arguments in the macro definition are replaced with positional indicators when the definition is stored. The i^{th} dummy argument on the macro name card is represented in the body of the macro by the index marker symbol # where #i is a symbol reserved for the use of the macro processor (i.e., not available to the programmers). These symbols are used in conjunction with the argument list prepared before expansion of a macro call. The symbolic dummy arguments are retained on the macro name card to enable the macro processor to handle argument replacement by name rather than by position.

As an example, consider the macro INCR used in Example 3. The stored macro definition would be:

Macro Definition Table MDT

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

During pass 2 it is necessary to substitute macro call arguments for the index markers stored in the macro definition. Thus upon encountering the call

```
LOOP      INCR      DATA1,DATA2,DATA3
```

the macro call expander would prepare an argument list array:

Argument List Array

8 bytes per entry

Index	*Argument*
0	"LOOP1*bbb*"
1	"DATA1*bbb*"
2	"DATA2*bbb*"
3	"DATA3*bbb*"

(*b* denotes the blank character)

The list would be used only while expanding this particular call. Suppose that a succeeding call were:

```
INCR      &ARG1=DATA3,&ARG2=DATA2,&ARG3=DATA1
```

The macro processor would find that '&ARG1', '&ARG2', and '&ARG3' occupy argument positions 1, 2, and 3 on the macro name card. The resulting argument list array would be:

Argument List Array

Index	*Argument*
0	*"bbbbbbbb"* (all blank)
1	"DATA3*bbb*"
2	"DATA2*bbb*"
3	"DATA1*bbb*"

MACRO DEFINITION TABLE: The Macro Definition Table (MDT) is a table of text lines; if input is from 80-column cards, the MDT can be a table with 80-byte strings as entries. Every line of each macro definition, except the MACRO line, is stored in the MDT. (The MACRO line is useless during macro expansion.) The MEND is kept to indicate the end of the definition; and the macro name line is retained to facilitate keyword argument replacement. Thus, for example, the INCR macro discussed might be stored as follows:

Macro Definition Table

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 Macro Name Table (MNT) serves a function very similar to that of the assembler's Machine-Op Table (MOT) and Pseudo-Op Table (POT). Each MNT entry consists of a character string (the macro name) and a pointer (index) to the entry in the MDT that corresponds to the beginning of the macro definition. The MNT entry for the INCR macro discussed might be:

	8 bytes	4 bytes
Index	*Name*	*MDT index*
⋮	⋮	⋮
3	"INCR*bbbb*"	15
⋮	⋮	⋮

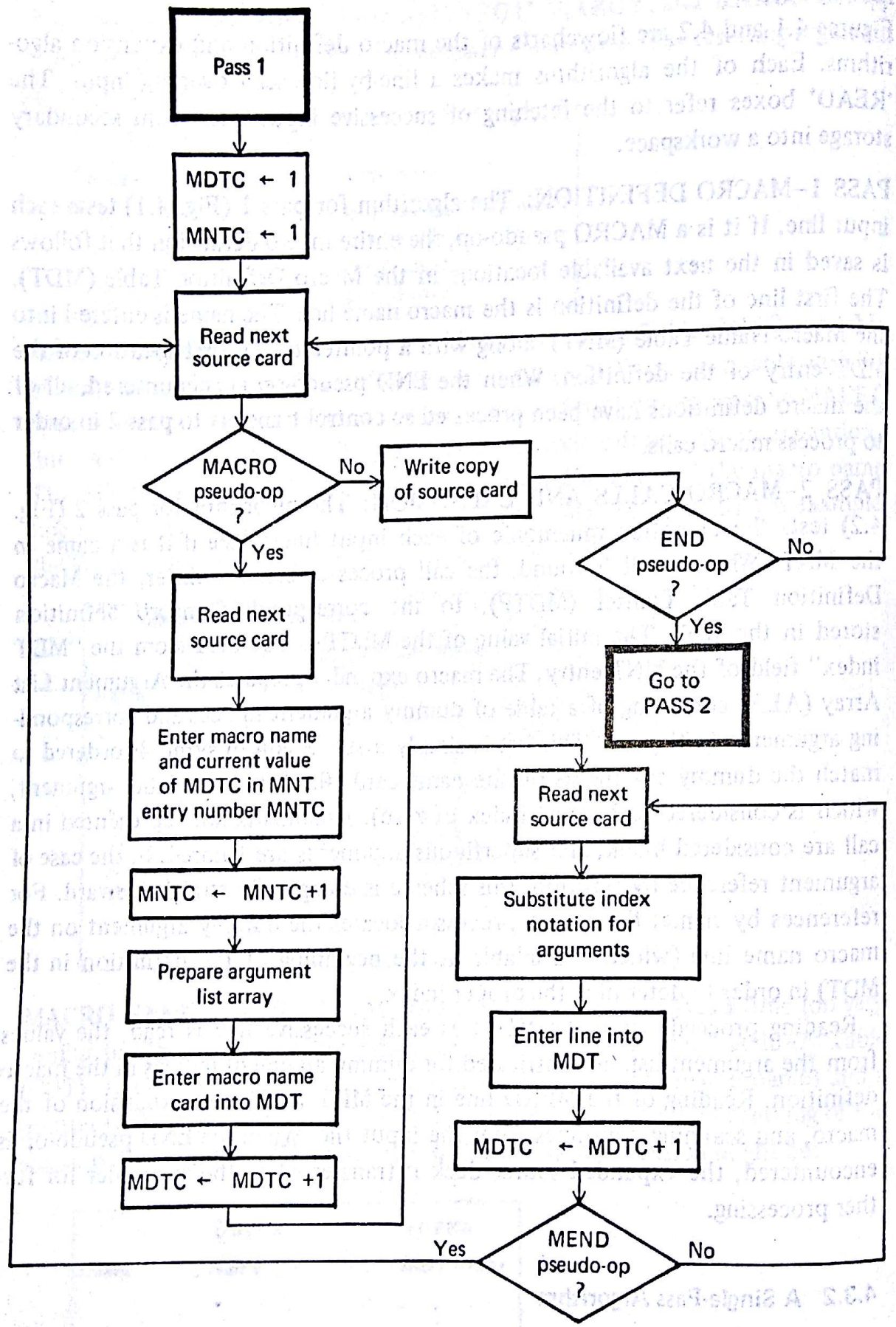

FIGURE 4.1 Pass 1–processing macro definitions

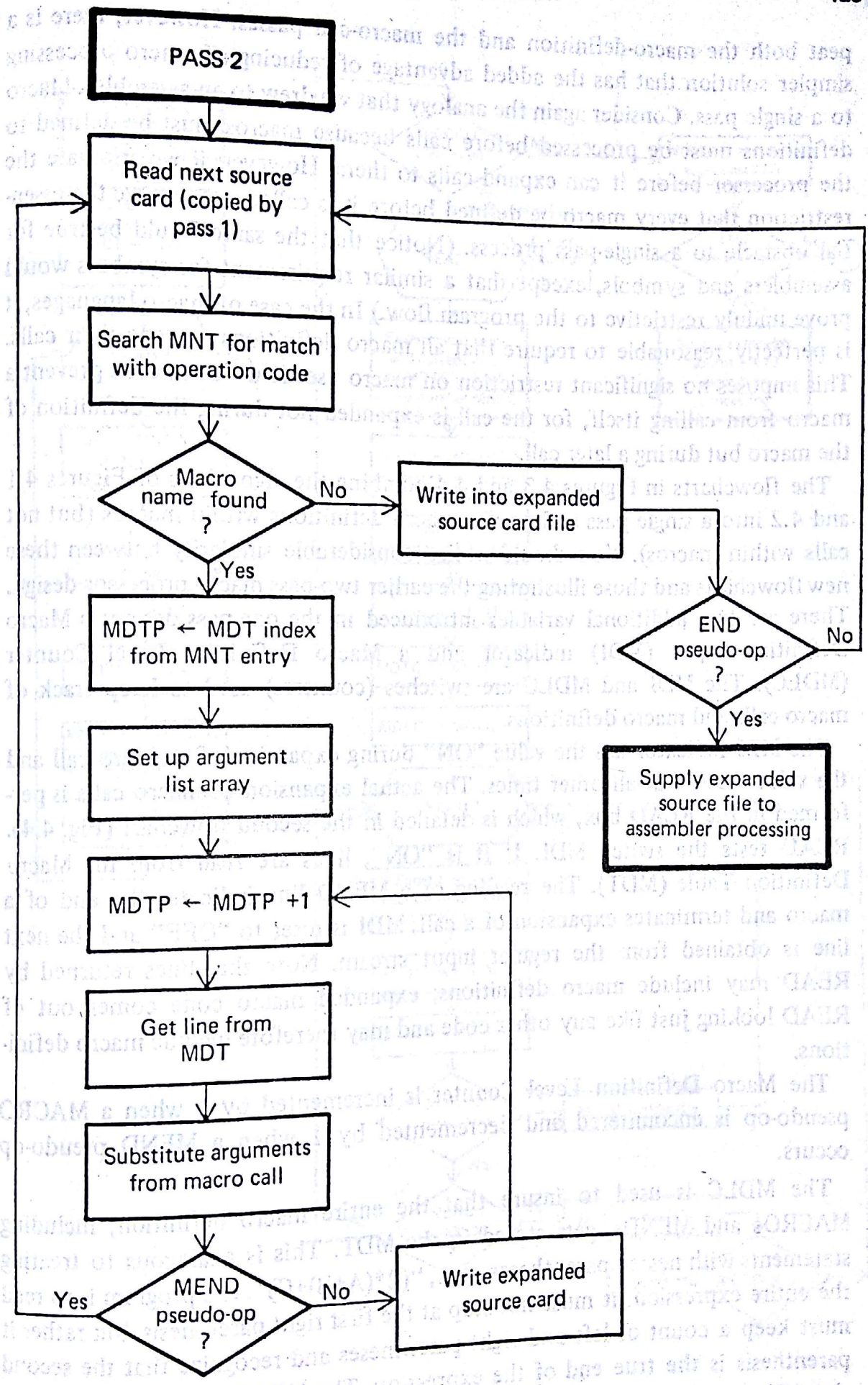

FIGURE 4.2 Pass 2–processing macro calls and expansion

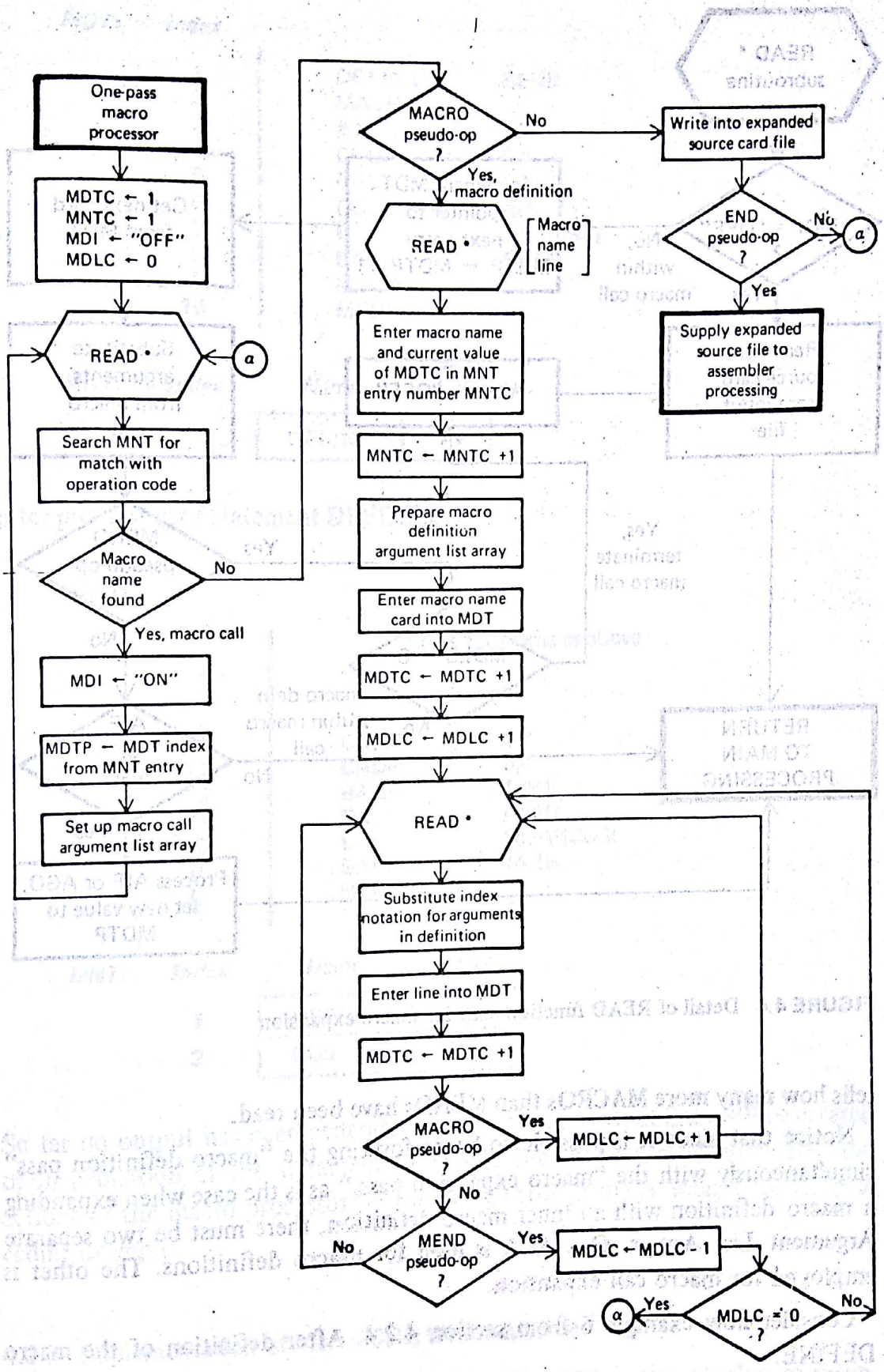

FIGURE 4.3 Simple one-pass macro processor

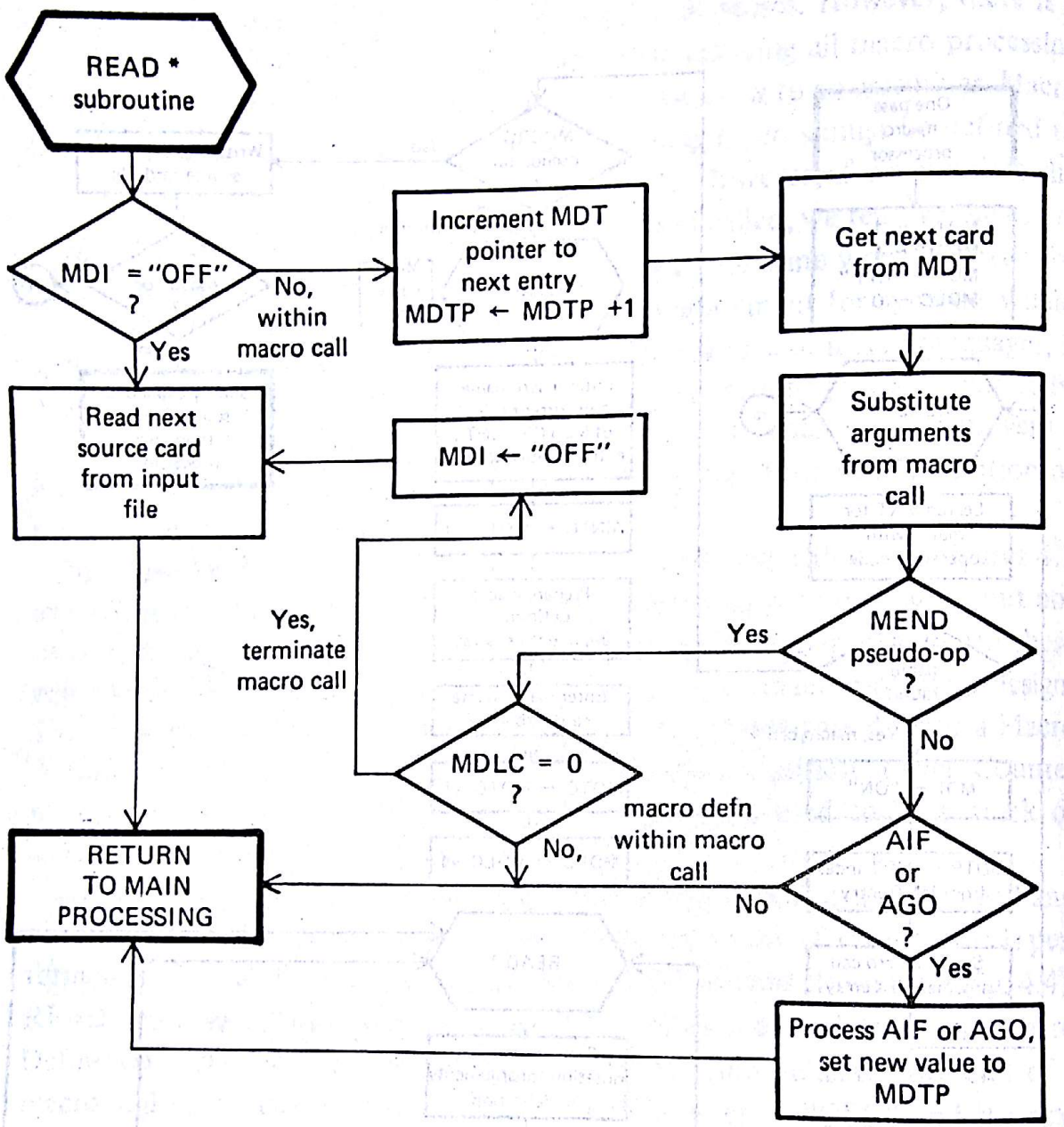

FIGURE 4.4 Detail of READ function used for macro expansion

tells how many more MACROs than MENDs have been read.

Notice that since it is possible to be performing the "macro definition pass" simultaneously with the "macro expansion pass," as is the case when expanding a macro definition with an inner macro definition, there must be two separate Argument List Arrays. One ALA is used for macro definitions. The other is employed for macro call expansion.

Consider now example 6 from section 4.2.4. After definition of the macro DEFINE:

MDT:

Index			
1	DEFINE	&SUB	
2	MACRO		
3	#1	&Y	
4	CNOP	0,4	
5	BAL	1,*+8	
6	DC	A(&Y)	
7	L	15,=V(#1)	
8	BALR	14,15	
9	MEND		
10	MEND		

MNT:

Index	*Name*	*MDT Index*
1	DEFINE	1

after processing of statement DEFINE COS:

MDT:

	Lines 1-10 same as above	
.	.	
.	.	
.	.	
11	COS	&Y
12	CNOP	0,4
13	BAL	1,*+8
14	DC	A(#1)
15	L	15,=V(COS)
16	BALR	14,15
17	MEND	

MNT:

Index	*Name*	*MDT Index*
1	DEFINE	1
2	COS	11

So far no output has been generated by the macro: output lines will be created upon expansion of the statement COS AR. The reader should "simulate" the actions of the macro processor by following through the flowchart for the preceding example.

4.3.3 Implementation of Macro Calls Within Macros

The basic problem in implementing macro calls within macros is that of *recur-*

sion. If a macro call is encountered during the expansion of a macro, the macro processor will have to expand the included macro call and then finish expanding the enclosing macro. The second call might be expanded by a second macro processor, which would look up the macro definition in the MDT and return the expanded code to the first macro processor. Having many macro processors is neither efficient nor general; however, if a single macro processor is to handle such nested macro calls, it must in some way save its status when it encounters nested calls.

Example 5 illustrates this problem. It defines two macros, ADDS and ADD1. The macro definitions in the MDT are shown in Figure 4.5.

Macro Definition Table MDT

Index	*Contents*	
1	ADD1	&ARG
2	L	1,#1
3	A	1,=F'1'
4	ST	1,#1
5	MEND	
6	ADDS	&ARG1,&ARG2,&ARG3
7	ADD1	#1
8	ADD1	#2
9	ADD1	#3
10	MEND	

FIGURE 4.5 MDT after macro definition of example 5

Consider the action of the macro processor of Figure 4.3 and 4.4 when it encounters the macro call

```
ADDS        DATA1, DATA2, DATA3
```

Our algorithm will prepare a macro call argument list array and set a pointer, MDTP, to line 6 of the MDT; switch MDI is set to "ON". The READ function increments MDTP, gets its next line from the MDT (line 7) and substitutes the argument, yielding the line

```
ADD1        DATA1
```

Now our algorithm is in trouble: it has encountered another macro call. ADD1 is a macro name, so the processor will prepare a new argument list array, set MDTP to line 1 of the MDT, and set MDI to "ON", its current value. The ADD1 macro will expand correctly; at the end MDTP will point to MDT line 5. MDI will be re-

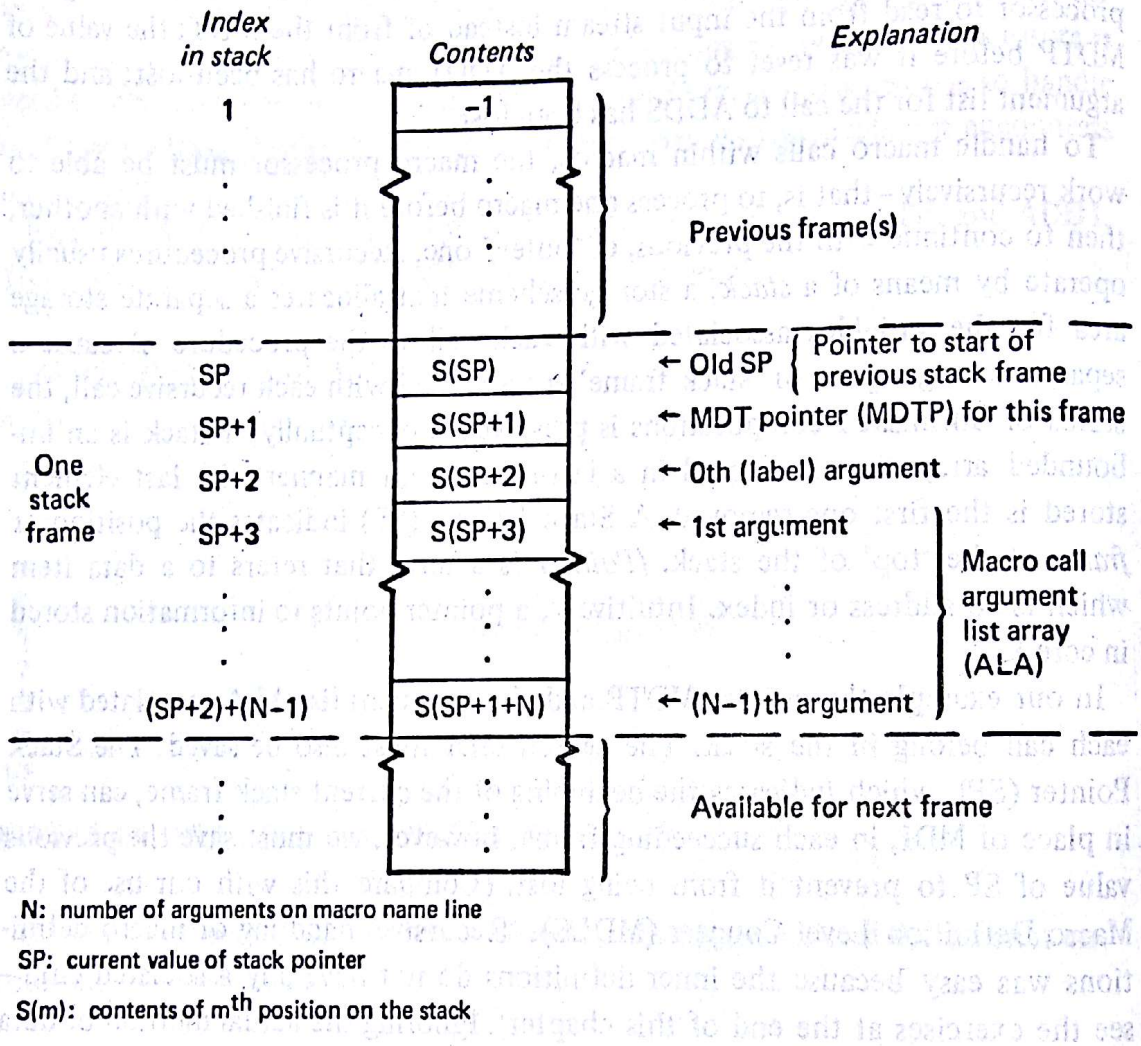

FIGURE 4.6 Stack organization

should confirm the fact that the algorithm described in these new flowcharts is indeed very similar to the previous algorithm (Figs. 4.3 and 4.4). Notice that the number of entries in a stack frame depends upon the number of elements in the argument list (i.e., the number of dummy arguments in the macro definition).

It is desirable for the reader to hand-simulate the operation of the macro processor by using the algorithm to expand a macro source program. Figure 4.9 illustrates the state of the stack and expanded source at various key points in the processing of Example 5. The corresponding Macro Definition Table (MDT) for this example was presented in Figure 4.5. It is strongly recommended that the reader go through all the steps of the macro expansion and compare results with Figures 4.9a and 4.9b.

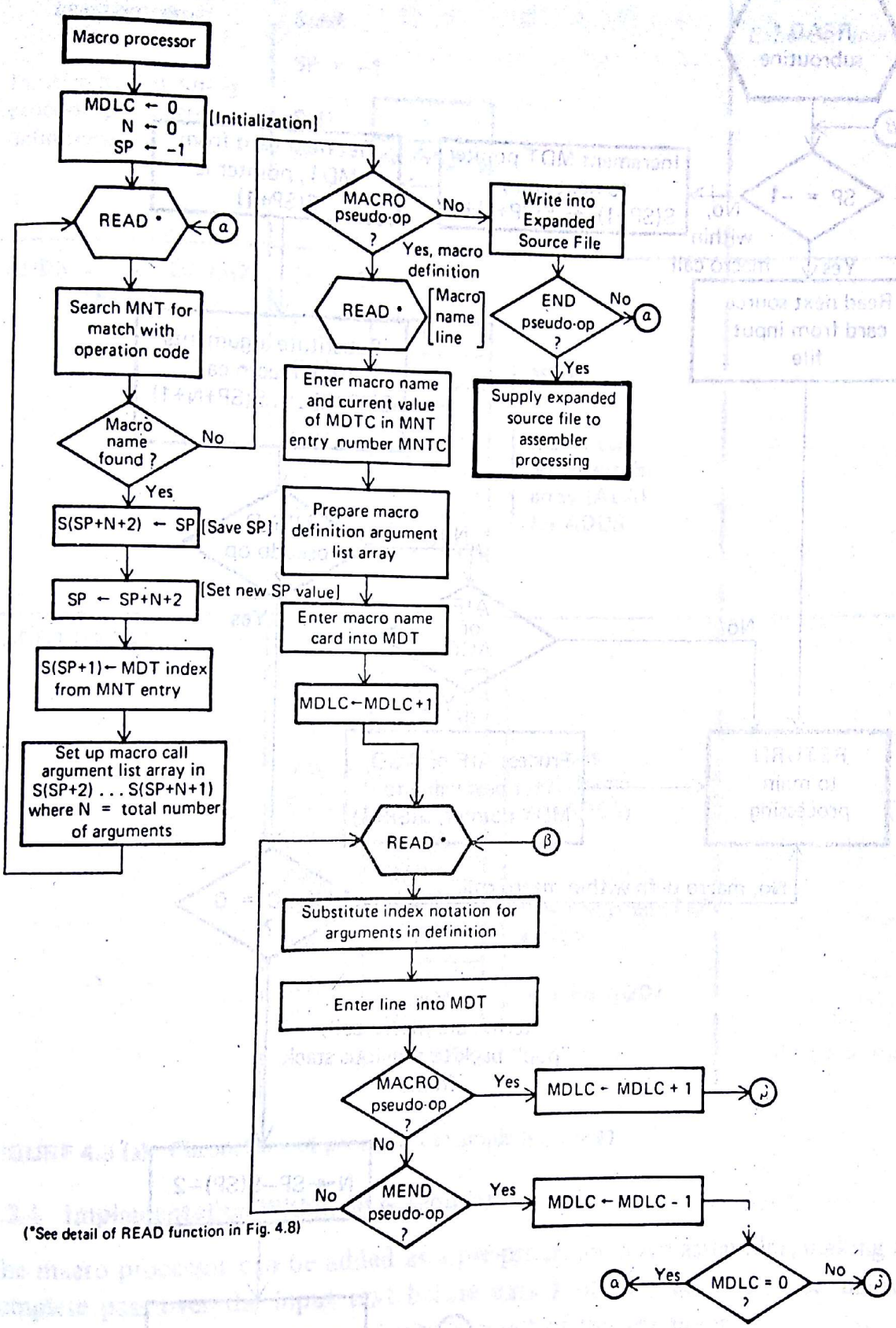

FIGURE 4.7 One-pass macro processor capable of handling macro calls within macro definitions

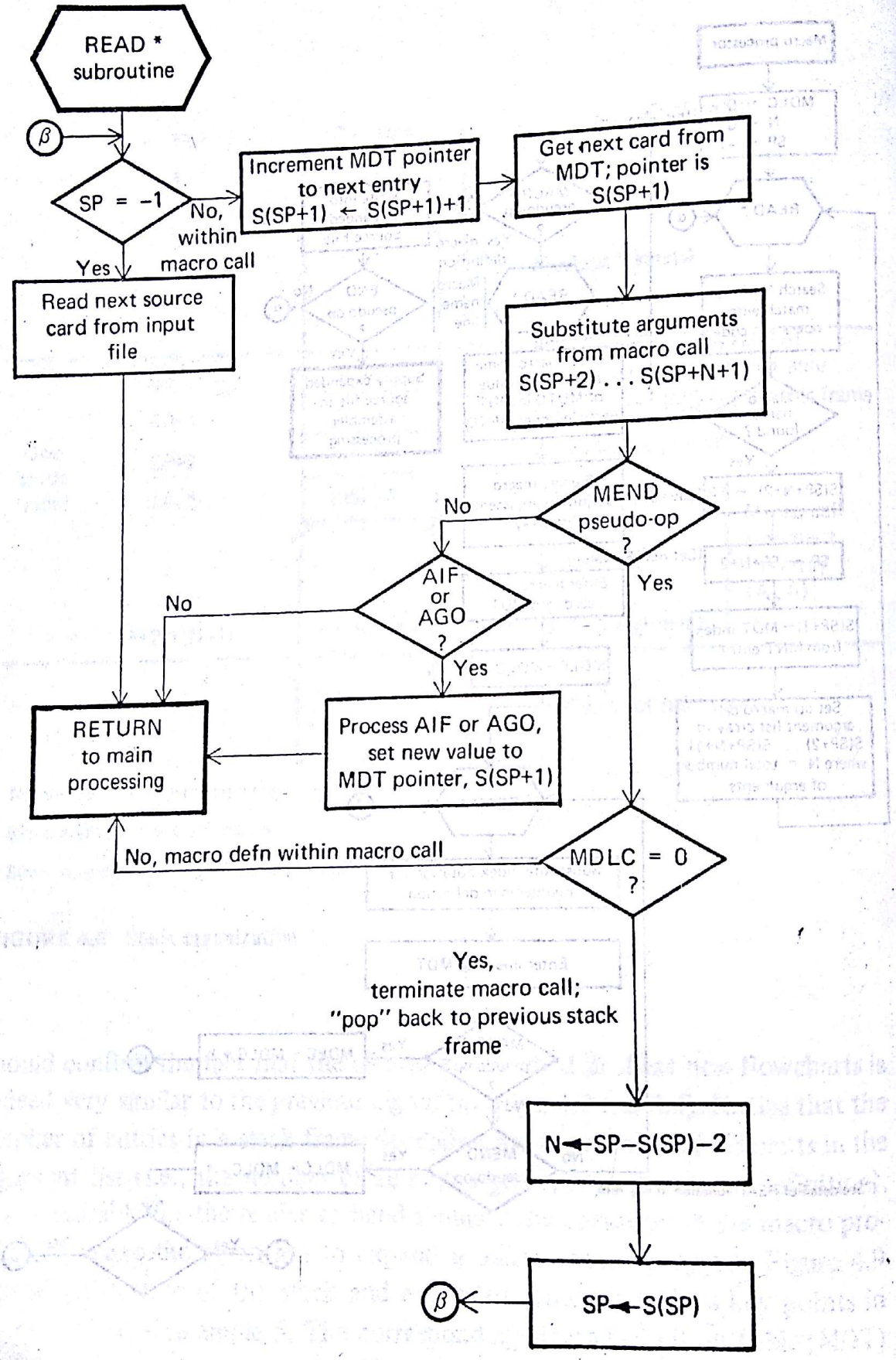

FIGURE 4.8 Detail of READ function for recursive macro expansion

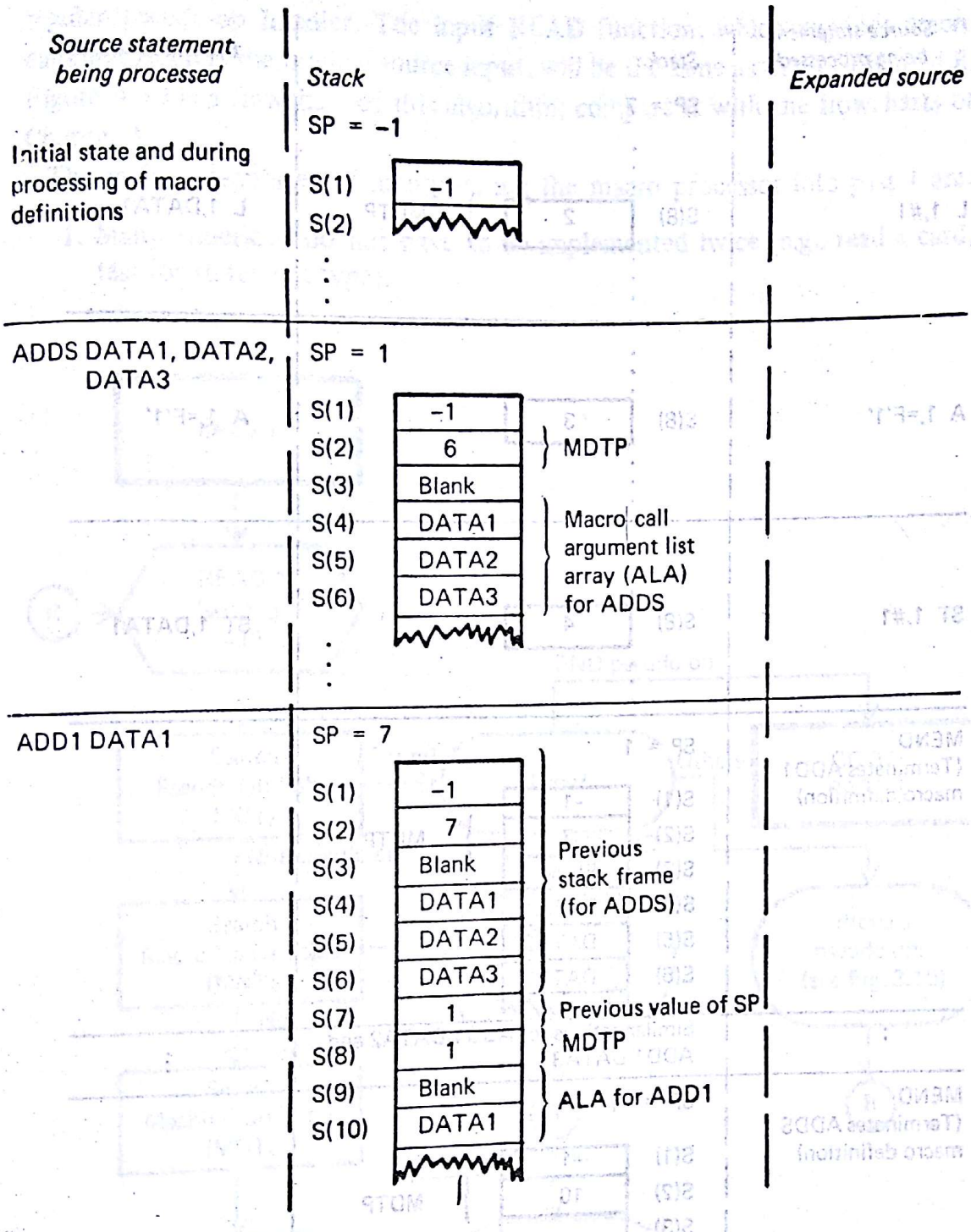

FIGURE 4.9 (a) Chronology of processing example 5 (part 1)

4.3.4 Implementation Within an Assembler

The macro processor can be added as a pre-processor to an assembler, making a complete pass over the input text before pass 1 of the assembler. The macro processor can also be implemented within pass 1 of the assembler.

The implementation of the macro processor within pass 1 eliminates the overhead of intermediate files, and we can improve this integration of macro pro-

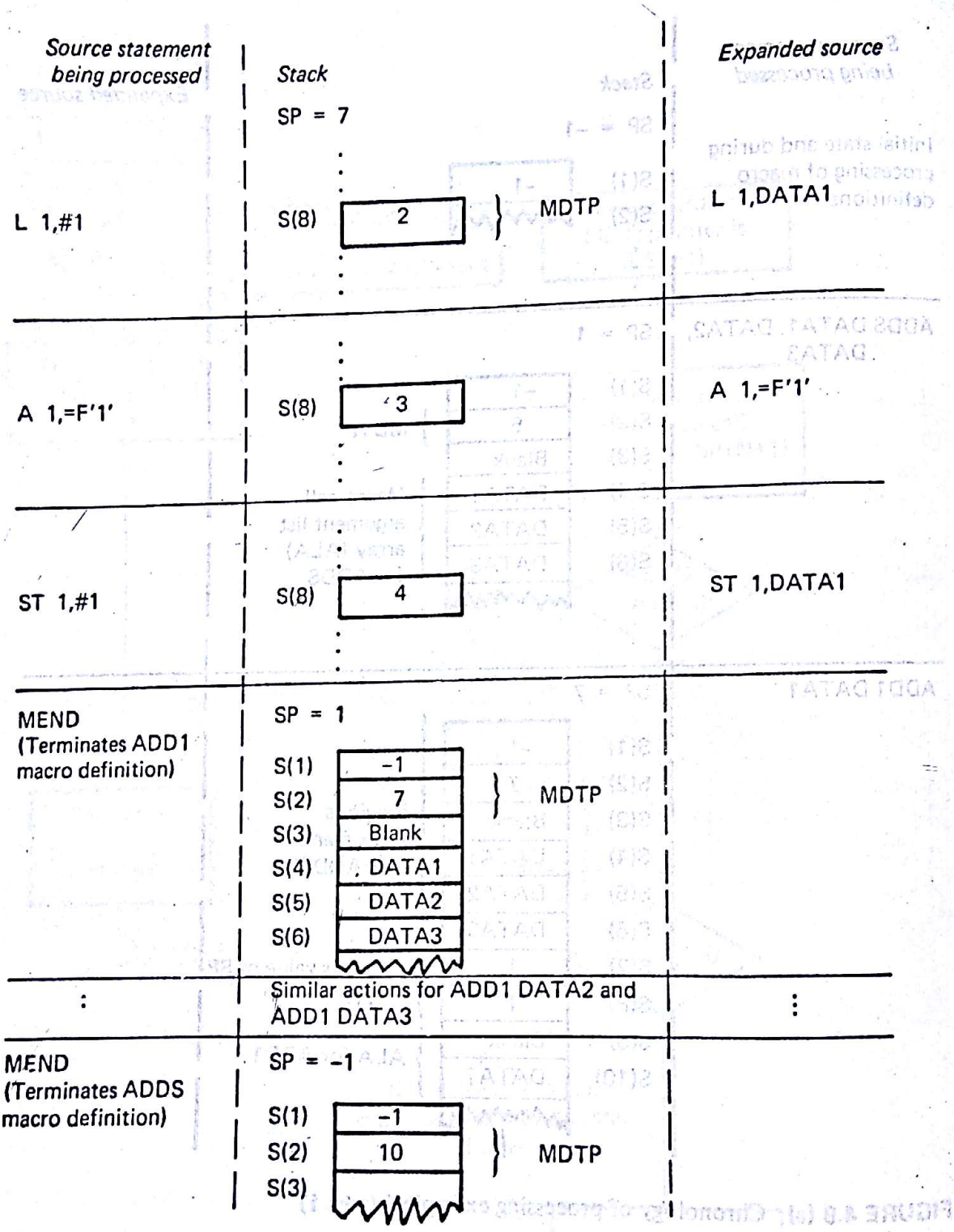

FIGURE 4.9 (b) Chronology of processing example 5 (part 2)

cessor and assembler by combining similar functions. For example, the Macro Name Table (MNT) can be combined with the assembler's op-code table (MOT or POT); a flag in each entry indicates whether or not it is a macro name. MACRO pseudo-ops to the macro processor can be detected by the assembler's

regular pseudo-op handler. The input READ function, which expands macro calls and receives the original source input, will be the same as that of Figure 4.8. Figure 4.10 is a flowchart of this algorithm; compare it with the flowcharts of Chapter 3.

The major advantages of incorporating the macro processor into pass 1 are:

1. Many functions do not have to be implemented twice (e.g., read a card, test for statement type).

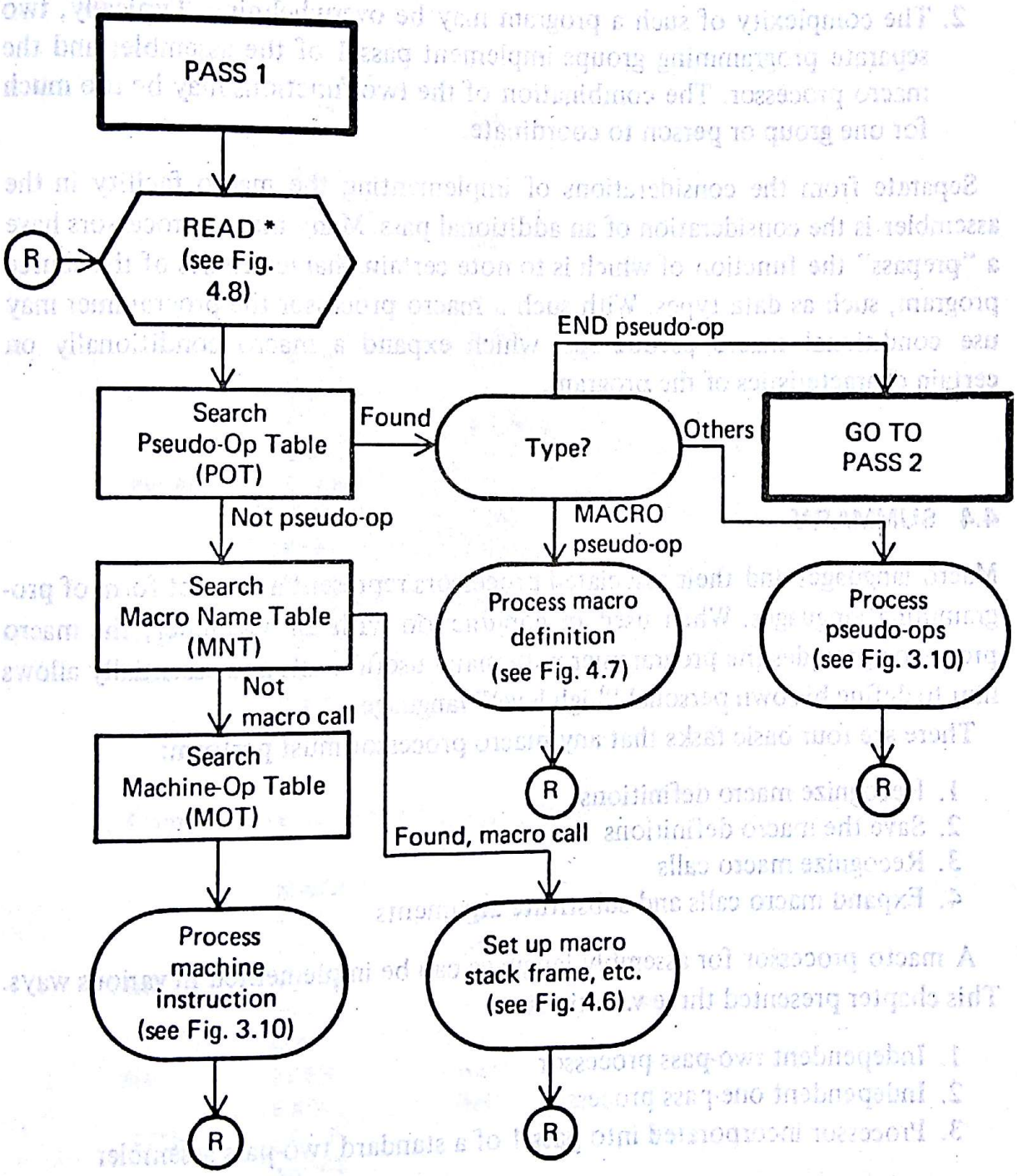

FIGURE 4.10 A macro processor combined with assembler pass 1