Data3bbb"	b denotes blank character

ALA

Index	Argument	
0	"bbbbbbb"	(all blanks)
1	"DATA3bbb"	
2	"DATA2bbb"	
3	"DATA1bbb"	

Macro definition table:

This a table of text lines, if i/p is from 80 column cards the MDT can be the table with 80 byte string as entries

Consider the example, the MDT would look like this:

MDT (80 bytes per entry)

Index	Card	
⋮	⋮	
15	&LAB	INCR &ARG1,&ARG2,&ARG3
16	#0	A 1,#1
17		A 2,#2
18		A 3,#3
19		MEND

Macro Name Table:

The MNT serves a function very similar to that of MOT & POT of assemblers Each entry consits of a character string (the macro name) & a pointer (index) to that entry in the MDT that corresponds to the beginning of macro definition:

MNT

Index	8 byteo NAME	4 byteo MDT index
:	:	:
3	"INCRbbb"	15

STEP 4:

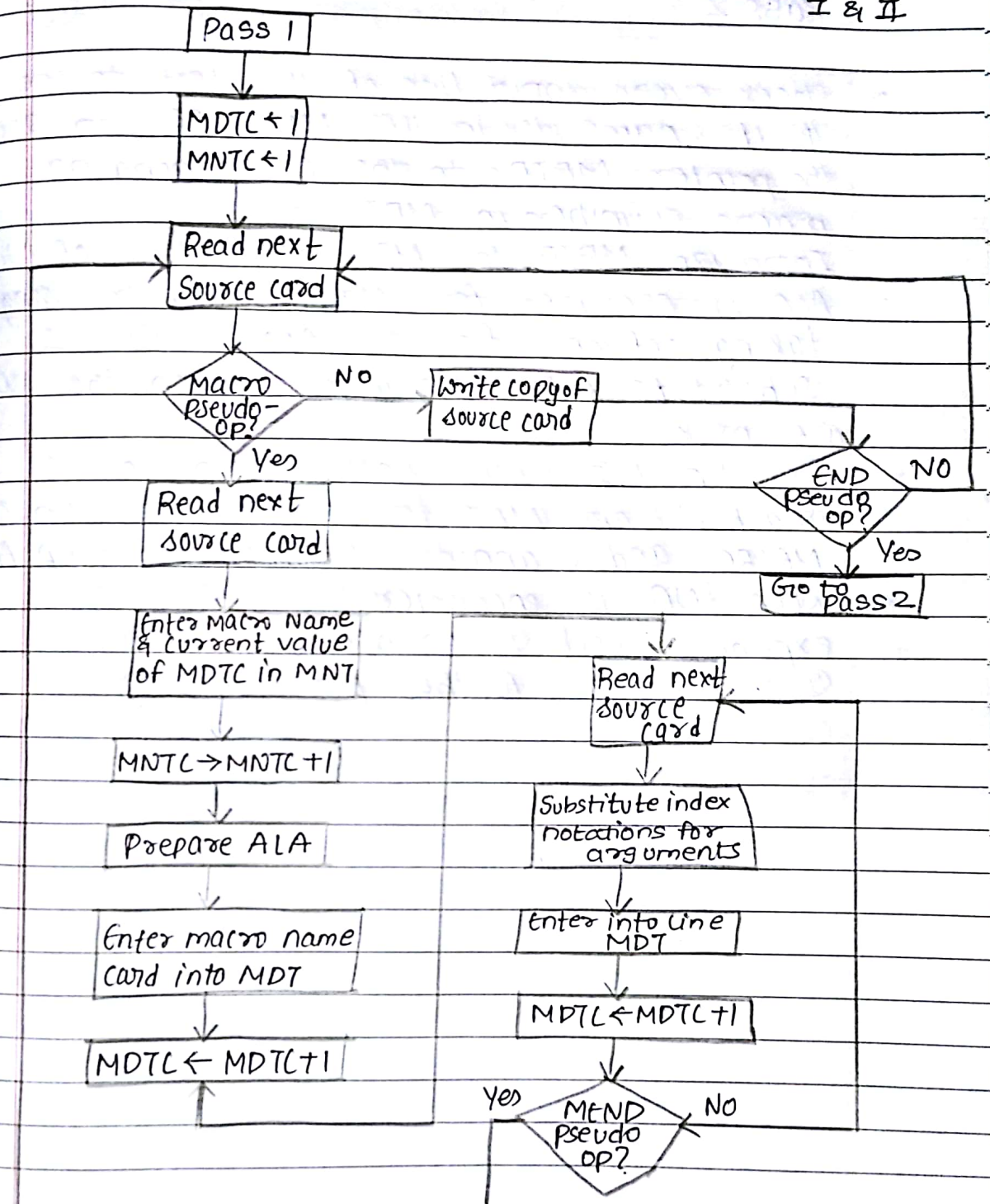
SPECIFY THE ALGORITHM FOR PASSI

- Check every input text line, if it is a MACRO pseudo-op: if Yeo go to nextstep or check further
- Save entire macro definition along with MNAME & MEND pseudo-ops in MDT.
- MNAME also entered in the MNT along with the pointer to the first location of MDT entry
- Continue this until entire body of macro is saved, if MEND pseudo-op follows stop the process or else continue further
- Repeat all steps in case of nested macros.
- After saving all definitions the control transfer to Pass-II for further processing of calls
- ALA gets made simultaneously as index markers are ~~used~~ stored for dummy arguments while storing macro definitions in MDT.

similar to
each entry
macro
in the
of

s a
step or
with
og with
MDT
ro is
op the
macros.
) transfers
calls
r markers
while

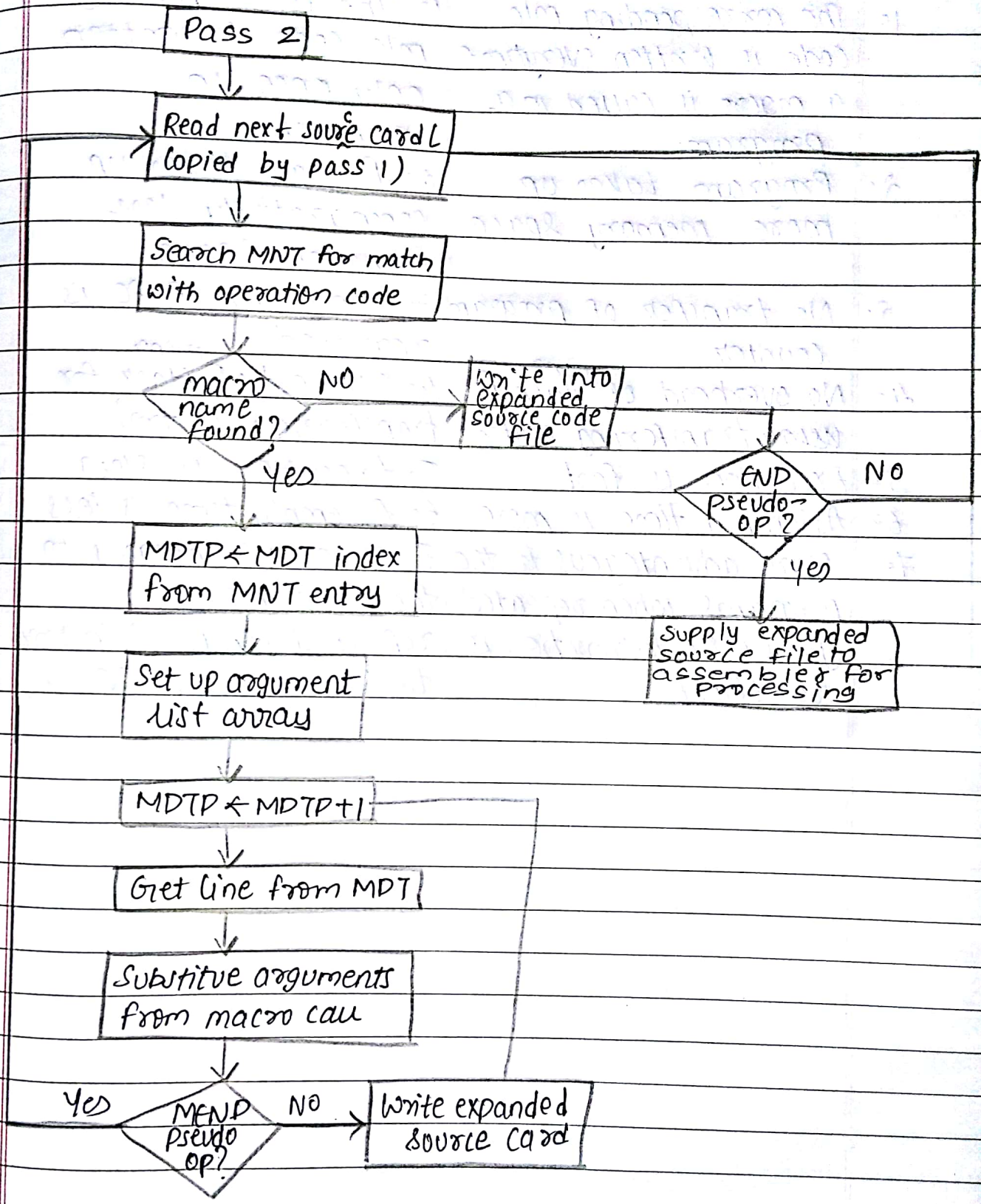
YOU CAN EXPLAIN MORE BY EXPLANING FLOWCHART OF PASS I & II



Pass 2

- Check every input line of the text to see if its name is in the MNT if yes set the pointer MDTP to the corresponding macro definition in MDT.
  Intialize MDTP to 'MDT index' field of MNT.
- ALA is prepared for Macro expander using taking reference from the ALA of Pass-I to substitute actual parameters when the calls are made.
- The step 1,2 contine until MEND line is reached and MDT terminates expansion of MACRO and scanning continues from i/p file
- When END is encountered the final & expanded code is transferred to the assembler for further processessing

FLOWCHART



	MACROS	PROCEDURE / SUBROUTINE
1.	The corresponding m/c code is written everytime a macro is called in a program.	1. The corresponding m/c code is written only once in memory
2.	Program takes up more memory space	2. Program takes up comparatively less memory space
3.	No transfer of program counter	3. Transfering of PC is required
4.	No overhead of using stack for transfering control	4. Overhead of using stack for transferring control
5.	Execution is fast	5. Execution is slow
6.	Assembly time is more	6. Assembly time is less
7.	More advantegous to the programs when repeated group of instruction is too short	7. More advantegous to the programs when repeated group of instruction is quite large.