Unit 3 Macros and Macro processor

Que: Define macro. Explain macro definition and macro call with an example?

Ans: Macro definitions typically appear at the start of a program. Each macro definition is enclosed between a macro header statement and a macro end statement, which have the mnemonic opcodes MACRO and MEND, respectively.

Statements included in the macro definition can use formal parameters: the character & is prefixed to the name of formal a parameter to differentiate a formal parameter's name from other symbolic names

A macro definition can have three kinds of statements in it.

1. A macro prototype statement declares the name of the macro and the names and kinds of its formal parameters.

2. A model statement is a statement from which an assembly language statement may be generated during macro expansion.

3. A macro preprocessor statement is used to perform auxiliary functions during macro expansion. Preprocessor statements are described in later sections

- The macro prototype statement has the following syntax:

  <macro name> [<formal parameter specification>[...]]

where <macro name> appears in the mnemonic field of an assembly statement and the formal parameter specification appears in the operand field of the statement. A <formal parameter specification> is of the form

&<parameter name> [<parameter kind >

A parameter can be either a positional parameter or a keyword parameter. A parameter is assumed to be a positional parameter by default, ie, if the specification <parameter kind> is omitted. Different kinds of parameters are handled differently during macro expansion

- A macro call has the syntax

  macro name> <actual parameter specification>[...]]

where <macro name> appears in the mnemonic opcode field of an assembly state- ment. Actual parameters appear in the operand field of the statement. <actual parameter specification> resembles <operand specification> in an assembly language statement

Example   (Macro definition and call)

the definition of the macro INCR that was discussed earlier in MACRO and MEND are the macro header and macro end statements, respectively. The prototype statement indicates that INCR has three parameters called MEM_VAL, INCR_VAL and REG.

Since parameter kind is not specified for any of the parameters, each parameter is assumed to be a positional parameter.

Statements with the operation codes MOVER, ADD and MOVEM are model statements No preprocessor statements are used in this macro.

A statement INCR A, B, AREG appearing in the program would be considered to be a call on macro INCR

```
MACRO
INCR      &MEM_VAL,  &INCR_VAL,  &REG
MOVER      &REG,  MEM_ VAL
ADD        &REG,  INCR_ VAL
MOVEM      &REG, &MEM_VAL
MEND
```

4.1 A Macro Definition

Que: What is Macro expansion, Discuss two diferent ways of macro expansion ?

Ans: MACRO EXPANSION

Macro expansion can be performed by using two kinds of language processors. A macro assembler performs expansion of each macro call in a program into a sequence of assembly statements and also assembles the resulting assembly program.

A macro preprocessor merely performs expansion of macro calls in a program. It produces an assembly program in which a macro call has been replaced by statements that resulted from its expansion but statements that were not macro calls have been retained in their original form. This program can be assembled by using an assembler.

The macro preprocessor operates as follows:

If the first statement in the program input to it is a macro header statement, it knows that one or more macro definitions exist in the program.

It processes and stores all macro definitions in its own data structures. After all macro definitions have been processed, it produces the output program as follows:

If the next statement in the program is not a macro call, it merely copies the statement into the output program. If it is a macro call, it visits die model statements found in the definition of the called macro and generates assembly statements from them through the substitution of formal parameters.

We assume that the macro preprocessor would help the programmer in distinguishing between original statements of a program and the statements generated during macro expansion by printing the symbol '+' before the label field of each generated statement .

We discuss two key notions used in implementing macro expansion before we present a scheme for it

1. Flow of control during expansion: Its rules determine the order in which model statements

in a macro's definition would be visited for expansion of a macro call.

2. Lexical substitution: Lexical substitution is used to generate an assembly statement from a model statement

- **Flow of control during expansion**

The default flow of control during macro expansion is sequential. Thus, unless the flow of control is altered by using a preprocessor statement the model statements in a macro definition are visited sequentially starting with the statement following the macro prototype statement and ending with the statement preceding the MEND statement.

A preprocessor statement can alter the flow of control during expansion such that some model statements are either not visited during expansion or are visited repeatedly during expansion.

The former results in conditional expansion, where in some model statements are not expanded for some of the macro calls, and the latter results in expansion time loops, where in some model statements may be expanded more than once.

Both these effects are important for semantic expansion Algorithm 4.1 gives an outline of macro expansion. The flow of control dur- ing macro expansion is implemented by using the macro expansion counter (MEC)

It points to the model statement that is to be expanded next; it is called a counter because of the default action of sequential control flow during expansion.

Algorithm 4.1 (Outline of macro expansion)

1. MEC := statement number of the first model statement following the prototype statement in the definition of the called macro:

2. While the statement pointed to by MEC is not a MEND statement

(a) If a model statement then

(i)Expand the statement through lexical Substitution

(ii) MEC:= MEC + 1;

(b) Else (i.e, the statement is a preprocess statement)

MEC := value specified in the preprocessor statement

3. Exit from macm.expansion

MEC is set to point at the statement following the prototype statement.

It is incremented by I after expanding a model statement .

Execution of a preprocessor statement can set MEC to a new value to implement conditional expansion or expansion time loops.

- **Lexical substitution**

A model statement may use strings of the following three type

1.An ordinary string, which is any string other than a string of type 2 or type 3 defined below

2.The name of a formal parameters which is preceded by the character ‘&’.

3. The name of a preprocessor variable, which is also preceded by the character ‘&’.

During lexical substitution, an ordinary string in a model statement is retained in its original form .

(Effectively, an ordinary string stands for itself) A name of a formal parameter or preprocesser variable appearing in the model statement is replaced by its value.

The value of a formal parameter is the corresponding actual parameter string used in a macro call where the rules of correspondence depend on what kind a parameter is.

We discuss two kinds of formal parameters, called positional and keyword parameters, below The value of a preprocessor variable is readily known to the preprocessor from its own data structures.

Que. Explain with example positional parameter, Keyword Parmeter and default specification of parameter

Ans :

- **Positional parameters**

For positional formal parameters, the specification <parameter kind> of syntax rule Macro definition and call is simply omitted. Thus, a positional formal parameter is written as &<parameter name>. e.g., &SAMPLE where SAMPLE is the name of a parameter.

In a call on a macro using positional parameters see syntax rule (Macro definition and call), the <actual parameter specification> is an ordinary string. The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:

1. Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement

2. Find the actual parameter specification that occupies the same ordinal position in the list of actual parameters in the macro call statement. If it is an ordinary string ABC, the value of formal parameter XYZ would be ABC

Example-1 (Parameter association for positional parameters in a macro call)

Consider the call

```
INCR                    A, B , AREG
```

on macro INCR of According to the rule of positional association, values of the formal parameters are as follows:

formal parameter	value
MEM_VAL	A
INCR_ VAL	B
REG	AREG

Lexical substitution in the model statements now leads to the code

```
+          MOVER      AREG ,A
+          ADD        AREG, B
+          MOVEM      AREG, A
```

- **Keyword parameters**

For keyword parameters, the specification <parameter kind> is the string in the syntax rule (Macro definition and call). The <actual parameter specification> is written as <formal parameter name> =<ordinary string>.

The value of a formal parameter XYZ is determined by the rule of keyword association as follows:

1. Find the actual parameter specification which has the form XYZ= <ordinary string>

2. If the <ordinary string> in the specification is some string ABC, the value of formal parameter XYZ would be ABC.

Note that the ordinal position of the specification XYZ=ABC in the list of actual parameters is immaterial. This feature is useful in writing a call on a macro that has a large number of parameters.

```
MACRO
INCR_M     &MEM_VAL= , &INCR_VAL= ,& REG =
MOVER      & REG, &MEM_VAL
ADD         & REG, &INCR_VAL
MOVEM       & REG, &MEM_VAL
MEND
```

4.2 A macro definition using keyword parameters

Example-2 (Parameter association for keyword parameters in a macro call)

4.2 A macro definition using keyword parameters shows macro INCR of Figure 4.1 **Macro definition** rewritten as macro INCR_M that uses keyword parameters. The following calls on this macro are equivalent

```
INCR_M          MEM_VAL=A, INCR_VAL=B, REG_AREG
……….
INCR_ M         INCR_VAL=B , REG_AREG, MEM_VAL=A
```

- **Specifying Default values of parameters**

If a parameter has the same value in most calls on a macro, this value can be specified as its default value in the macro definition itself.

If a macro call does not explicitly specify the value of the parameter, the preprocessor uses its default value, otherwise, it uses the value specified in the macro call.

This way, a programmer would have to specify a value of the parameter only when it differs from its default value specified in the macro definition.

Default values of keyword parameters can be specified by extending the syntax of formal parameter specification as follows:

&<parameter name>[<parameter kind >[< default value>]]

Example-3 (Specifying default values of keyword parameters)

If a program uses register AREG for performing most of its arithmetic, most calls on macro INCR_M of Figure 4.2 would contain the specification &REG=AREG. BELOW Fig macro INCR_D which is analogous to macro INCR_M except that it specifies a default value for parameter REG. Among the following calls

```
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
```

the first two calls are equivalent to the calls in Example 2 (keyword parameter) because the default value REG would be used during their expansion.

The value BREG that is explicitly specified for REG in the third call overrides it default value, so BREG will be used to perform the arithmetic in its expanded code.

```
MACRO
INCR_D        &MEM _VAL=, & INCR_VAL=,  &REG=AREG
MOVER        &REG, &MEM _VAL
ADD            &REG, &INCR_VAL
MOVEM        &REG, &MEM _VAL
MEND
```

4.3 A macro definition specifying a default value for a keyword parameter

- ## Macros with mixed parameter lists

A macro definition may use both positional and keyword parameters. In such a case, all positional parameters must precede all keyword parameters in a macro call. For example, in the macro call

```
SUMUP   A, B, G =20, H=X
```

A and B are positional parameters while G and H are keyword parameters. Correspondence between actual and formal parameters is established by applying the rules governing positional and keyword parameters separately

Other uses of parameters

The model statements of Examples 1-3 have used formal parameters only in operand fields. However, use of parameters is not restricted to these fields. Formal parameters can also be used in the label and opcode fields of model statements.

Example 4 (Use of parameters in label and opcode fields)

Consider the macro definition

```
          MACRO
          CALC      &X, &Y, &OP=MULT, &LAB=
&LAB      MOVER     AREG, &X
          &OP       AREG, &Y
          MOVEM     AREG,&X
          MEND
```

Here parameter &LAB has been used in the label field of a statement and &OP in the op-code field. Expansion of the call CALC A, B, LAB=LOOP would lead to the following code:

```
+ LOOP    MOVER     AREG, A
+         MULT      AREG , B
+         MOVEM     AREG, A
```

Que 4.- Explain Nested Macro calls with example

Ans: NESTED MACRO CALLS

A model statement in a macro may constitute a call on another macro. Such a call is known as a nested macro call.

We refer to the macro that contains the nested call as the outer macro and the macro that is called in the nested call as the inner macro.

The macro preprocessor performs expansion of nested macro calls using the last –in-first-out (LIFO) rule.

Thus, at any moment it is engaged in expanding the innermost macro call whose expansion is not yet complete.

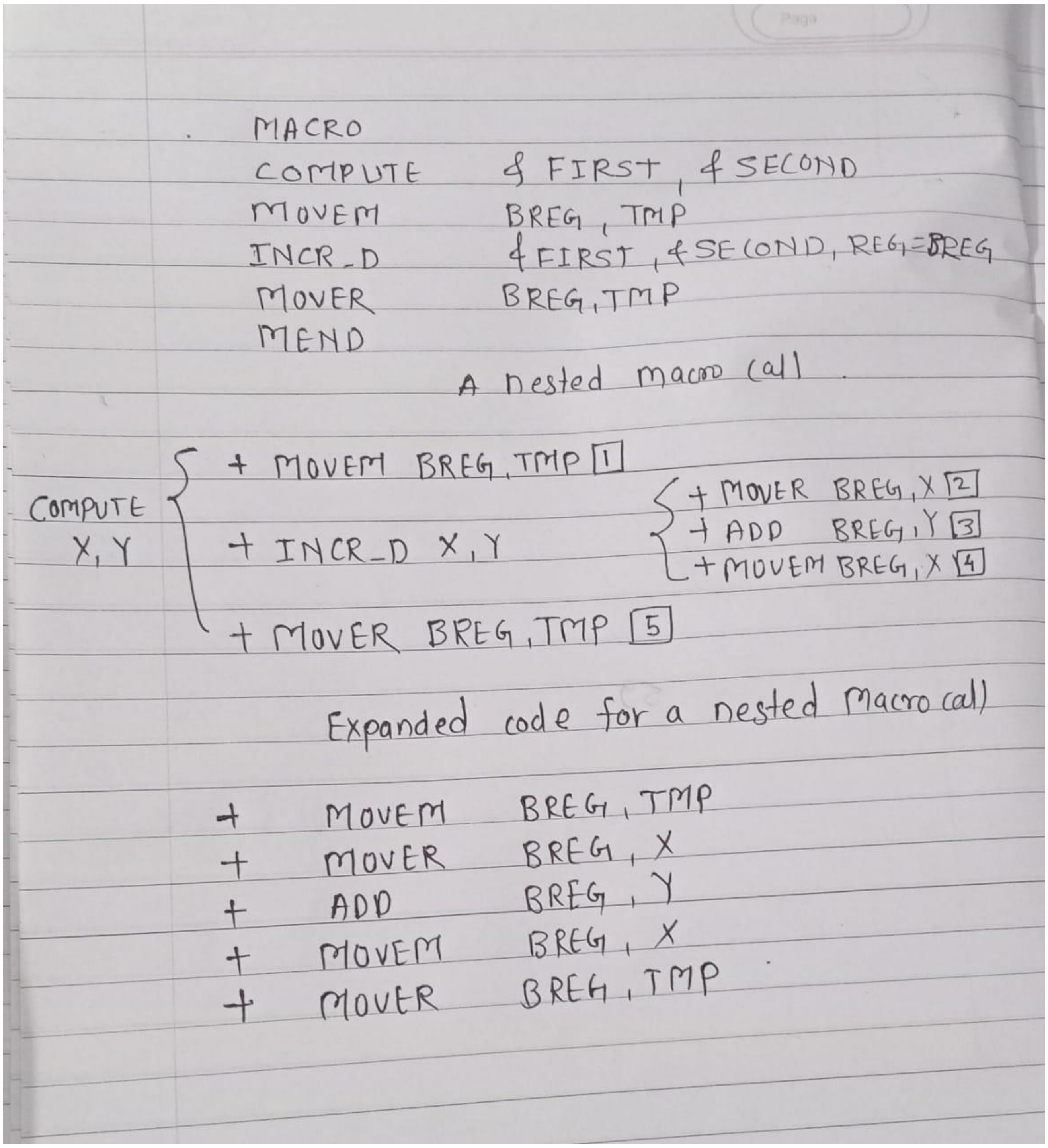
Example 4.7 illustrates operation of the LIFO rule of expanding nested macro calls.

Example 4.7 (Nested macro calls) Macro COMPUTE of eg 2 contains a nested call on macro INCR_D of above eg 1 ,2 and 3 shows the expanded code for the call

```
     COMPUTE    X,Y
```

After performing lexical substitution in the second model statement of COMPUTE the preprocessor recognizes it to be a call on macro INCR _D, so it starts expanding that call. It leads to the generation of statements marked 2,3 AND 4) in Below fig After completing expansion of the call on INCR_ D, the preprocessor resumes expansion of COMPUTE,

so the third model statement of COMPUTE is now expanded. Thus the expanded code for the call on COMPUTE is as follows



```
MACRO
COMPUTE     &FIRST, &SECOND
MOVEM       BREG, TMP
INCR_D      &FIRST, &SECOND, REG=BREG
MOVER       BREG,TMP
MEND
```

A nested macro call.

```
          { + MOVEM BREG,TMP [1]
COMPUTE   {                          { + MOVER  BREG,X [2]
  X,Y     { + INCR_D X,Y             { + ADD    BREG,Y [3]
          {                          { + MOVEM  BREG,X [4]
          { + MOVER BREG,TMP [5]
```

Expanded code for a nested Macro call)

```
+   MOVEM   BREG, TMP
+   MOVER   BREG, X
+   ADD     BREG, Y
+   MOVEM   BREG, X
+   MOVER   BREG, TMP
```

Que- Which are the advanced macro facilities for alteration of flow of control during expansion ,Attributes of parameter and Expansion time Variables.

Ans: ADVANCED MACRO FACILITIES

Advanced macro facilities are aimed at supporting semantic expansion. These facilities can be classified as follows:

1.Facilities for altering flow of control during expansion

2. Attributes of parameters.

3. Expansion time variables

This section describes some advanced macro facilities and illustrates their use in performing conditional expansion of model statements and in writing expansion time loops.

- **Altering flow of control during expansion**

Flow of control during macro expansion can be altered by performing an expansion-time control transfer, which directs the preprocessor to a particular model statement for expansion. It is achieved as follows:

A unique sequencing symbol is written in the label field of a statement in a macro definition.

It provides a convenient method of identifying that statement for expansion-time control transfer.

A preprocessor statement with the mnemonic AIF or AGO uses this sequencing symbol as an operand .

If the condition specified in the AIF statement is satisfied, the macro preprocessor puts the statement number of the statement that has the sequencing symbol in its label field into the MEC used The action is performed unconditionally in the case of the AGO statement.

A sequencing symbol has the syntax, <ordinary string>. Its appearance in the label field of a statement constitutes its definition.

Its purpose is merely to identify a statement for expansion-time control transfers, so it does not appear in the expanded code when that statement is expanded.

An AIF statement has the syntax

AIF (<expression>) <sequencing symbol>

where <expression> is a relational expression involving ordinary strings formal parameters and their attributes, and expansion time variables.

If the relational expression evaluates to true, expansion time control is transferred to the statement that contains <sequencing symbol> in its label field; otherwise, the expansion time control flow is not altered.

An AGO statement has the syntax

AGO <sequencing symbol>

and unconditionally transfers expansion time control to the statement that contains <sequencing symbol > in its label field. An ANOP statement is written as

<sequencing symbol> ANOP

and simply has the effect of defining the sequencing symbol. It is inserted prior to a statement in the macro definition to which expansion-time control flow is to be transferred if that statement has another symbol in its label field.

The ANOP statement does not appear in the expanded code of a macro call.

- **Attributes of formal parameters**

An attribute is written by using the syntax

<attribute name> <formal parameter spec>

and represents information about the value of the formal parameter) i.e., about the corresponding actual parameter. The type, length and size attributes have the names T,L and S. Attributes can be used in preprocessor statements.

Example(Attributes of formal parameters)

```
           MACRO
           DCL_CONST      &A
           AIF             (L'&A EQ 1).NEXT
          _ _
.NEXT     _ _
          _ _
         MEND
```

Here expansion time control is transferred to the statement having .NEXT in its label field only if the actual parameter corresponding to the formal parameter A has the length of 1.

- **Expansion time variables**

An expansion time variable (EV) is a variable that is meant for use only during expansion of macro calls.

Accordingly, its value can be used only within a macro definition in a preprocessor statement that assigns a value to an expression variable, in a model statement, and in the expression of an AIF statement.

A macro definition must contain the declaration of every expansion time variable that it uses.

Two kinds of expansion time variables exist. A local expansion time variable can be used only within one macro definition and it does not retain its value across calls on that macro.

A global expansion time variable can be used in every macro definition that has a declaration for it and it retains its value across macro calls, Local and global expression variables are created through declaration statements with the following syntax:

LCL <EV specification>[,<EV specification>..]

GBL <EV specification>[,<EV specification> ..]

where <EV specification> has the syntax & <EV name>. where <EV name> is an ordinary string.

Values of expansion time variables can be manipulated through the preprocessor statement SET. It has the following syntax:

<EV specification> SET <SET-expression>

where <EV specification> appears in the label field and SET in the opcode field. A SET statement assigns the value of <SET-expression> to the expansion time variable specified in <EV specification>.

illustrates use of expansion time variables.

(Expansion time variables)

```
MACRO
CONSTANTS
LCL             &A
 &A             SET         1
                DB           &A
&A              SET          &A+1
                DB
                MEND
```

A call on macro CONSTANTS is expanded as follows: The local expansion time variable A is created. The first SET statement assigns the value '1' to it. The first DB statement thus declares a byte constant “1”. The second SET statement assigns the value 2 to A, so the second DB statement declares a constant "2" example.

Advanced macro facilities.

(I) AIF
→ AIF (exp) <seq. symbol>

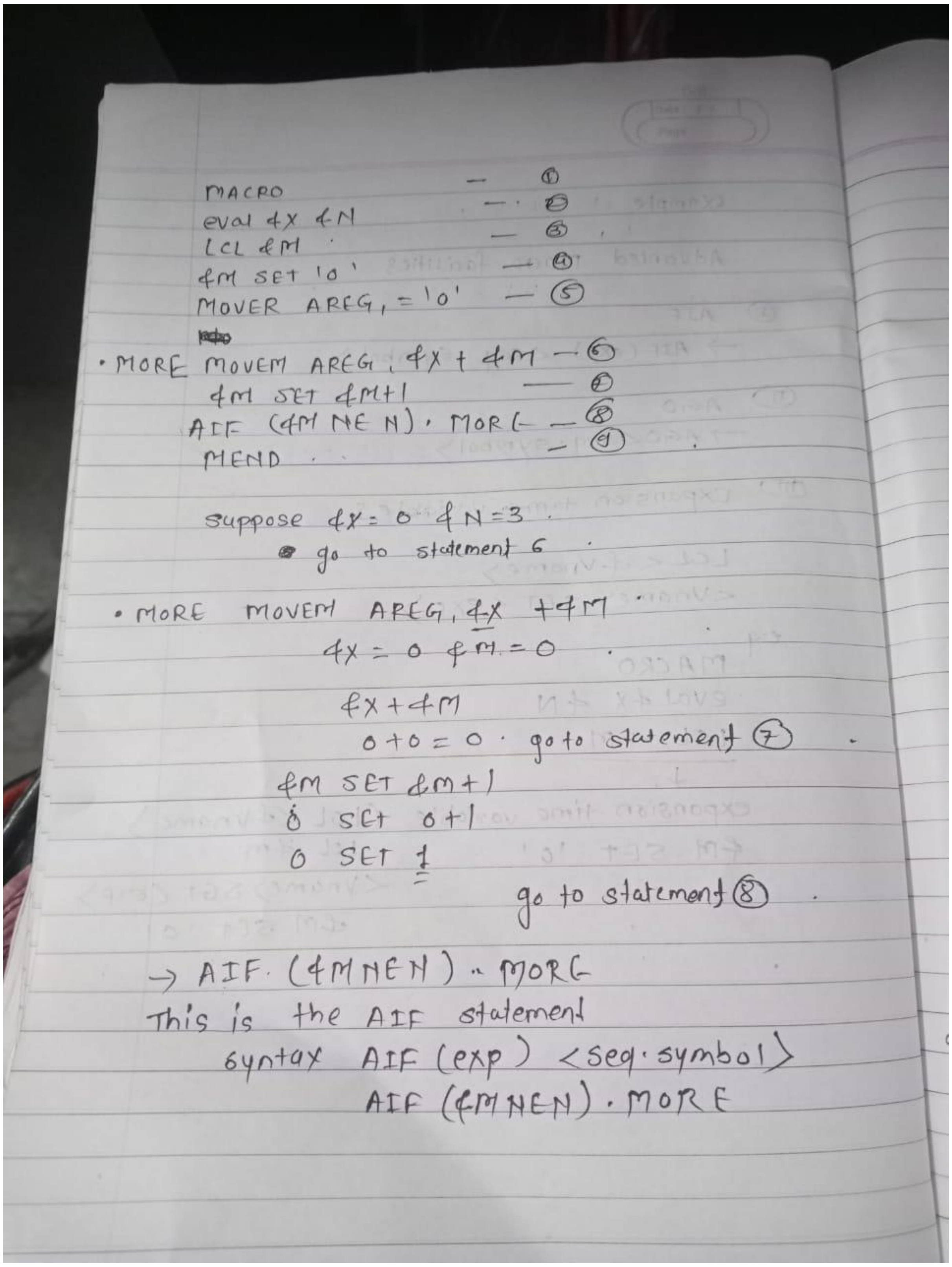
(II) AGO
→ AGO<seq. symbol>

(III) Expansion time variables.

LCL<&vname>
<Vname> SET <exp>

e.g.
```
MACRO
eval &X &N
LCL &M.
```
↓.
expansion time variable (LCL <&vname>
&M SET '0'
LCL &m .
<Vname> SET <exp>
&M SET '0'

.



```
      MACRO                        — (1)
      eval &X &N                   — (2)
      LCL &M                       — (3)
      &M SET '0'                   — (4)
      MOVER AREG, ='0'             — (5)
• MORE MOVEM AREG, &X + &M         — (6)
      &M SET &M+1                  — (7)
      AIF (&M NE N) . MORE         — (8)
      MEND                         — (9)
```

suppose &X = 0 &N = 3 .

go to statement 6 .

• MORE MOVEM AREG, &X + &M

&X = 0 &M = 0 .

&X + &M

0 + 0 = 0 . go to statement (7) .

&M SET &M + 1

0 SET 0 + 1

0 SET 1

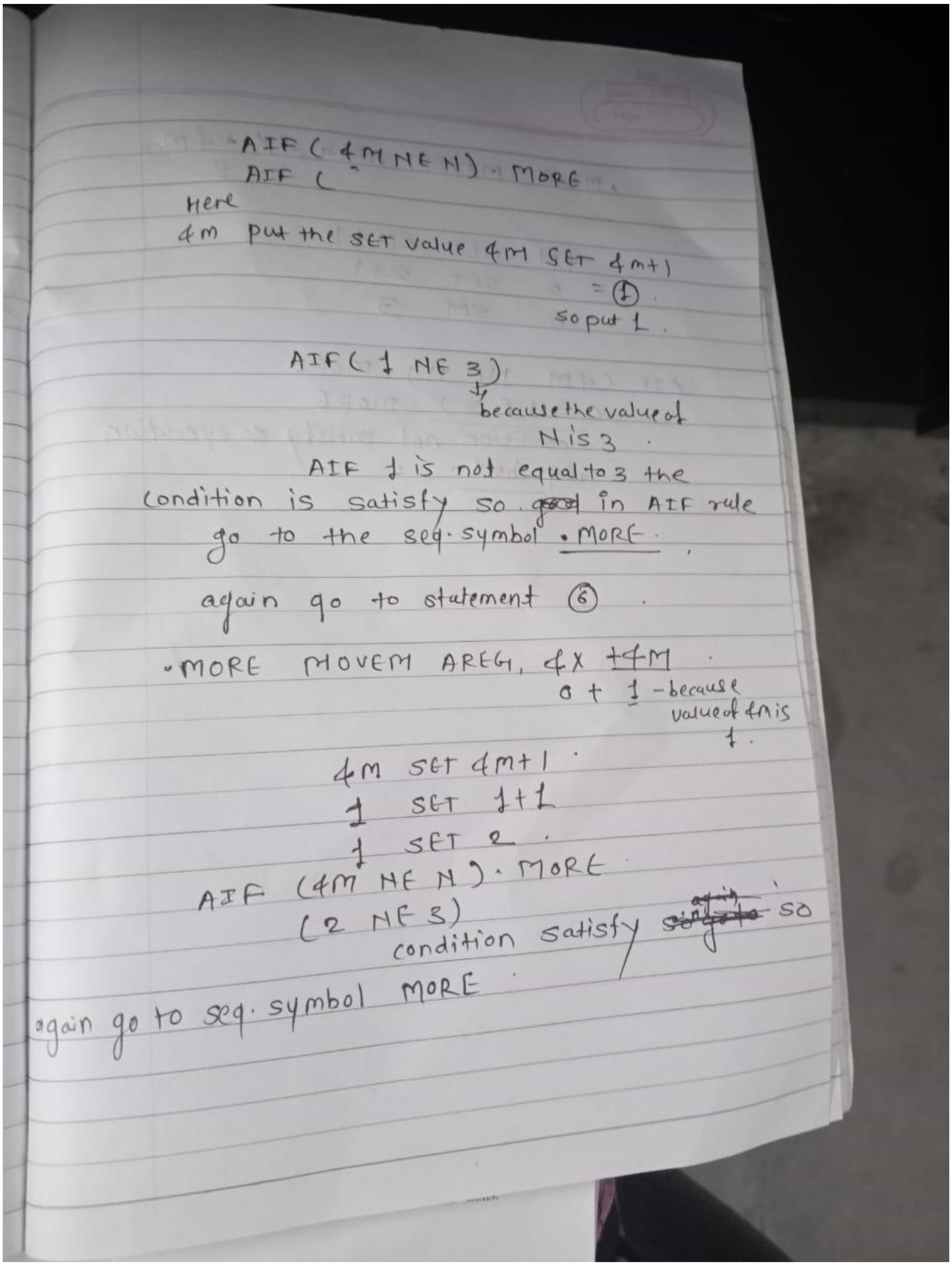
go to statement (8) .

→ AIF (&M NE N) . MORE

This is the AIF statement

syntax AIF (exp) <seq.symbol>

AIF (&M NE N) . MORE



AIF (&M NE N) . MORE .
AIF (

Here
&M put the SET value &M SET &M+1
= $\textcircled{1}$.
so put 1 .

AIF(1 NE 3)
because the value of N is 3 .

AIF 1 is not equal to 3 the condition is satisfy so . ~~goo~~ in AIF rule go to the seq. symbol **.MORE** .

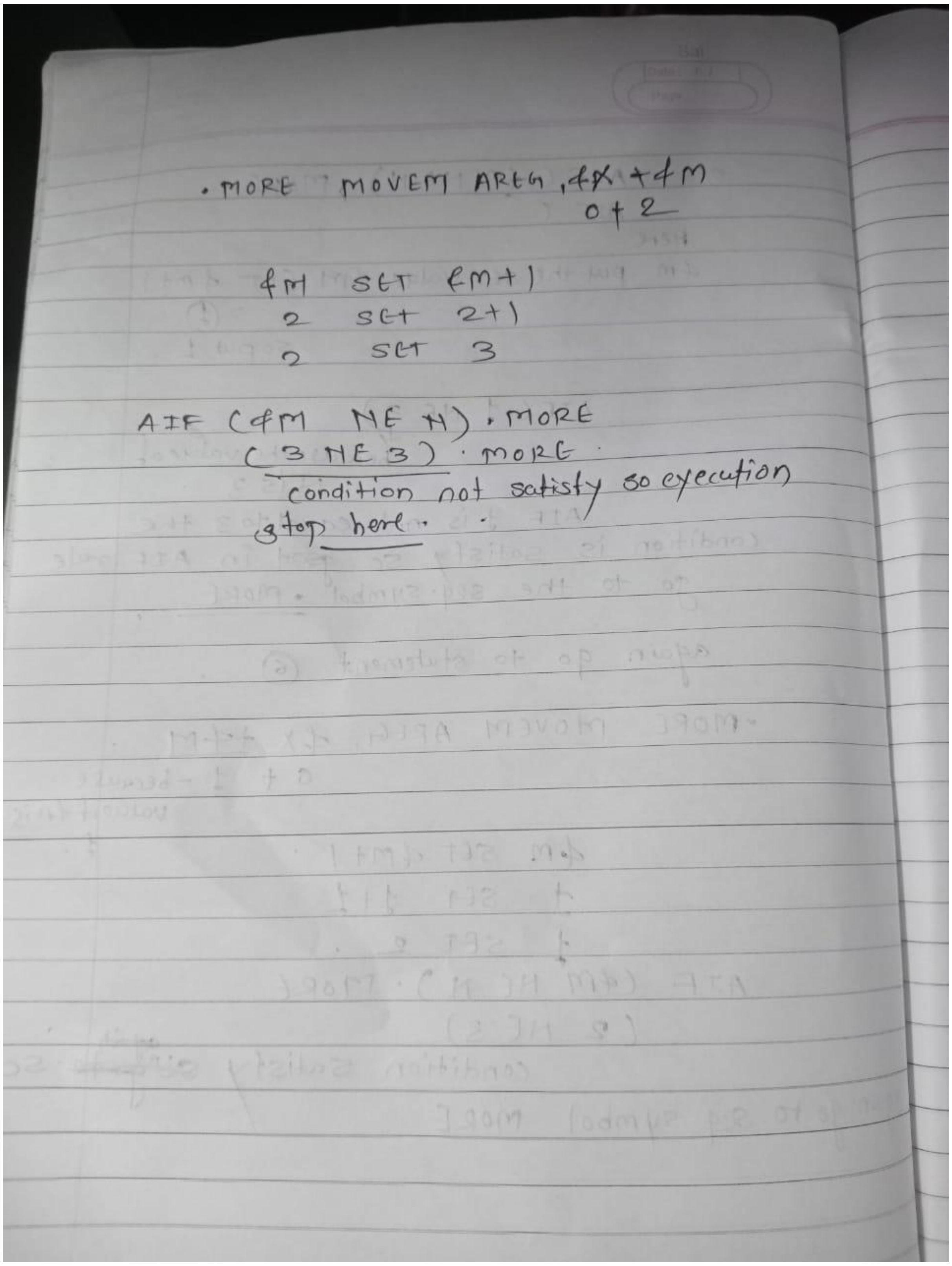
again go to statement $\textcircled{6}$.

.MORE MOVEM AREG, &X +&M .
0 + 1 - because value of &m is 1 .

&m SET &m+1 .
1 SET 1+1
1 SET 2 .

AIF (&M NE N) . MORE
(2 NE 3)
condition satisfy ~~so go to~~ again so

again go to seq. symbol MORE .



.MORE MOVEM AREA, &X + &M
0 + 2

&M SET &M+1
2 SET 2+1
2 SET 3

AIF (&M NE N).MORE
(3 NE 3).MORE
condition not satisfy so execution
stop here.

Que. Explain Semantic expansion with example?

Ans:

Semantic Expansion

Semantic expansion is the generation of statements tailored to the requirements of a specific usage.

Its use makes a macro adaptive Semantic expansion can be achieved by a combination of advanced macro facilities like the AIF, AGO statements and expansion time variables.

The CLEAR macro of an instance of semantic expansion. Here, the number of MOVEM AREG, .. statements generated by a call on CLEAR is determined by the value of the second parameter of CLEAR.

Macro EVAL of is another instance of conditional expansion wherein one of two alternative code sequences is generated depending on peculiarities of actual parameters of a macro call. **illustrates semantic expansion by using the type attribute.**

Example (Semantic expansion by using the type attribute) Macro CREATE _CONST creates a constant whose value is 25, whose name is given by the second parameter in a call and whose type matches the type of the first parameter.

```
        MACRO
        CREATE _CONST       &X ,&Y
        AIF                (T’ &X EQ B) .BYTE
&Y      DW                  25
        AGO                 .OVER
BYTE    ANOP
&Y      DB                  25
.OVER   MEND
```

Que. Explain data structure of Macro- Preprocessor

Ans:

Data Structures of the Macro Preprocessor

During expansion of a macro, the actual parameter table (APT), the parameter default table (PDT), and the expansion time variables' table (EVT) contain information concerning parameters and expansion time variables.

While processing a model statement in the body of code in the macro definition, the preprocessor would search these tables by using the name of a parameter or expansion time variable as a key.

and make a substitution using the value found in the table. This search could be eliminated by converting the statements in the body of code into a suitable intermediate code while storing them in the MDT. For example, the model statement

```
MOVER          AREG,     &ABC
```

where ABC is the fifth parameter of the macro, could be stored as

```
MOVER       AREG, (P, 5)
```

in the MDT, where (P,5) stands for the words 'parameter #5'.

Thus, storing the intermediate code of statements in the MDT would eliminate searches in the tables.

An interesting offshoot of this decision is that the first component of the pairs stored in the APT, which was the name of a parameter, is no longer used during macro expansion. Hence instead of using the APT.

which contained pairs of the form (<formal parameter name>, <value>), we can use another table called APTAB which contains only values of parameters. As discussed earlier in the context of parameter named ABC, information in the APTAB would be accessed by using the parameter number found in the intermediate code of a statement

For converting the statement MOVER AREG, &ABC into the intermediate code

MOVER AREG, (P, 5), ordinal numbers have to be assigned to all parameters of a macro. A table named parameter name table (PNTAB) could be used for this pur pose. Parameter names would be entered in PNTAB in the same order in which they appear in the prototype statement of the macro. The entry # of a parameter's entry in PNTAB would now be its ordinal number. It would be used in the intermediate code of a statement

In effect, the information (<formal parameter name>, <value>) in the APT has been split into two tables:

- PNTAB contains formal parameter names.
- APTAB contains values of formal parameters, most of which are actual parameters and others are defaults.

Note that the PNTAB is used while processing a macro definition while the APTAB is used during macro expansion.

Similar analysis leads to splitting of the execution time variables' table (EVT) into EVNTAB and EVTAB and splitting of the sequencing symbol table (SST) into SSNTAB and SSTAB.

The name of an expansion time variable is entered in the EVNTAB while processing its declaration. The name of a sequencing symbol is entered in the SSNTAB while processing the definition of the sequencing symbol or a reference to it, whichever occurs earlier. This aspect resembles construction of the symbol table in a single-pass assembler

The parameter default table (PDT) can be split analogously; however, some more simplifications are possible.

The positional parameters (if any) of a macro appear before keyword parameters in the prototype statement.

Hence if a macro BETA has p positional parameters and k keyword parameters, the keyword parameters have the ordinal numbers p+1… p+k.

Due to this numbering, the following two kinds of redundancies appear in the PDT: The first component of each entry is redundant as in the APTAB and the EVTAB.

Further, entries 1... p are redundant since positional parameters cannot have default specifications.

Hence entries only need to exist for parameters numbered p+1 …p+k. To accommodate these changes, we replace the parameter default table (PDT) by a keyword parameter default table (KPDTAB). KPDTAB of macro BETA would have only k entries in it. To note that the first entry

NAME OF TABLE	FIELDS IN EACH ENTRY
Macro name table (MNT)	Macro name, Number of positional parameters (#PP) Number of keyword parameters (#KP) Number of expansion time variables (#EV). MDT pointer (MDTP). KPDTAB pointer (KPDTP). SSTAB pointer (SSTP)
Parameter Name Table (PNTAB)	Parameter name
Expansion time Variables' Name Table (EVNTAB)	Expansion time variable name
Sequencing Symbol Name Table (SSNTAB)	Sequencing symbol name
Keyword Parameter Default Table (KPDTAB)	Parameter name, Default value
Macro Definition Table (MDT)	Lable, Opcode Operands
Actual Parameter Table (APTAB)	Value
Expansion time Variables Table (EVTAB)	Value
Sequencing Symbol Table (SSTAB)	MDT entry #

Tables of the macro preprocessor

of KPDTAB corresponds to the parameter numbered p+1, we store p, the number of positional parameters of macro BETA, in a new field of its MNT entry.

Each MNT entry would now contain three pointers called the MDTP, the KPDTP and the SSTP. which are pointers to the MDT.

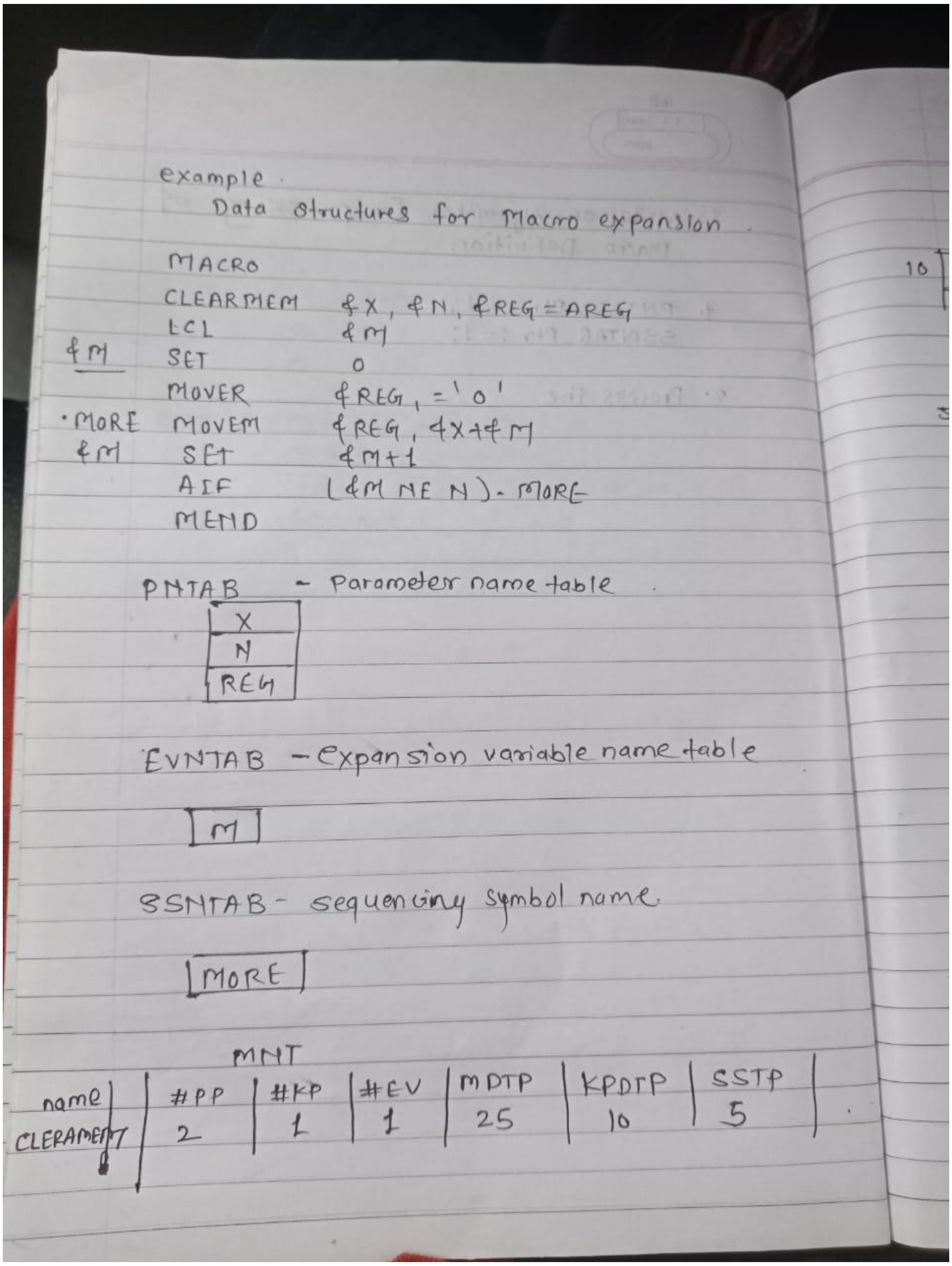
The KPDTAB and the SSNTAB for the macro, respectively. Instead of using different MDT's for different macros, for simplicity we could create a single MDT and use different sections in it for storing statements from the code bodies of different macros. A similar arrangement can be used with the KPDTAB and the SSNTAB.

Construction and use of the macro preprocessor data structures can be summarized as follows: PNTAB and KPDTAB are constructed by processing the prototype statement. Entries are added to the EVNTAB and SSNTAB as declarations of expansion time variables and definitions/references of sequencing symbols are processed.

MDT entries are constructed while processing the model statements and preprocessor statements in the macro body. An entry is added to SSTAB when the definition of a sequencing symbol is encountered.

APTAB is constructed while processing a macro call.

EVTAB is constructed at the start of macro expansion



example.

Data structures for Macro expansion.

```
      MACRO
      CLEARMEM   &X, &N, &REG=AREG
      LCL        &M
&M    SET        0
      MOVER      &REG, ='0'
.MORE MOVEM      &REG, &X+&M
&M    SET        &M+1
      AIF        (&M NE N).MORE
      MEND
```

PNTAB - Parameter name table.

X
N
REG

EVNTAB - Expansion variable name table

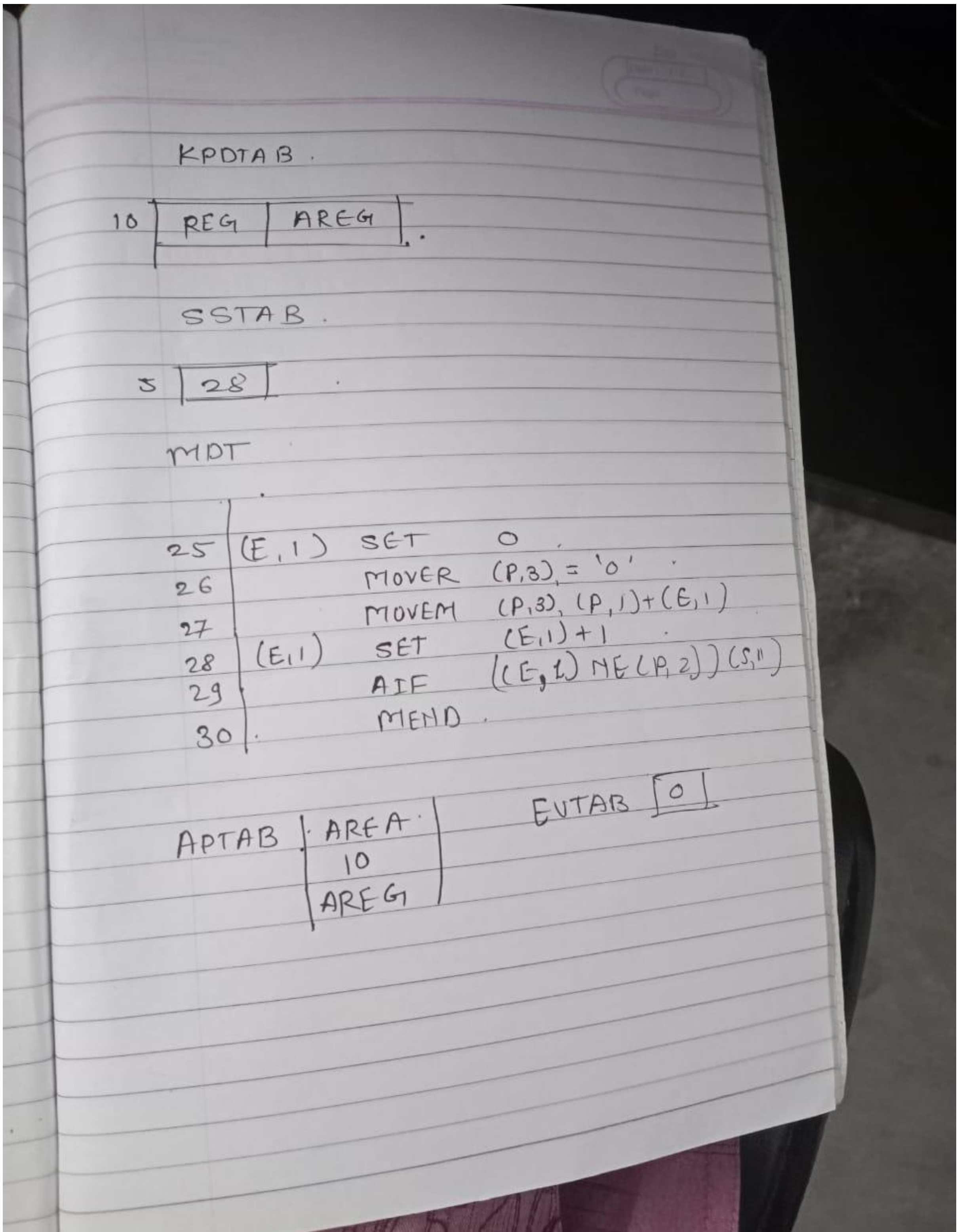
M

SSNTAB - sequencing symbol name

MORE

MNT

name	#PP	#KP	#EV	MDTP	KPDTP	SSTP
CLERAMENT	2	1	1	25	10	5



KPDTAB

10	REG	AREG

SSTAB

5	28

MDT

```
25 | (E,1)  SET     0
26 |        MOVER   (P,3), = '0'
27 |        MOVEM   (P,3), (P,1)+(E,1)
28 | (E,1)  SET     (E,1) + 1
29 |        AIF     ((E,1) NE (P,2)) (S,1)
30 |        MEND
```

APTAB

AREA
10
AREG

EVTAB $\boxed{0}$

Que. Write an Algorithm for Procsing of Macro Definition

Ans: (Processing of a macro definition)

```
1. PNTAB_ptr := 1;
   SSNTAB _ptr:= 1;
2. Process the macro prototype statement and form the MNT entry for the macro
(a) name:= macro name; #PP: =0; #KP: =0;
(b) For each positional parameter
   (i) Enter parameter name in PNTAB [PNTAB_ptr).
   (ii) PNTAB _ptr:= PNTAB_ptr + 1;
    (iii) #PP:= #PP+1;

(c) KPDTP:= KPDTAB_ptr;
(d) For each keyword parameter
(i) Enter parameter name and default value (if any), in the entry KPDTAB [KPDTAB_ ptr].
(ii) Enter parameter name in PNTAB [PNTAB_ptr].
(iii) KPDTAB_ptr:= KPDTAB_ ptr + 1;
(iv) PNTAB _ptr := PNTAB_ ptr + 1;
(v) #KP:= #KP + 1;
(e) MDTP:= MDT_ ptr;
(f) #EV:=0;
(g) SSTP:= SSTAB_ptr;
3. While not a MEND statement

(a) If an LCL statement then
For each expansion time variable declared in the statement: Enter name of the expansion time variable in EVNTAB
#EV:= #EV+1;

(b) If a model statement then
```

(i) If the label field contains a sequencing symbol then If the symbol is present in SSNTAB then q entry number of the symbol in SSNTAB else Enter the sequencing symbol in SSNTAB[SSNTAB_ptr].

q:=SSNTAB_ptr;

SSNTAB_ptr:= SSNTAB -ptr +1;

SSTAB _ptr:= SSTAB_ ptr +1;

SSTAB[SSTP+q-1]:= MDT_ ptr;

(ii) Construct an intermediate code for the statement as follows:

For every occurrence of a parameter in the statement: Replace the parameter by the specification (P, #n), where n is the entry number occupied by the parameter in the PNTAB .

For every occurrence of an expansion time variable in the statement: Replace the expansion time variable by the specification (E, #n), where n is the entry number occupied by the expansion time variable in the EVNTAB.

For every occurrence of a sequencing symbol in the statement: Replace the sequencing symbol by the specification (S. #n), where n is the entry number occupied by the sequencing symbol in the SSNTAB.

(iii) Record the intermediate code of the statement in MDT [MDT _ptr];

(iv) MDT _ptr:= MDT _ptr + 1;

(c) If a SET,AIF or AGO statement then

(i) If a SET statement then

For every occurrence of an expansion time variable in the statement: Replace the expansion time variable by the specification (E, #n), where n is the entry number occupied by the expansion time variable in the EVNTAB.

(ii) If an AIF or AGO statement then

If the sequencing symbol used in the statement is present in the SSNTAB then q entry number in the SSNTAB then

q:= entry number in the SSNTAB;

else

Enter the sequencing symbol in SSNTAB [SSNTAB_ptr]. q:=SSNTAB_ptr].

q:=SSTAB_ptr;

SSNTAB_ ptr:= SSNTAB_ptr + 1;

SSTAB_ptr:=SSTAB_ptr +1;

Replace the sequencing symbol by (S, SSTP+q-1).

(iii) Record the intermediate code of the statement in MDT (MDT_ ptr):

(iv) MDT_ ptr:= MDT ptr + 1;

4. (The statement is a MEND statement)

(a) Record the intermediate code of the statement in MDT [MDT_ptr];

(b) MDT_ ptr:= MDT_ ptr + 1;

(c) If SSNTAB_ptr = 1 (i.e., the SSNTAB is empty) then SSTP := 0;

(d) If #KP= 0 then KPDTP = 0;