PVG's College of Engineering & S. S. Dhamankar Institute of Management, Nashik
206, Behind Reliance Petrol Pump, Dindoricad, Mhasrul, Nashik-422004
Phone: 0253-6480000 / 36 /44
Toll Free Number: 18002665330
Website: https://pvgcoenashik.org
Email: admission@pvgcoenashik.org
Approved by AICTE, New Delhi, DTE, Mumbai and Affiliated to Savitribai Phule Pune University, Pune.
DTE Code: EN5330

"MACRO PROCESSOR AND COMPILER "

Prepared By

Prof. Anand N. Gharu

(Assistant Professor)

PVGCOE Computer Dept.

CLASS : TE COMPUTER 2019

SUBJECT : SPOS (SEM-I)

UNIT : II

02 AUG 2021

SYLLABUS :-

Introduction, Features of a Macro facility: Macro instruction arguments, Conditional Macro expansion, Macro calls within Macros, Macro instructions, Defining Macro, Design of two pass Macro processor, Concept of single pass Macro processor.

Introduction to Compilers: Phases of Compiler with one example, Comparison of Compiler and Interpreter.

CONTENTS :-

1. Introduction, Features of a Macro facility: Macro instruction arguments,
2. Conditional Macro expansion,
3. Macro calls within Macros
4. Macro instructions, Defining Macro
5. Macro Definition (Macro Processor)
6. Compare Macro & Subroutines
7. Concept of single pass Macro processor
8. Introduction to Compilers: Phases of Compiler with one example, Comparison of Compiler and Interpreter

MACRO DEFINITION

Writing a macro is another way of ensuring modular programming in assembly language.

- A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
- In NASM, macros are defined with **%macro** and **%endmacro** directives.
- The macro begins with the %macro directive and ends with the %endmacro directive.

MACRO DEFINITION

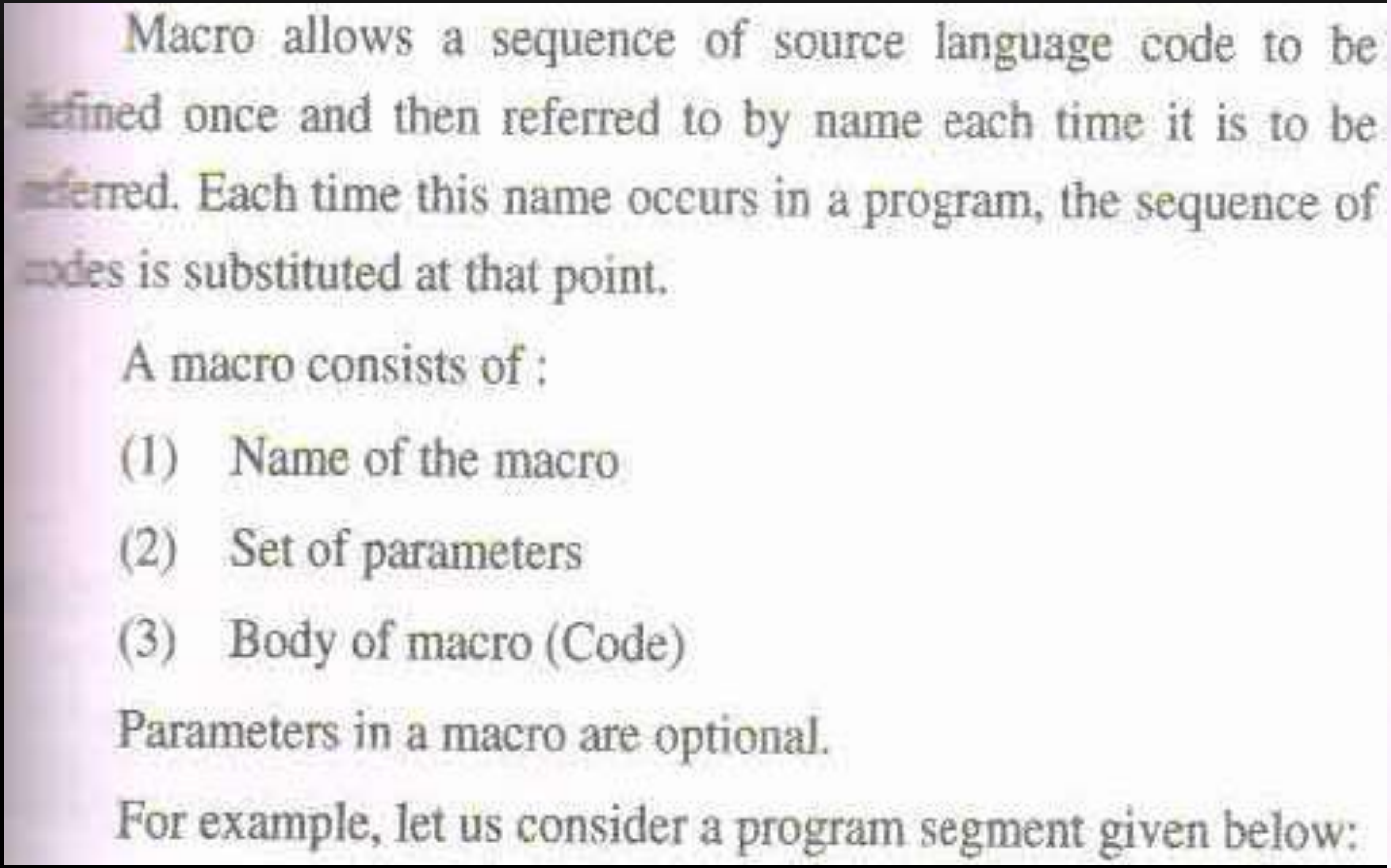
The Syntax for macro definition −

```
%macro macro_name number_of_params

<macro body>

%endmacro
```

FEATURES OF MACROPROCESSOR

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

MACRO DEFINITION



Macro allows a sequence of source language code to be defined once and then referred to by name each time it is to be referred. Each time this name occurs in a program, the sequence of codes is substituted at that point.

A macro consists of :

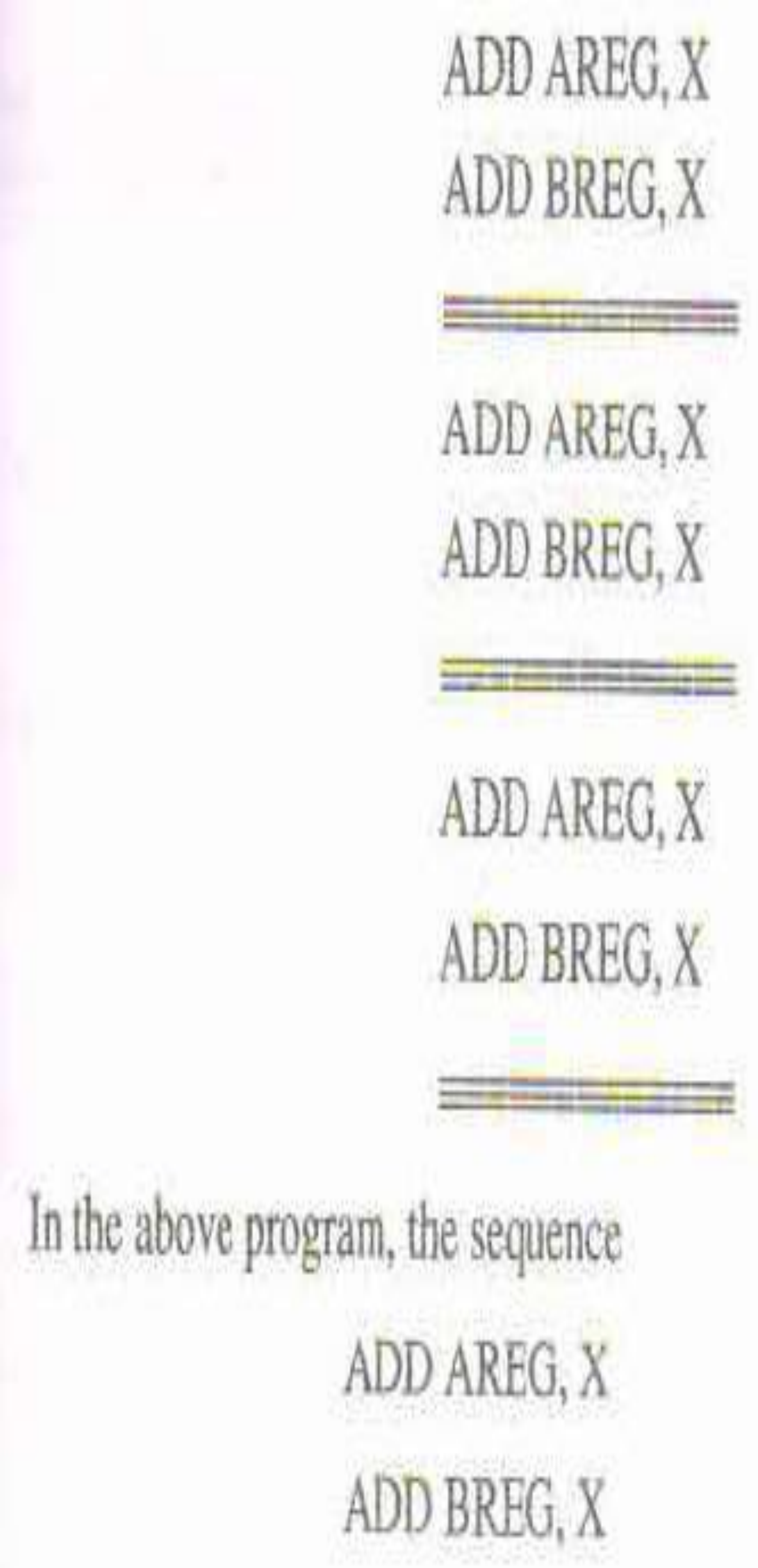
(1) Name of the macro

(2) Set of parameters

(3) Body of macro (Code)

Parameters in a macro are optional.

For example, let us consider a program segment given below:



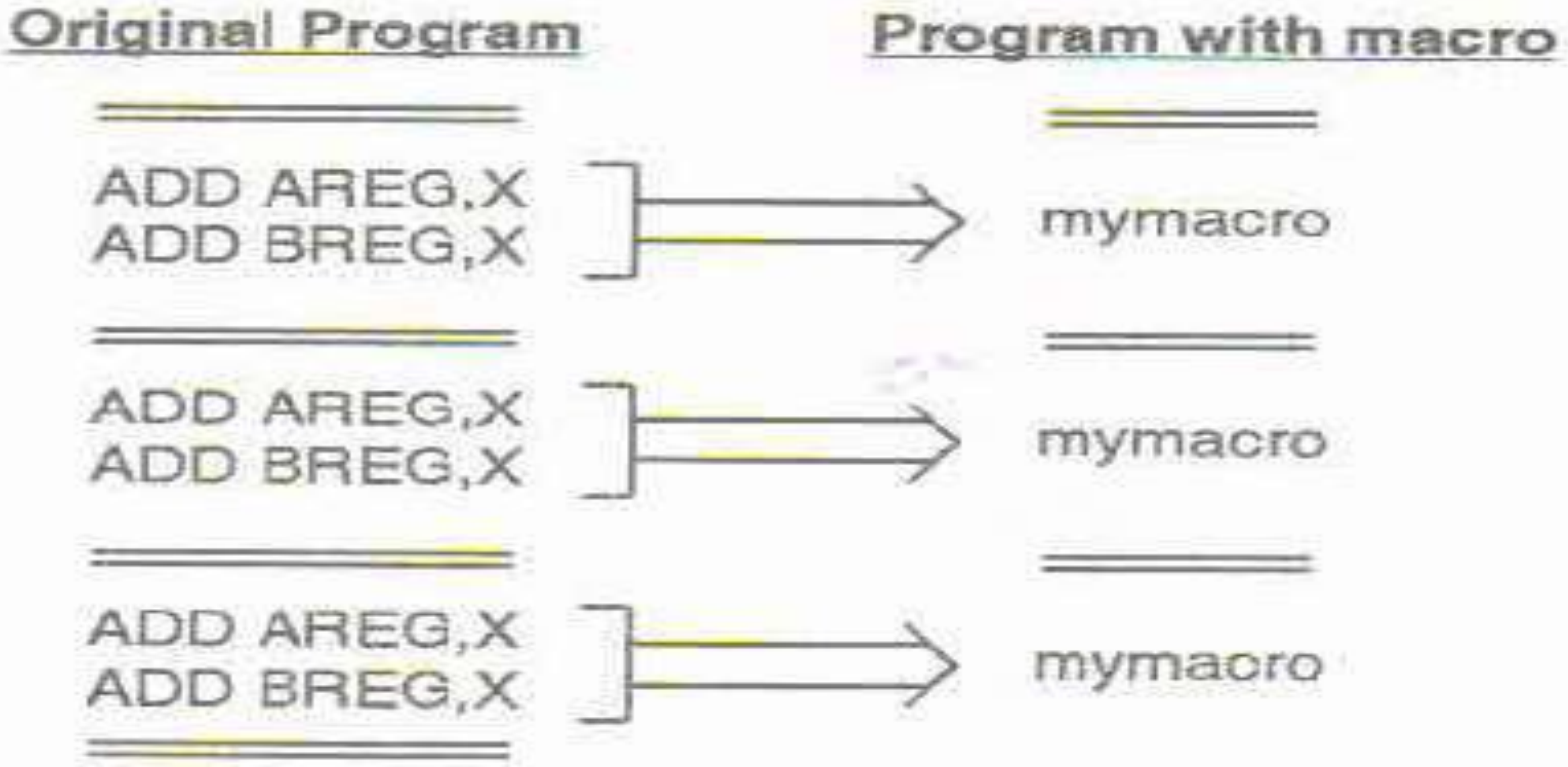
```
ADD AREG, X
ADD BREG, X

ADD AREG, X
ADD BREG, X

ADD AREG, X
ADD BREG, X
```

In the above program, the sequence

```
ADD AREG, X
ADD BREG, X
```

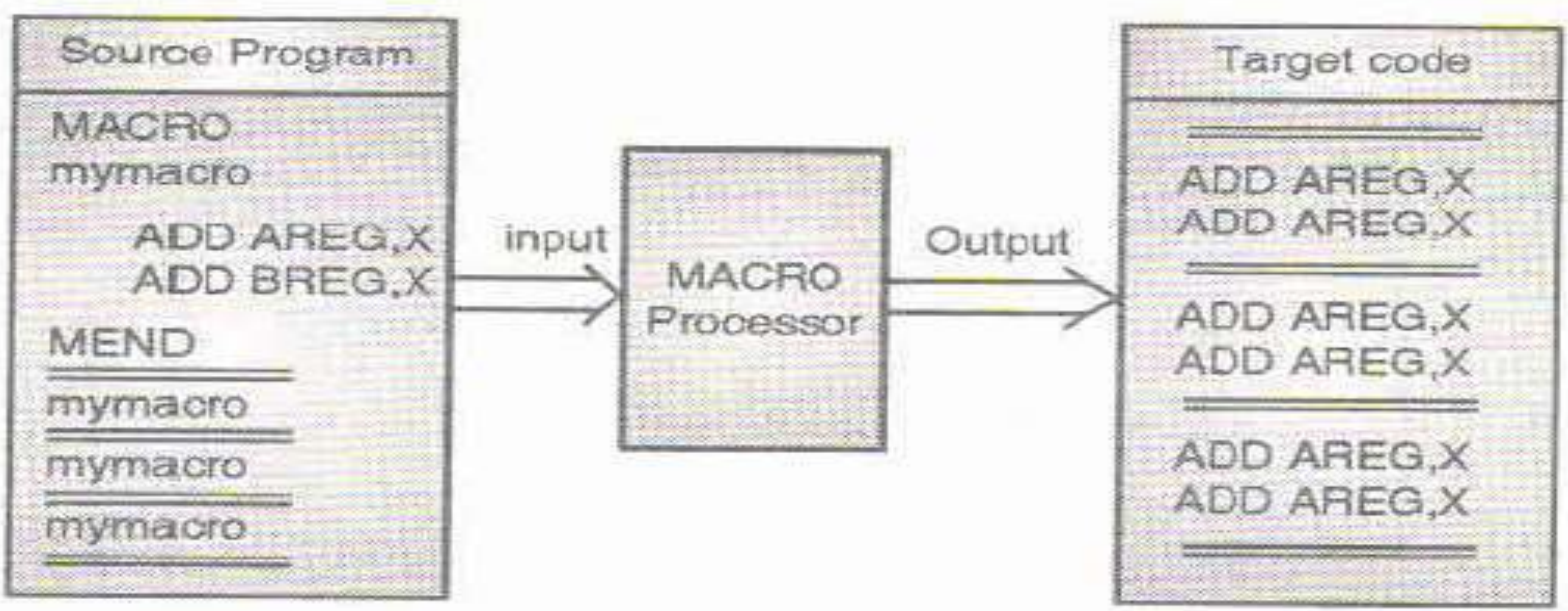
Start of definition	**MACRO**
macro name	mymacro
macro body	ADD AREG, X ADD BREG, X
End of macro definition	MEND

The example given above can be expressed with macro. It is given in Fig. 2.1.1.



(S3.1)**Fig. 2.1.1 : Re-writing a program with macros**

A macro processor takes a source with macro definition and macro calls and replaces each macro call with its body.

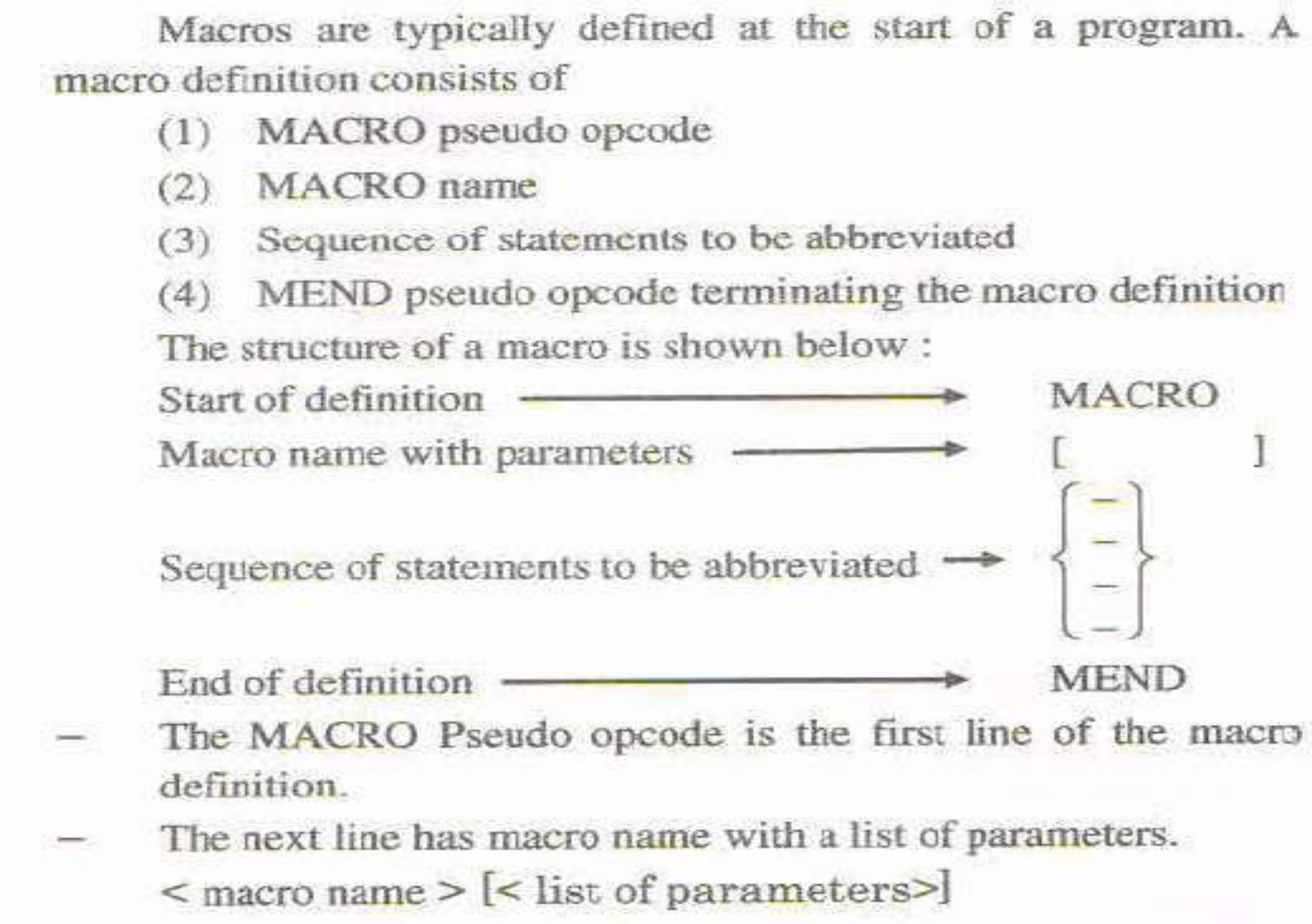


(S3.2)**Fig. 2.1.2 : Macro expansion**

COMPARE MACRO & SUBROUTINE

Sr. No	Macro	Subroutine
1	Macro can be called only in the program it is defined.	Subroutine can be called from other programs also.
2	More space is required	Less space is required
3	Execution speed is faster.	Execution speed is slower
4	Macro Can not handle labels	Subroutines can handle labels
5	Macro is executed by assembler	Subroutine is executed by hardware
6	Code size increase	Code size does not increase.
7	Simple to write and use	Complex to write and understand
8	Macro name[parameter] Mend	Subroutine name(parameter) end

DEFINING MACRO



Macros are typically defined at the start of a program. A macro definition consists of

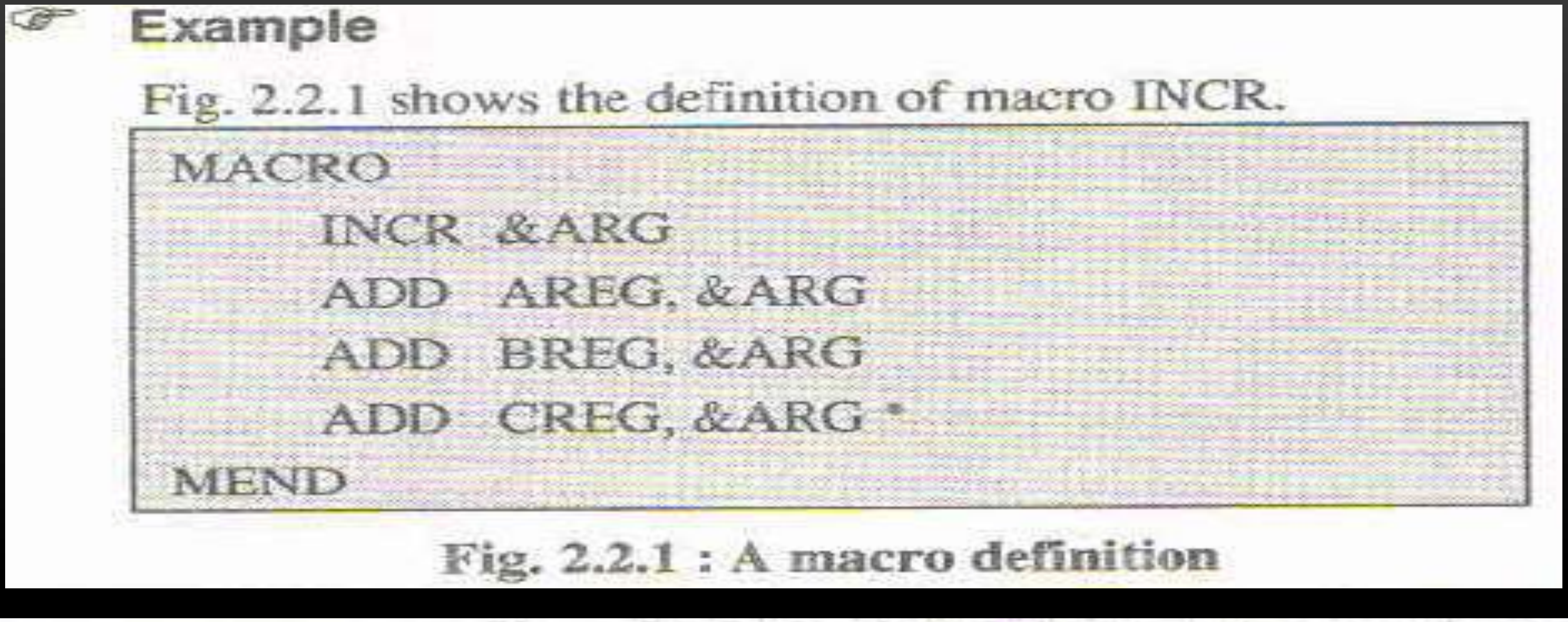
(1) MACRO pseudo opcode
(2) MACRO name
(3) Sequence of statements to be abbreviated
(4) MEND pseudo opcode terminating the macro definition

The structure of a macro is shown below :

Start of definition → MACRO
Macro name with parameters → [ ]
Sequence of statements to be abbreviated → { — — — — }
End of definition → MEND

- The MACRO Pseudo opcode is the first line of the macro definition.
- The next line has macro name with a list of parameters.

< macro name > [< list of parameters>]

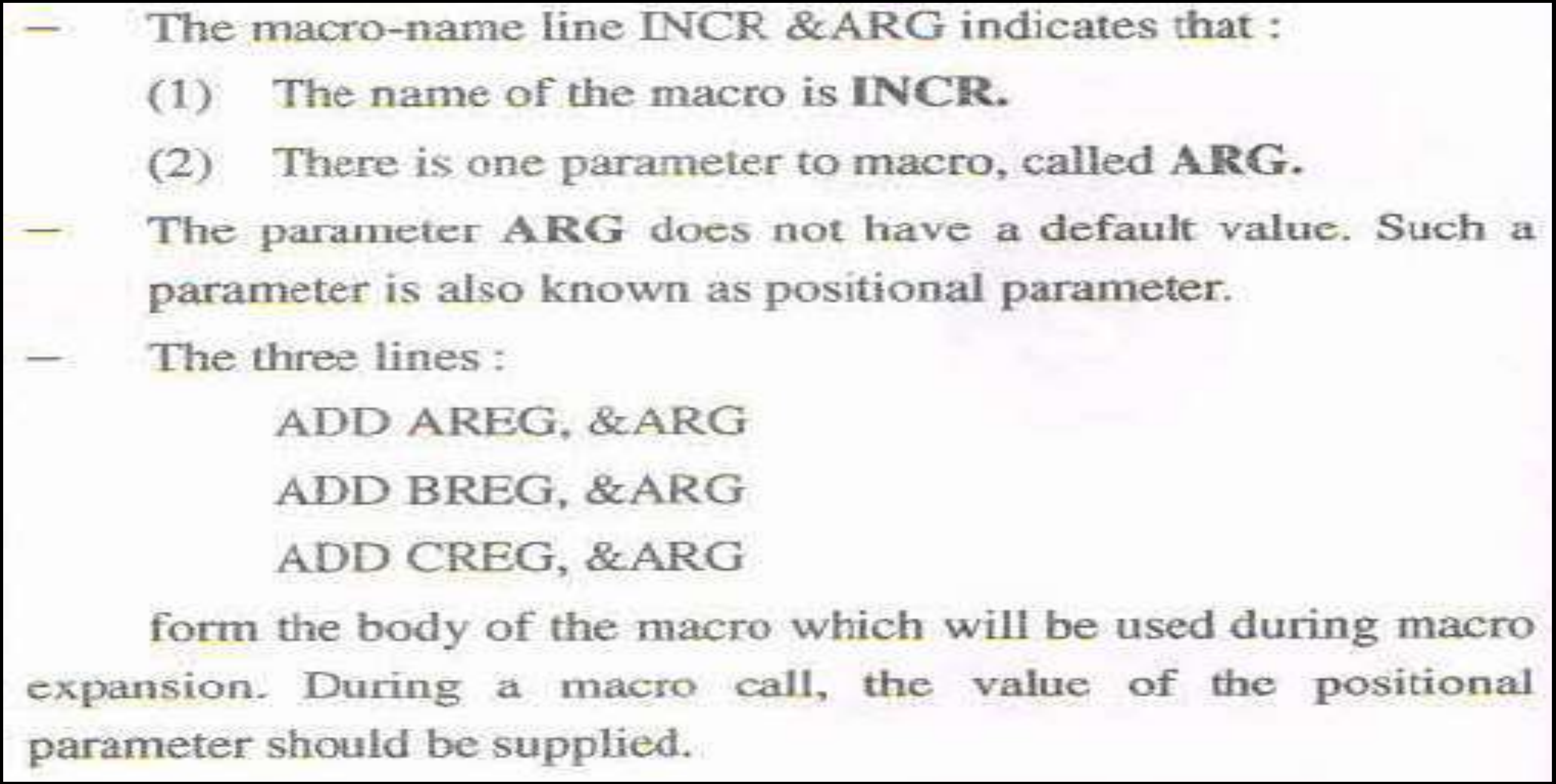


☞ **Example**

Fig. 2.2.1 shows the definition of macro INCR.

```
MACRO
        INCR  &ARG
        ADD   AREG, &ARG
        ADD   BREG, &ARG
        ADD   CREG, &ARG
MEND
```

Fig. 2.2.1 : A macro definition

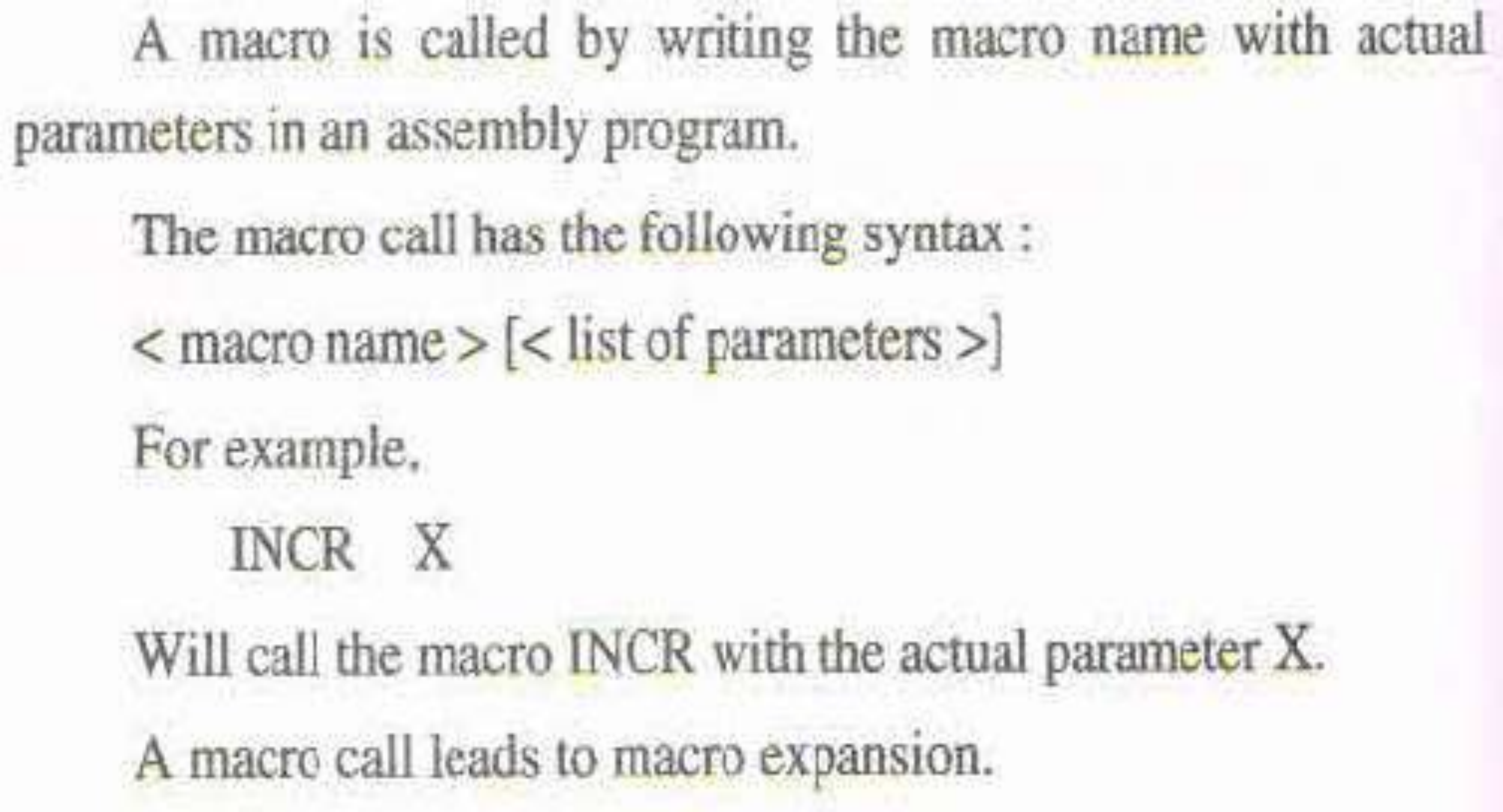


- The macro-name line INCR &ARG indicates that :
  - (1) The name of the macro is **INCR.**
  - (2) There is one parameter to macro, called **ARG.**
- The parameter **ARG** does not have a default value. Such a parameter is also known as positional parameter.
- The three lines :

```
ADD AREG, &ARG
ADD BREG, &ARG
ADD CREG, &ARG
```

form the body of the macro which will be used during macro expansion. During a macro call, the value of the positional parameter should be supplied.

CALLING MACRO



A macro is called by writing the macro name with actual parameters in an assembly program.

The macro call has the following syntax :

< macro name > [< list of parameters >]

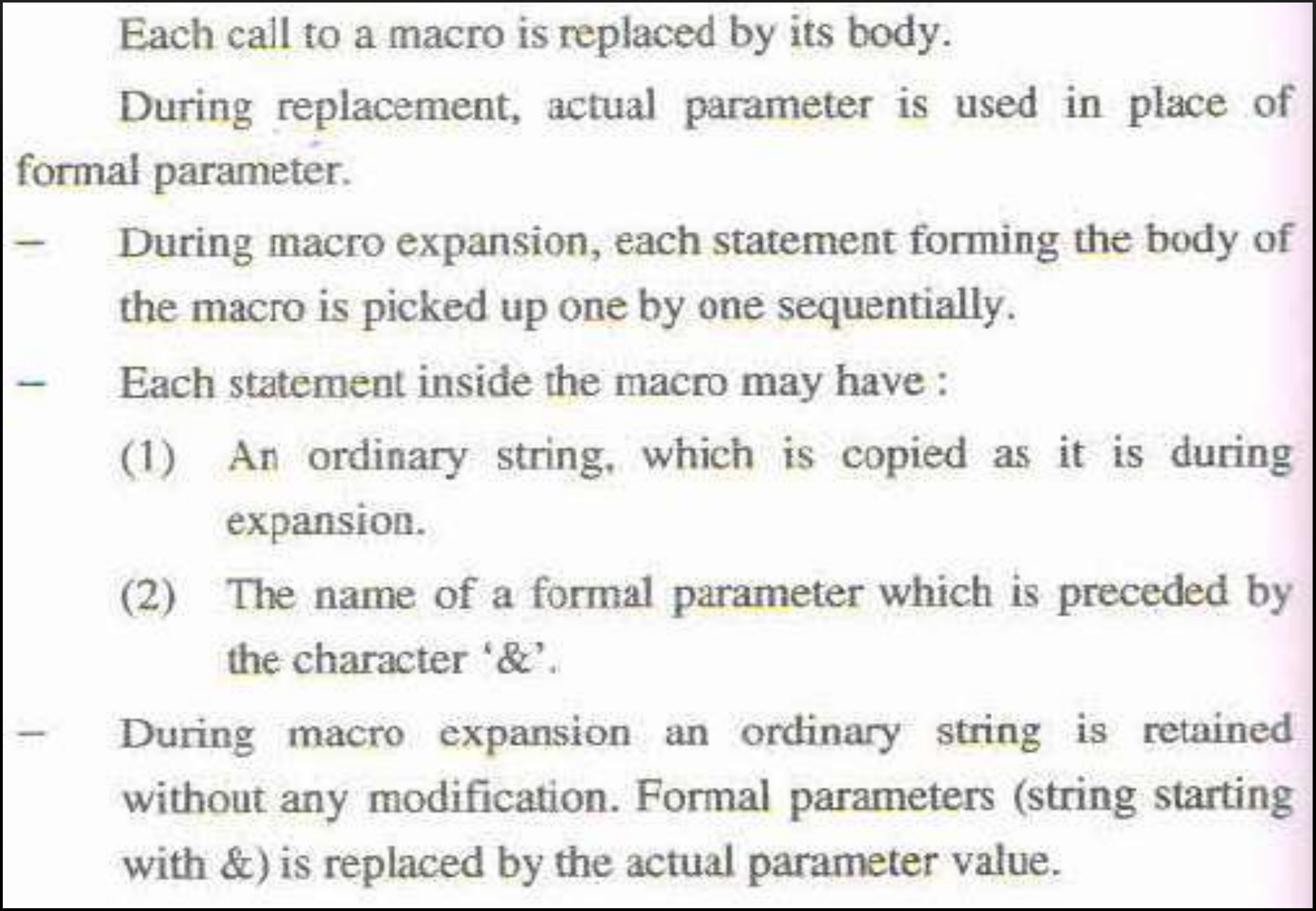
For example,

```
    INCR    X
```

Will call the macro INCR with the actual parameter X.

A macro call leads to macro expansion.

MACRO EXPANSION



Each call to a macro is replaced by its body.

During replacement, actual parameter is used in place of formal parameter.

- During macro expansion, each statement forming the body of the macro is picked up one by one sequentially.
- Each statement inside the macro may have :
  1. (1) An ordinary string, which is copied as it is during expansion.
  2. (2) The name of a formal parameter which is preceded by the character '&'.
- During macro expansion an ordinary string is retained without any modification. Formal parameters (string starting with &) is replaced by the actual parameter value.

Consider a macro as given in Fig. 2.2.2

```
MACRO
   INCR     &VARIABLE,
&INCR_BY,&USE_REG
   MOVER      &USE_REG, &VARIABLE
   ADD        &USE_REG, &INCR_BY
   MOVEM      &USE_REG, &VARIABLE
MEND
```

Fig. 2.2.2 : A macro definition

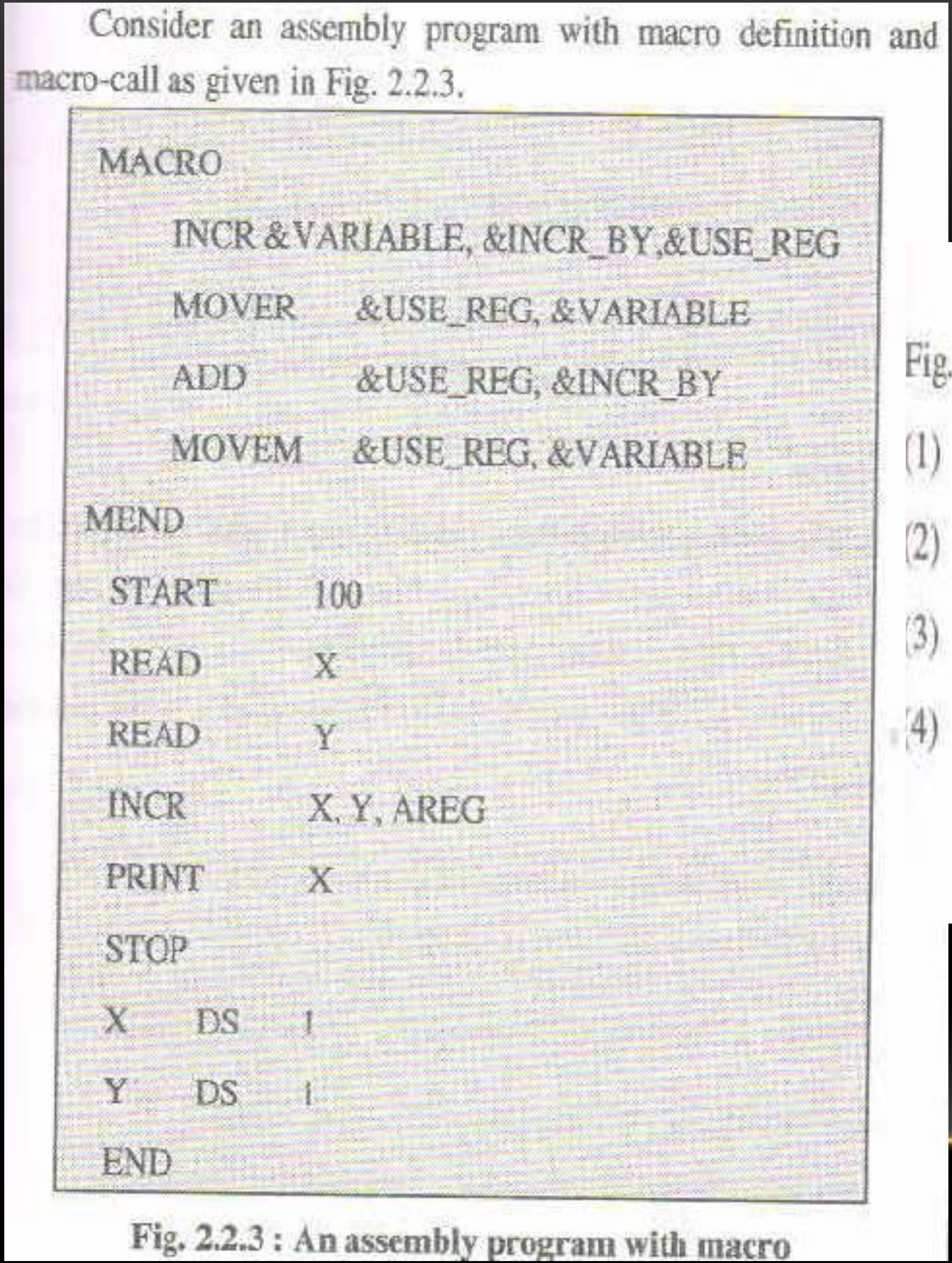
- Name of the macro is **INCR**
- There are three positional parameters

  These parameters are :

  (1) VARIABLE

  (2) INCR_BY

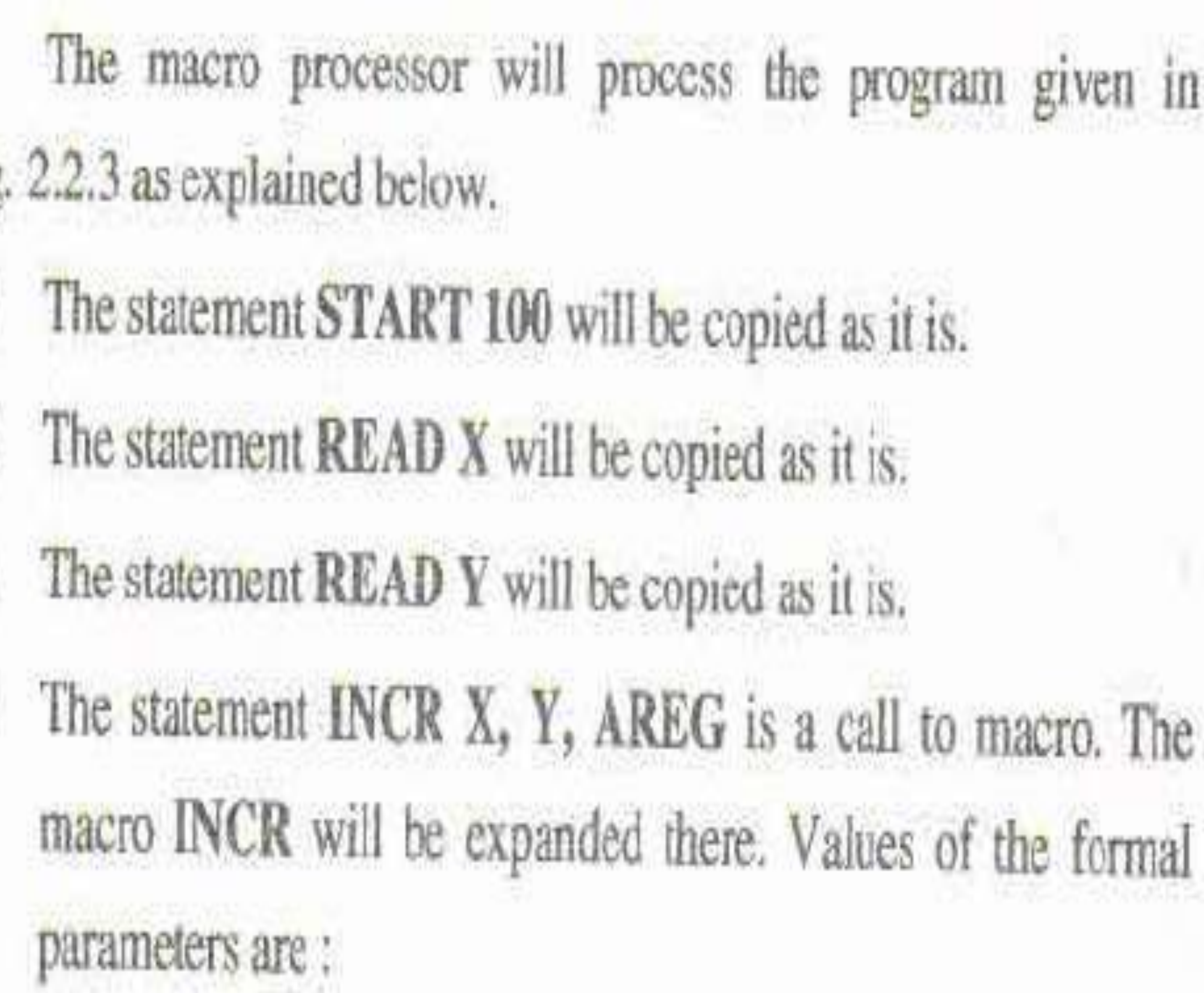
  (3) USE_REG
- The body of macro **INCR** contains three statements.



Consider an assembly program with macro definition and macro-call as given in Fig. 2.2.3.

```
MACRO
    INCR &VARIABLE, &INCR_BY,&USE_REG
    MOVER    &USE_REG, &VARIABLE
    ADD      &USE_REG, &INCR_BY
    MOVEM    &USE_REG, &VARIABLE
MEND
 START    100
 READ     X
 READ     Y
 INCR     X, Y, AREG
 PRINT    X
 STOP
 X   DS   1
 Y   DS   1
 END
```

Fig. 2.2.3 : An assembly program with macro



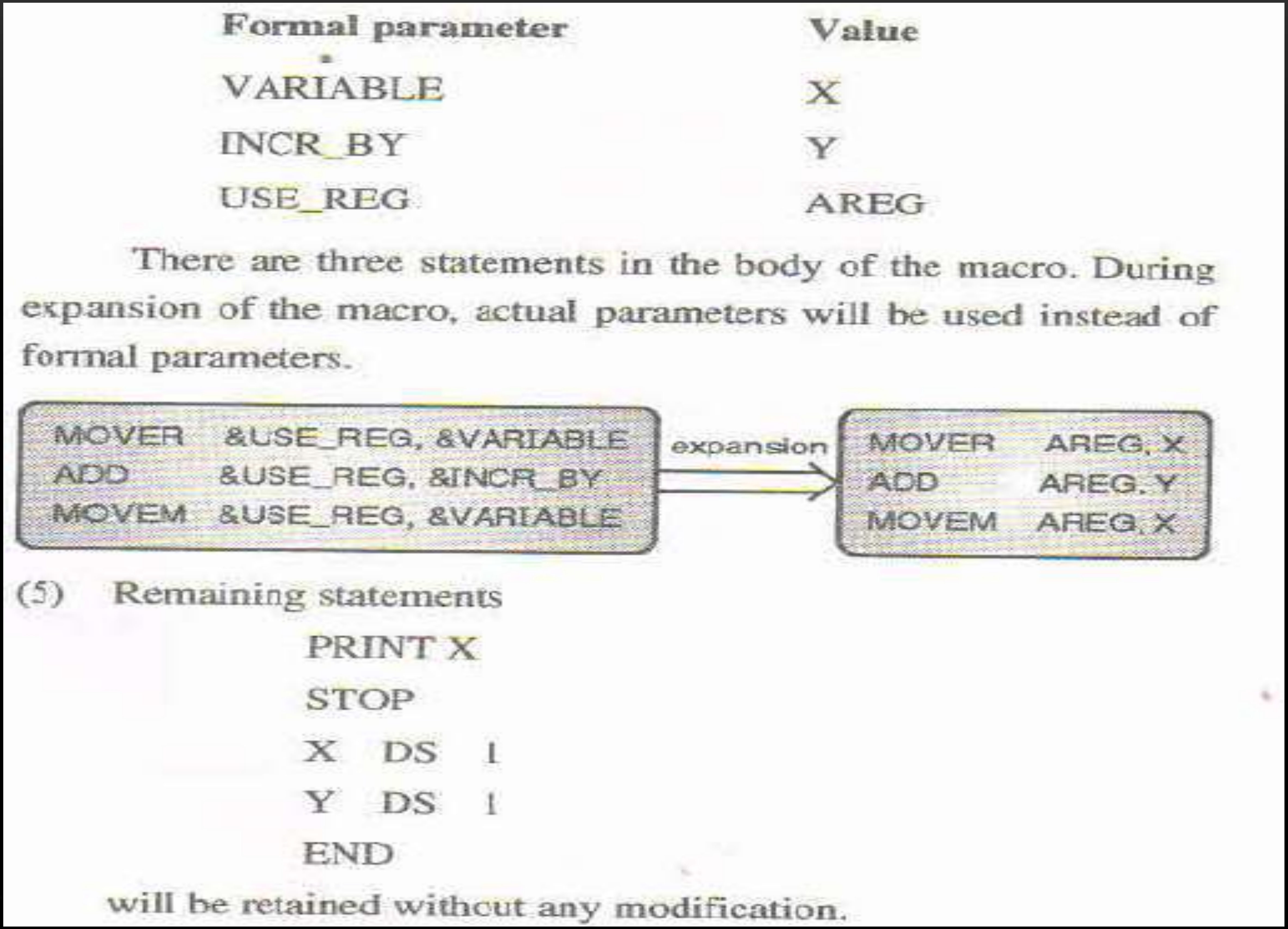
The macro processor will process the program given in Fig. 2.2.3 as explained below.

(1) The statement **START 100** will be copied as it is.

(2) The statement **READ X** will be copied as it is.

(3) The statement **READ Y** will be copied as it is.

(4) The statement **INCR X, Y, AREG** is a call to macro. The macro **INCR** will be expanded there. Values of the formal parameters are :



Formal parameter	Value
VARIABLE	X
INCR_BY	Y
USE_REG	AREG

There are three statements in the body of the macro. During expansion of the macro, actual parameters will be used instead of formal parameters.

```
MOVER   &USE_REG, &VARIABLE
ADD     &USE_REG, &INCR_BY
MOVEM   &USE_REG, &VARIABLE
```

expansion

```
MOVER   AREG, X
ADD     AREG, Y
MOVEM   AREG, X
```

(5) Remaining statements

```
PRINT X
STOP
X  DS  1
Y  DS  1
END
```

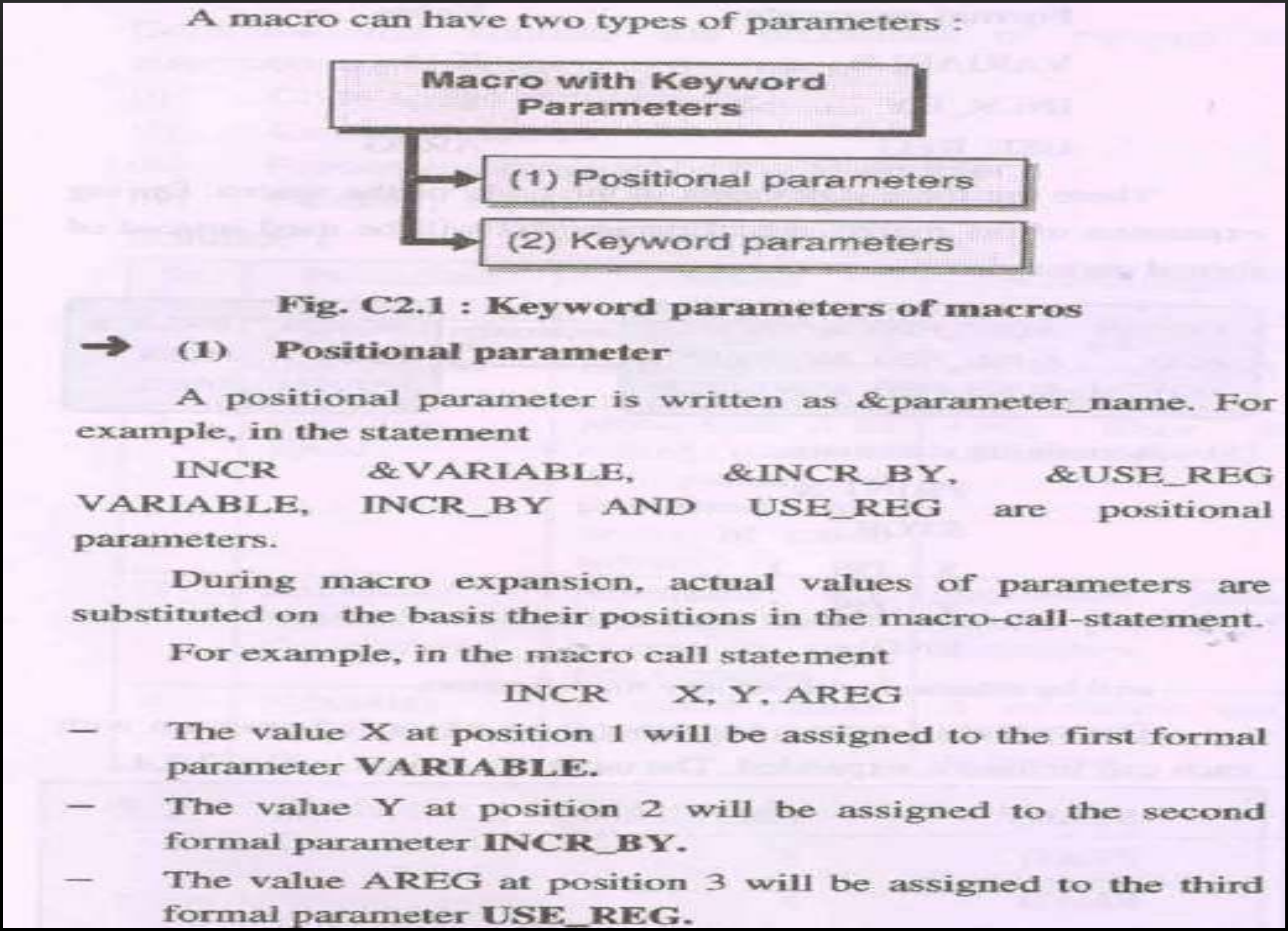
will be retained without any modification.

The output of macro processor is an expanded program with each call to macro, expanded. The output is shown in Fig. 2.2.4.

```
START       100
READ        X
READ        Y
MOVER       AREG, X   ┐
ADD         AREG, Y   │  INCR   X, Y, AREG
MOVEM       AREG, X   ┘  is expanded here.

PRINT       X
STOP
X     DS    1
Y     DS    1
END
```

Fig. 2.2.4 : Expanded code

MACRO WITH KEYWORD



A macro can have two types of parameters :

Macro with Keyword Parameters

(1) Positional parameters

(2) Keyword parameters

Fig. C2.1 : Keyword parameters of macros

→ **(1) Positional parameter**

A positional parameter is written as &parameter_name. For example, in the statement

INCR &VARIABLE, &INCR_BY, &USE_REG VARIABLE, INCR_BY AND USE_REG are positional parameters.

During macro expansion, actual values of parameters are substituted on the basis their positions in the macro-call-statement.

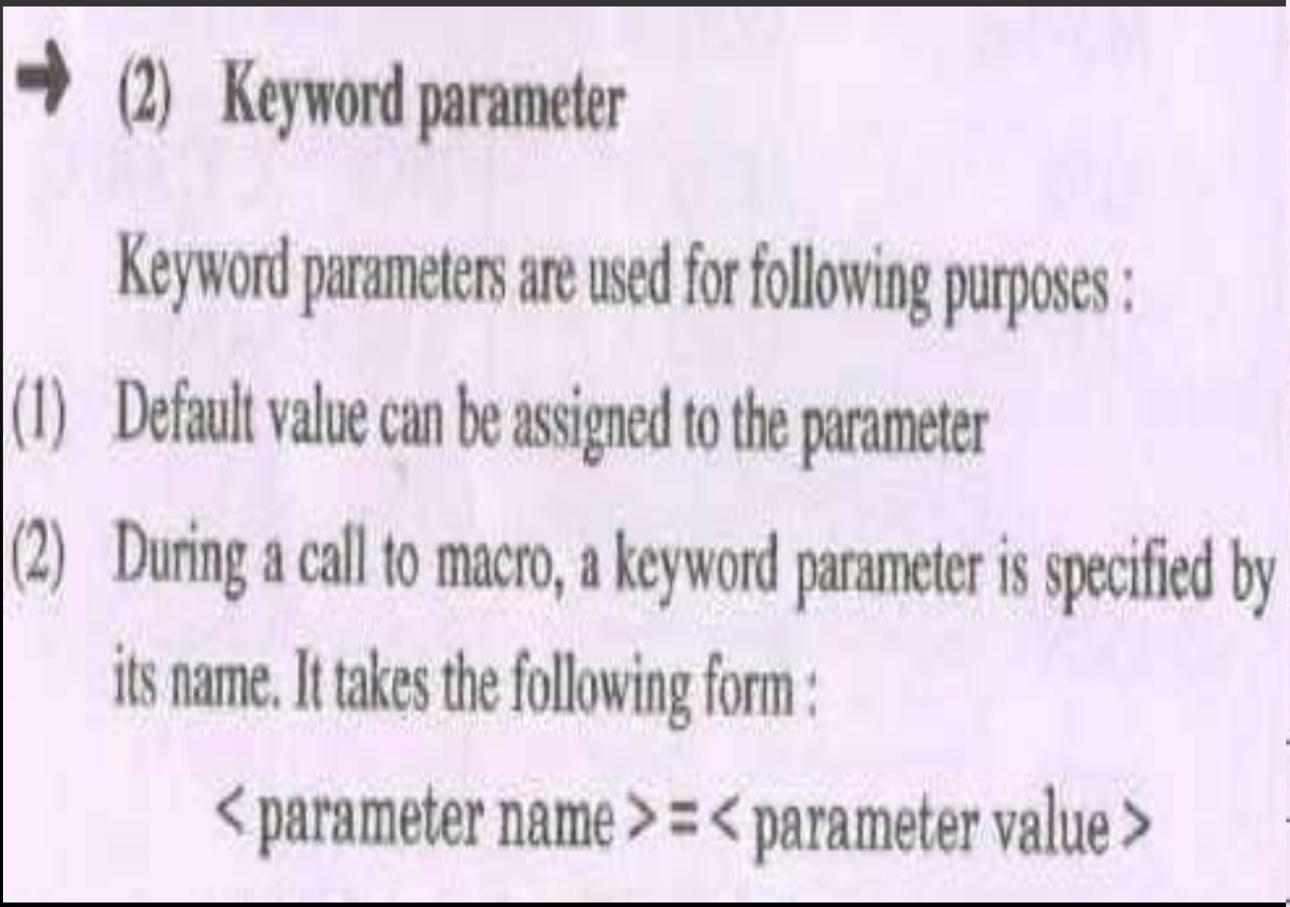
For example, in the macro call statement

INCR X, Y, AREG

— The value X at position 1 will be assigned to the first formal parameter **VARIABLE.**

— The value Y at position 2 will be assigned to the second formal parameter **INCR_BY.**

— The value AREG at position 3 will be assigned to the third formal parameter **USE_REG.**



➜ (2) Keyword parameter

Keyword parameters are used for following purposes :

(1) Default value can be assigned to the parameter

(2) During a call to macro, a keyword parameter is specified by its name. It takes the following form :

< parameter name > = < parameter value >

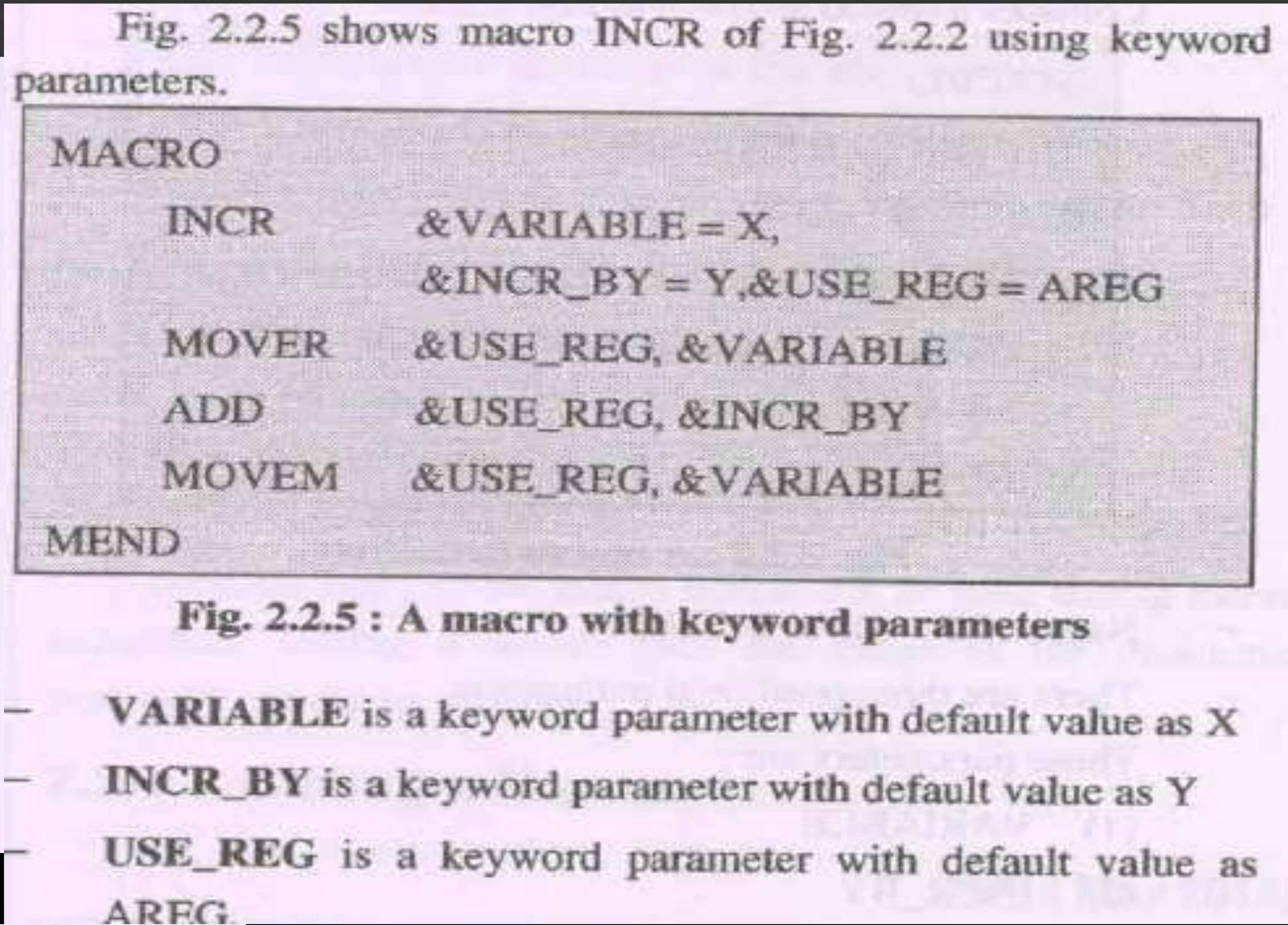
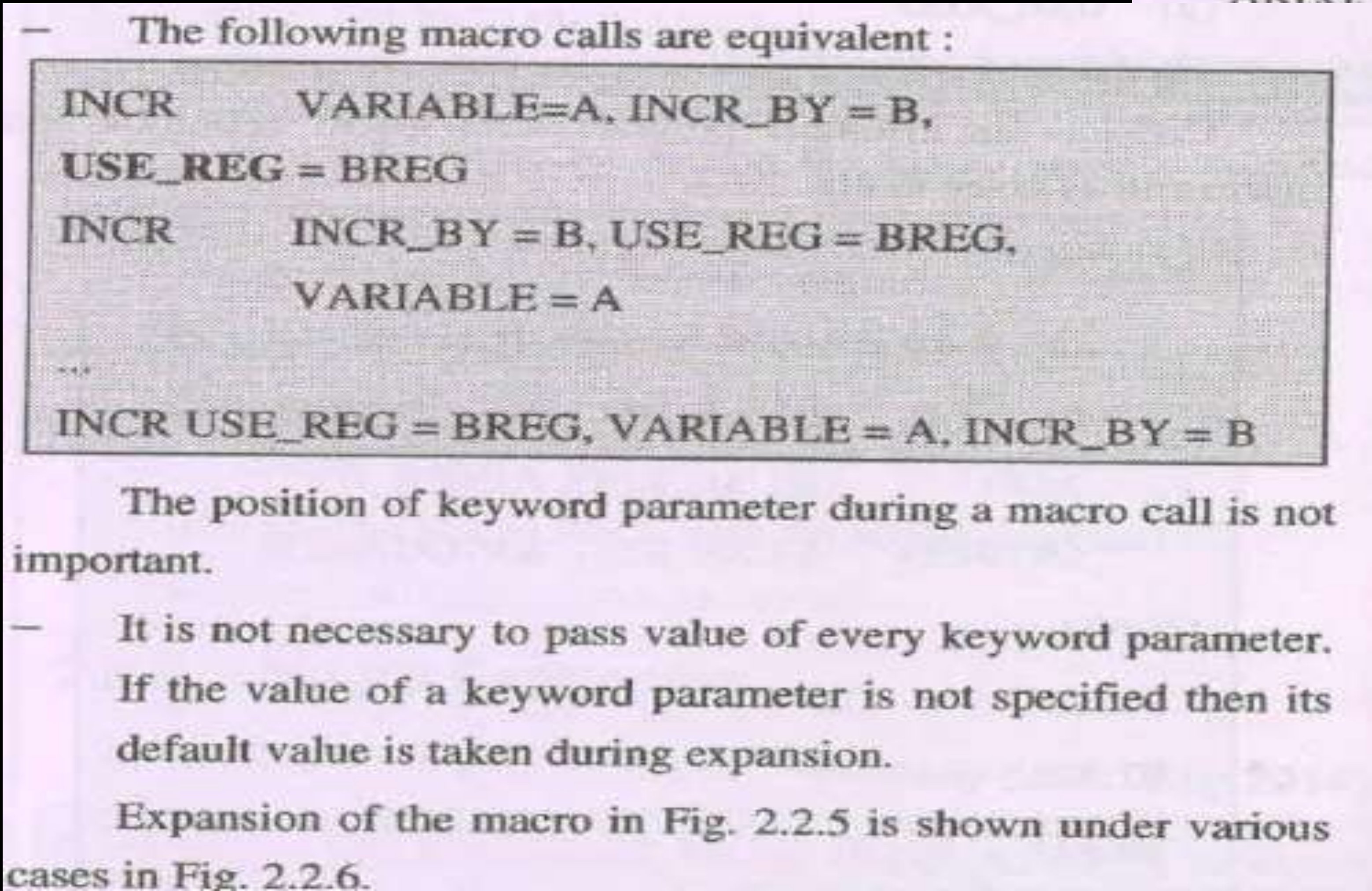


Fig. 2.2.5 shows macro INCR of Fig. 2.2.2 using keyword parameters.

```
MACRO
     INCR      &VARIABLE = X,
               &INCR_BY = Y,&USE_REG = AREG
     MOVER     &USE_REG, &VARIABLE
     ADD       &USE_REG, &INCR_BY
     MOVEM     &USE_REG, &VARIABLE
MEND
```

Fig. 2.2.5 : A macro with keyword parameters

- **VARIABLE** is a keyword parameter with default value as X
- **INCR_BY** is a keyword parameter with default value as Y
- **USE_REG** is a keyword parameter with default value as AREG.



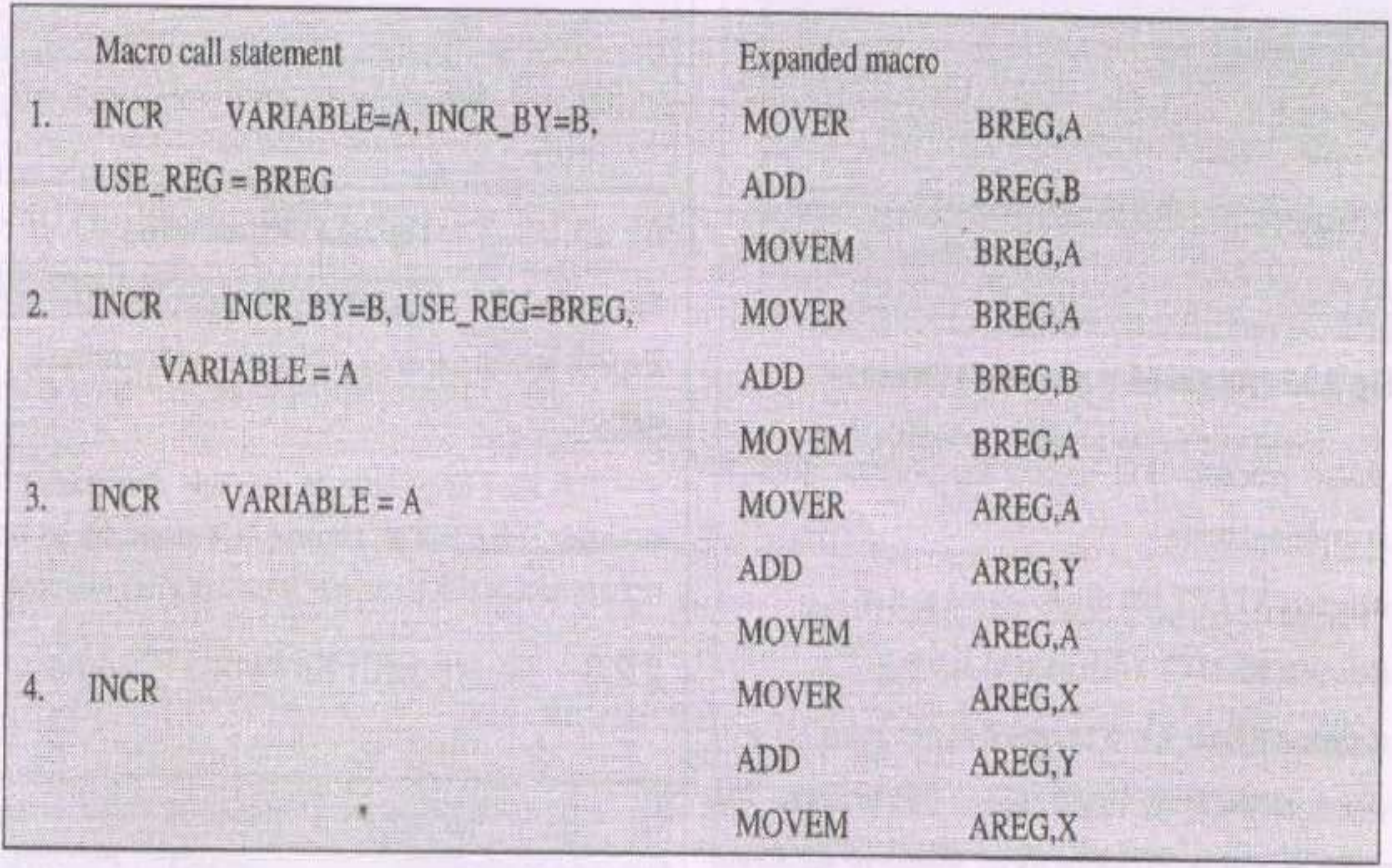
- The following macro calls are equivalent :

```
INCR      VARIABLE=A, INCR_BY = B,
USE_REG = BREG
INCR      INCR_BY = B, USE_REG = BREG,
          VARIABLE = A
...
INCR USE_REG = BREG, VARIABLE = A, INCR_BY = B
```

The position of keyword parameter during a macro call is not important.

- It is not necessary to pass value of every keyword parameter. If the value of a keyword parameter is not specified then its default value is taken during expansion.

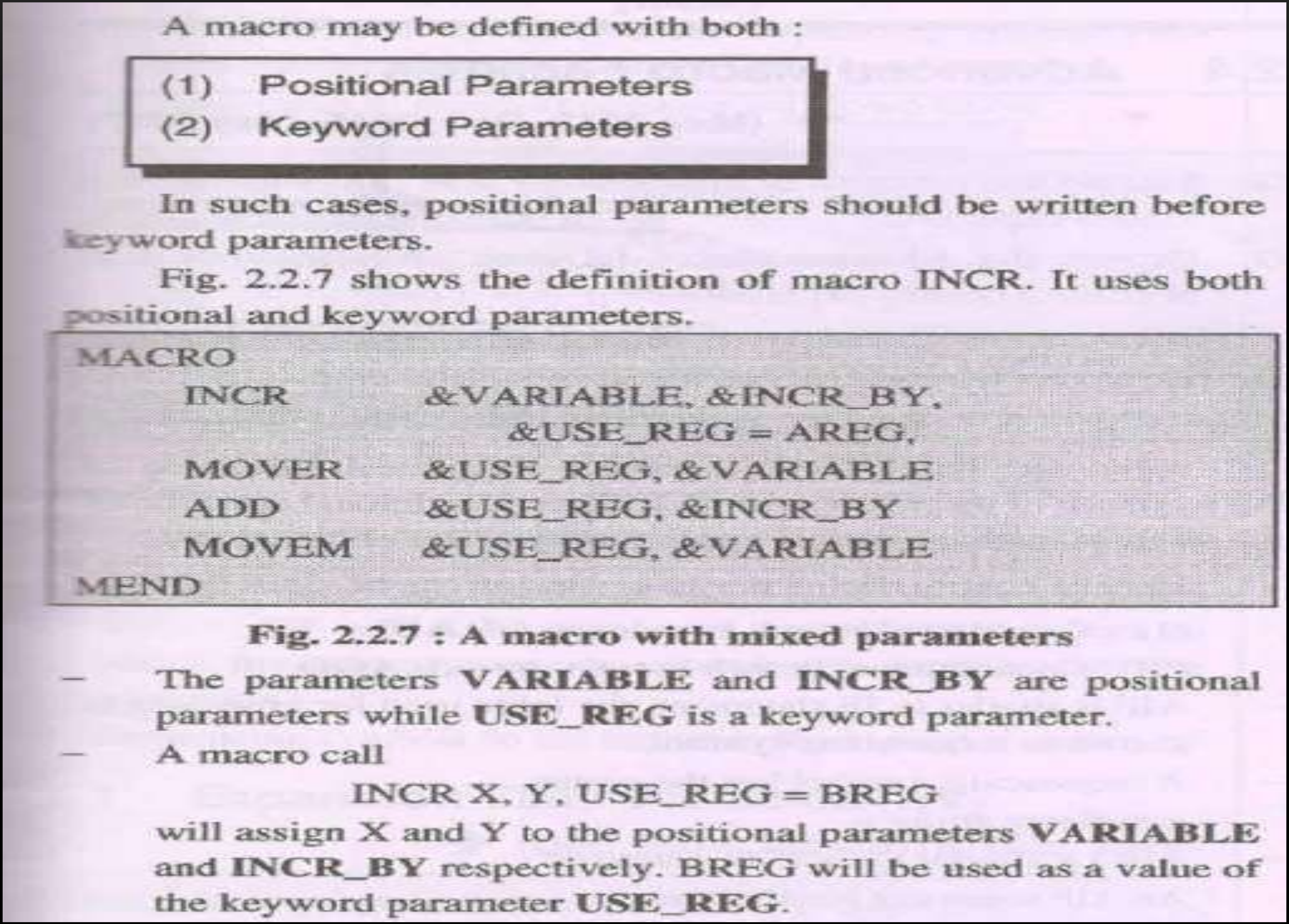
Expansion of the macro in Fig. 2.2.5 is shown under various cases in Fig. 2.2.6.



	Macro call statement	Expanded macro	
1.	INCR VARIABLE=A, INCR_BY=B, USE_REG = BREG	MOVER	BREG,A
		ADD	BREG,B
		MOVEM	BREG,A
2.	INCR INCR_BY=B, USE_REG=BREG, VARIABLE = A	MOVER	BREG,A
		ADD	BREG,B
		MOVEM	BREG,A
3.	INCR VARIABLE = A	MOVER	AREG,A
		ADD	AREG,Y
		MOVEM	AREG,A
4.	INCR	MOVER	AREG,X
		ADD	AREG,Y
		MOVEM	AREG,X

Fig. 2.2.6: Various cases of expansion

MACRO WITH MIXED PARAMETER



A macro may be defined with both :

(1) Positional Parameters
(2) Keyword Parameters

In such cases, positional parameters should be written before keyword parameters.

Fig. 2.2.7 shows the definition of macro INCR. It uses both positional and keyword parameters.

```
MACRO
     INCR       &VARIABLE, &INCR_BY,
                     &USE_REG = AREG,
     MOVER      &USE_REG, &VARIABLE
     ADD        &USE_REG, &INCR_BY
     MOVEM      &USE_REG, &VARIABLE
MEND
```

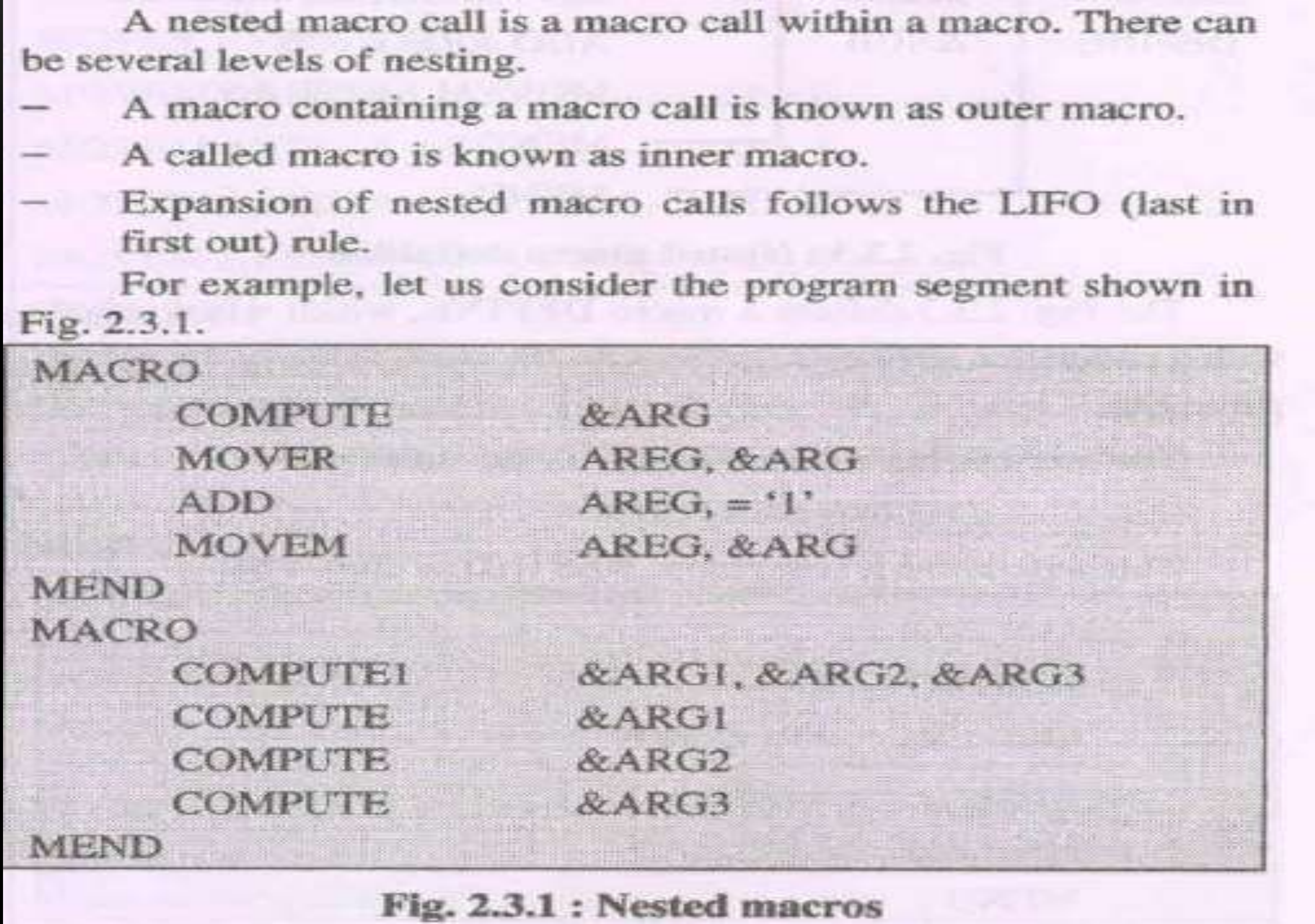
Fig. 2.2.7 : A macro with mixed parameters

- The parameters **VARIABLE** and **INCR_BY** are positional parameters while **USE_REG** is a keyword parameter.
- A macro call

  INCR X, Y, USE_REG = BREG

  will assign X and Y to the positional parameters **VARIABLE** and **INCR_BY** respectively. BREG will be used as a value of the keyword parameter **USE_REG**.

NESTED MACRO CALL



A nested macro call is a macro call within a macro. There can be several levels of nesting.

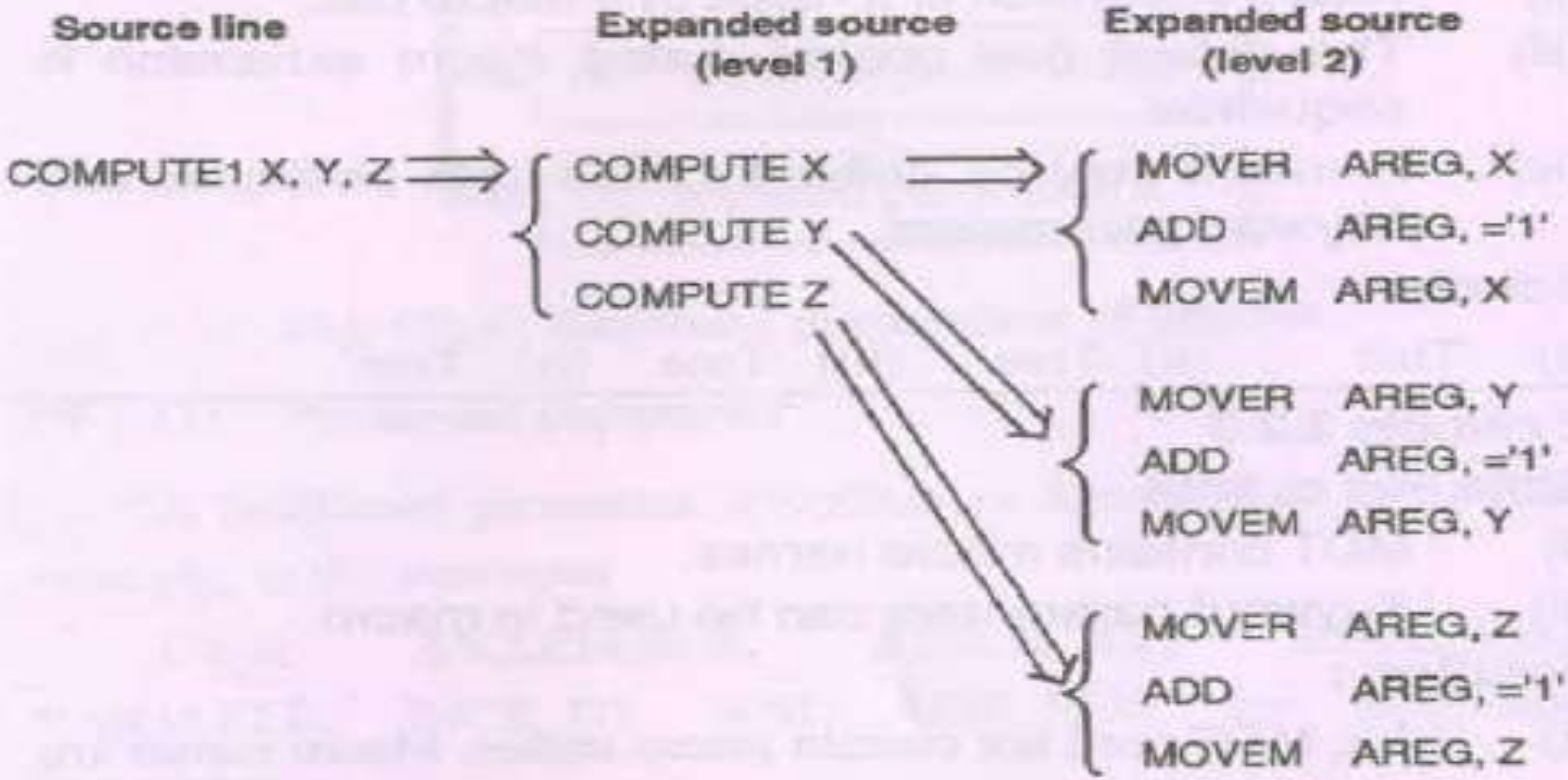
- A macro containing a macro call is known as outer macro.
- A called macro is known as inner macro.
- Expansion of nested macro calls follows the LIFO (last in first out) rule.

For example, let us consider the program segment shown in Fig. 2.3.1.

```
MACRO
        COMPUTE         &ARG
        MOVER           AREG, &ARG
        ADD             AREG, = '1'
        MOVEM           AREG, &ARG
MEND
MACRO
        COMPUTE1        &ARG1, &ARG2, &ARG3
        COMPUTE         &ARG1
        COMPUTE         &ARG2
        COMPUTE         &ARG3
MEND
```

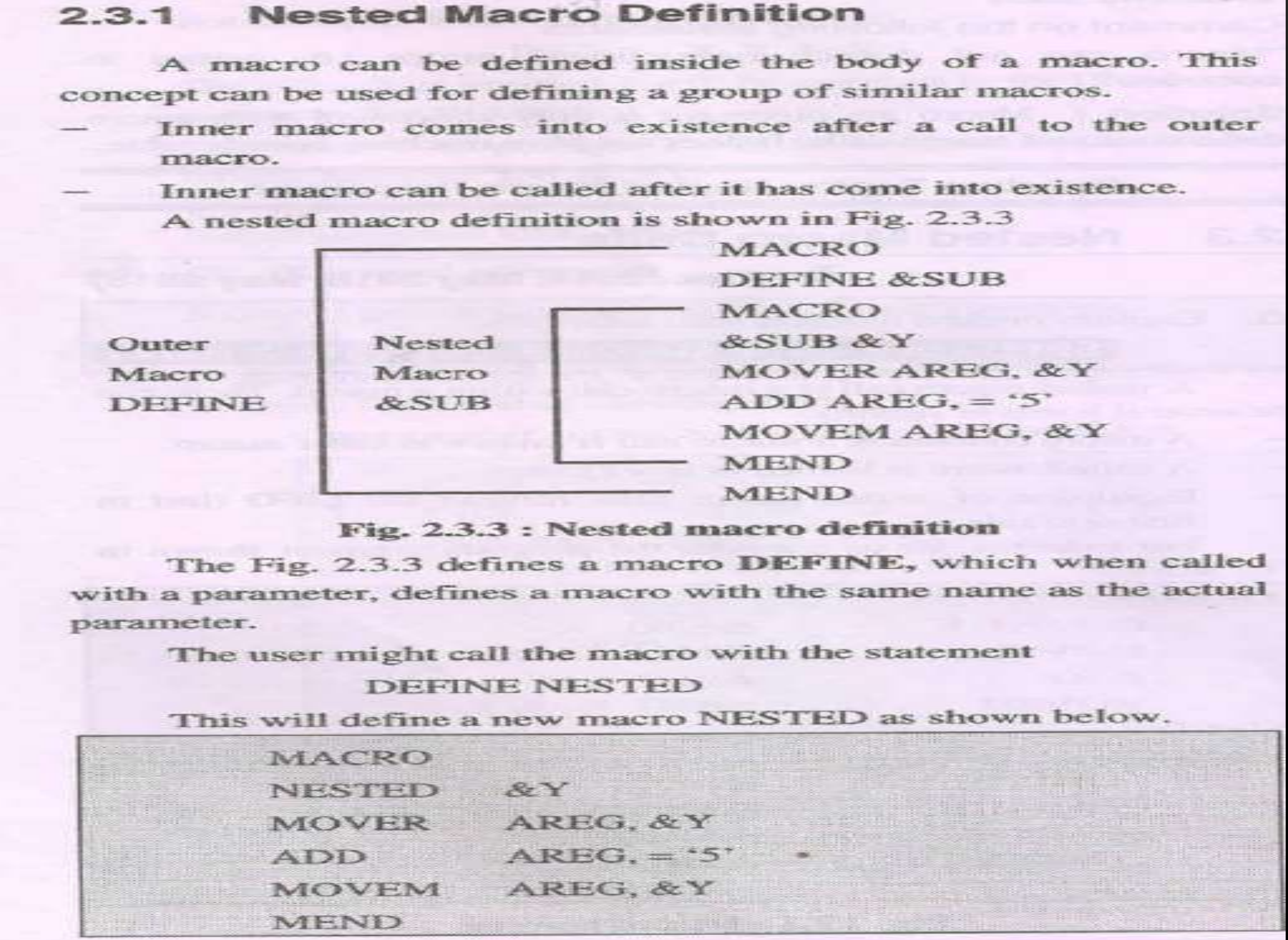
Fig. 2.3.1 : Nested macros

The definition of macro **COMPUTE1** contains three separate calls to a previously defined macro **COMPUTE**. Such macros are expanded on multiple levels. Expansion of COMPUTE1 X, Y, Z is shown in Fig. 2.3.2.



(S3.4)**Fig. 2.3.2 : Expansion of compute X, Y, Z**

NESTED MACRO DEFINITION



2.3.1 Nested Macro Definition

A macro can be defined inside the body of a macro. This concept can be used for defining a group of similar macros.

- Inner macro comes into existence after a call to the outer macro.
- Inner macro can be called after it has come into existence.

A nested macro definition is shown in Fig. 2.3.3

```
                                          MACRO
                                          DEFINE &SUB
                                          MACRO
Outer          Nested                     &SUB &Y
Macro          Macro                      MOVER AREG, &Y
DEFINE         &SUB                       ADD AREG, = '5'
                                          MOVEM AREG, &Y
                                          MEND
                                          MEND
```

Fig. 2.3.3 : Nested macro definition

The Fig. 2.3.3 defines a macro **DEFINE**, which when called with a parameter, defines a macro with the same name as the actual parameter.

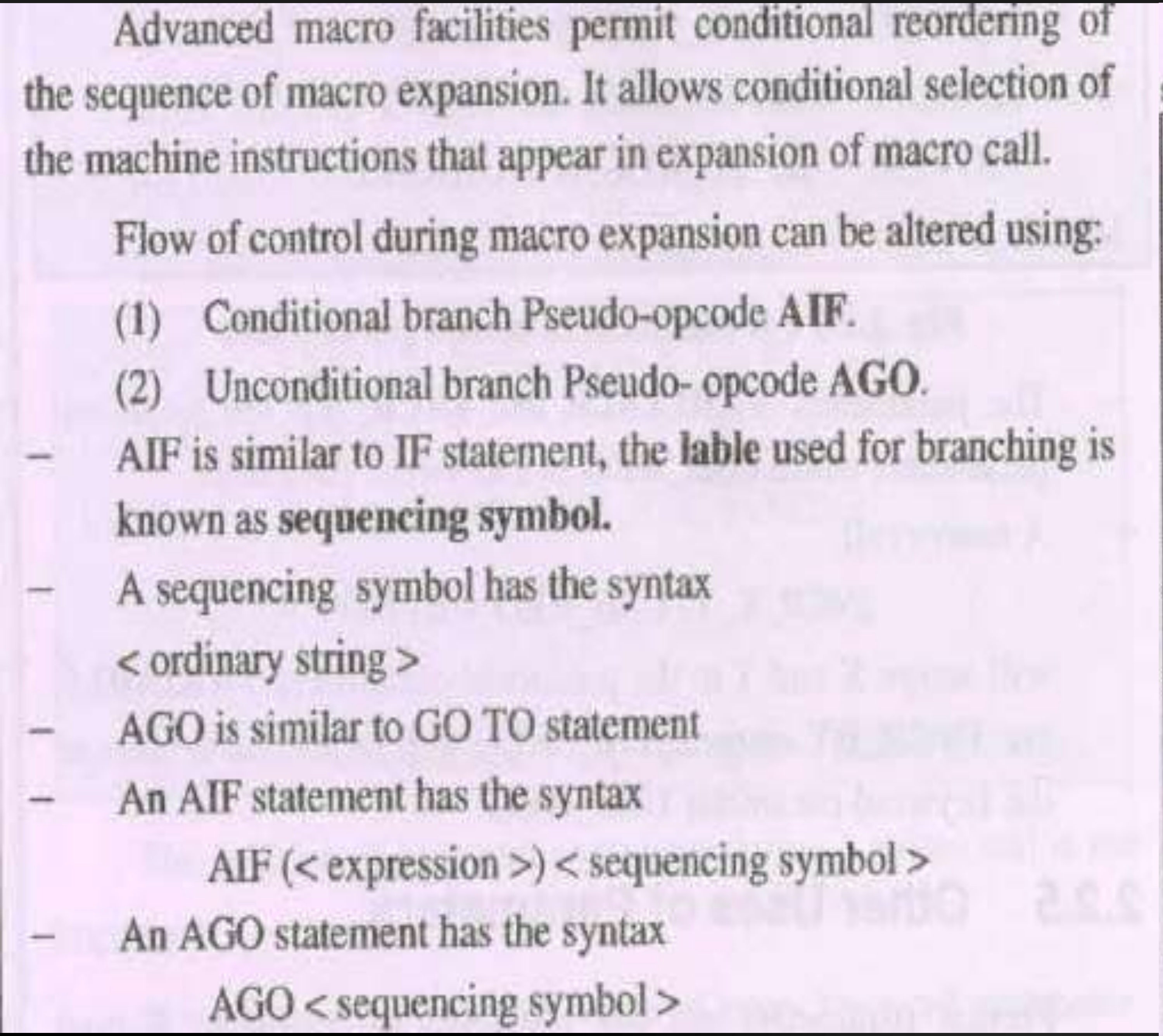
The user might call the macro with the statement

DEFINE NESTED

This will define a new macro NESTED as shown below.

```
MACRO
NESTED     &Y
MOVER      AREG, &Y
ADD        AREG, = '5'
MOVEM      AREG, &Y
MEND
```

ADVANCED MACRO FACILITY



Advanced macro facilities permit conditional reordering of the sequence of macro expansion. It allows conditional selection of the machine instructions that appear in expansion of macro call.

Flow of control during macro expansion can be altered using:

(1) Conditional branch Pseudo-opcode **AIF**.

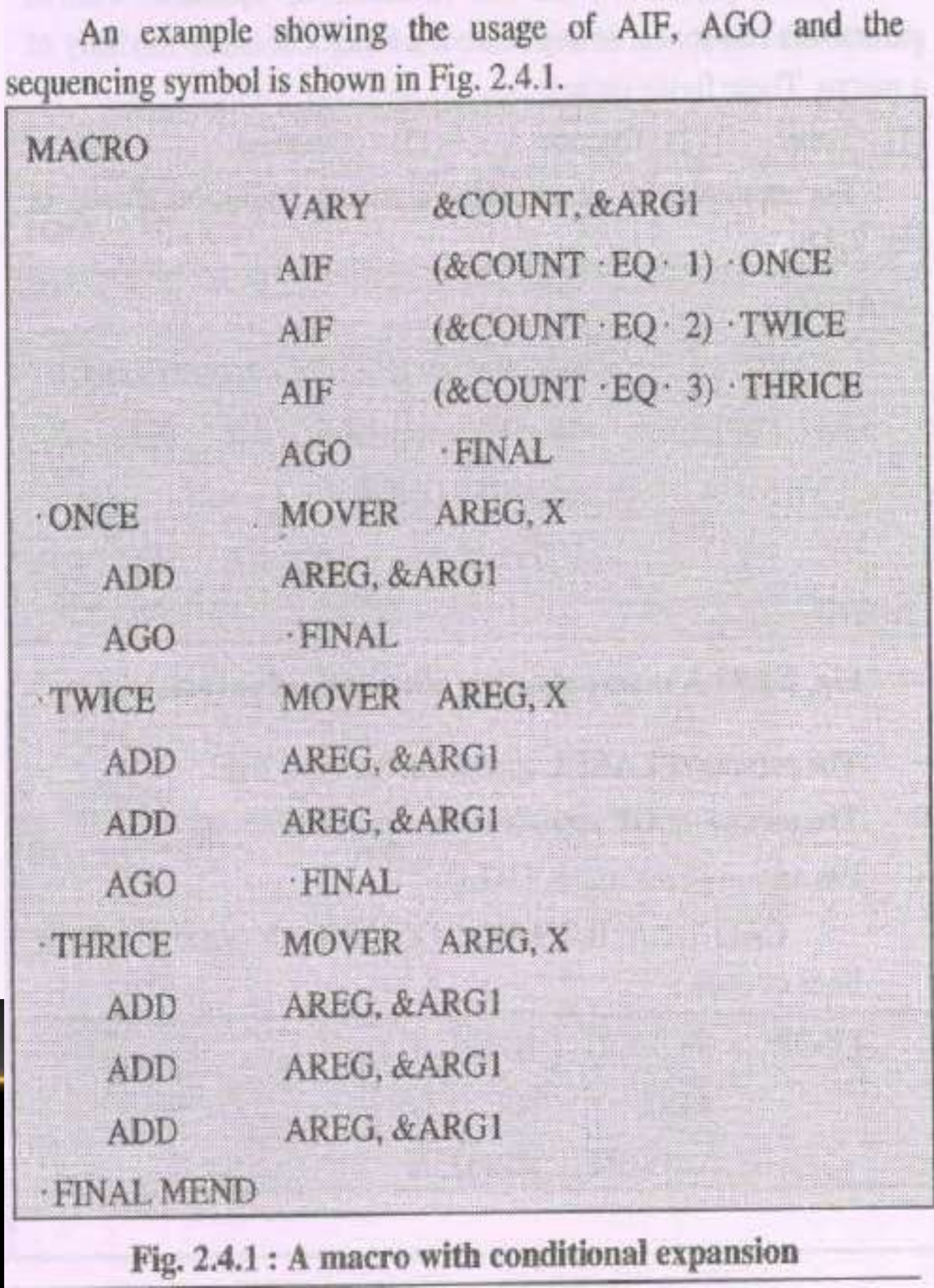
(2) Unconditional branch Pseudo- opcode **AGO**.

- AIF is similar to IF statement, the **lable** used for branching is known as **sequencing symbol.**
- A sequencing symbol has the syntax

  < ordinary string >
- AGO is similar to GO TO statement
- An AIF statement has the syntax

  AIF (< expression >) < sequencing symbol >
- An AGO statement has the syntax

  AGO < sequencing symbol >



An example showing the usage of AIF, AGO and the sequencing symbol is shown in Fig. 2.4.1.

```
MACRO
            VARY      &COUNT, &ARG1
            AIF       (&COUNT .EQ. 1) .ONCE
            AIF       (&COUNT .EQ. 2) .TWICE
            AIF       (&COUNT .EQ. 3) .THRICE
            AGO       .FINAL
.ONCE       MOVER     AREG, X
     ADD    AREG, &ARG1
     AGO    .FINAL
.TWICE      MOVER     AREG, X
     ADD    AREG, &ARG1
     ADD    AREG, &ARG1
     AGO    .FINAL
.THRICE     MOVER     AREG, X
     ADD    AREG, &ARG1
     ADD    AREG, &ARG1
     ADD    AREG, &ARG1
.FINAL MEND
```

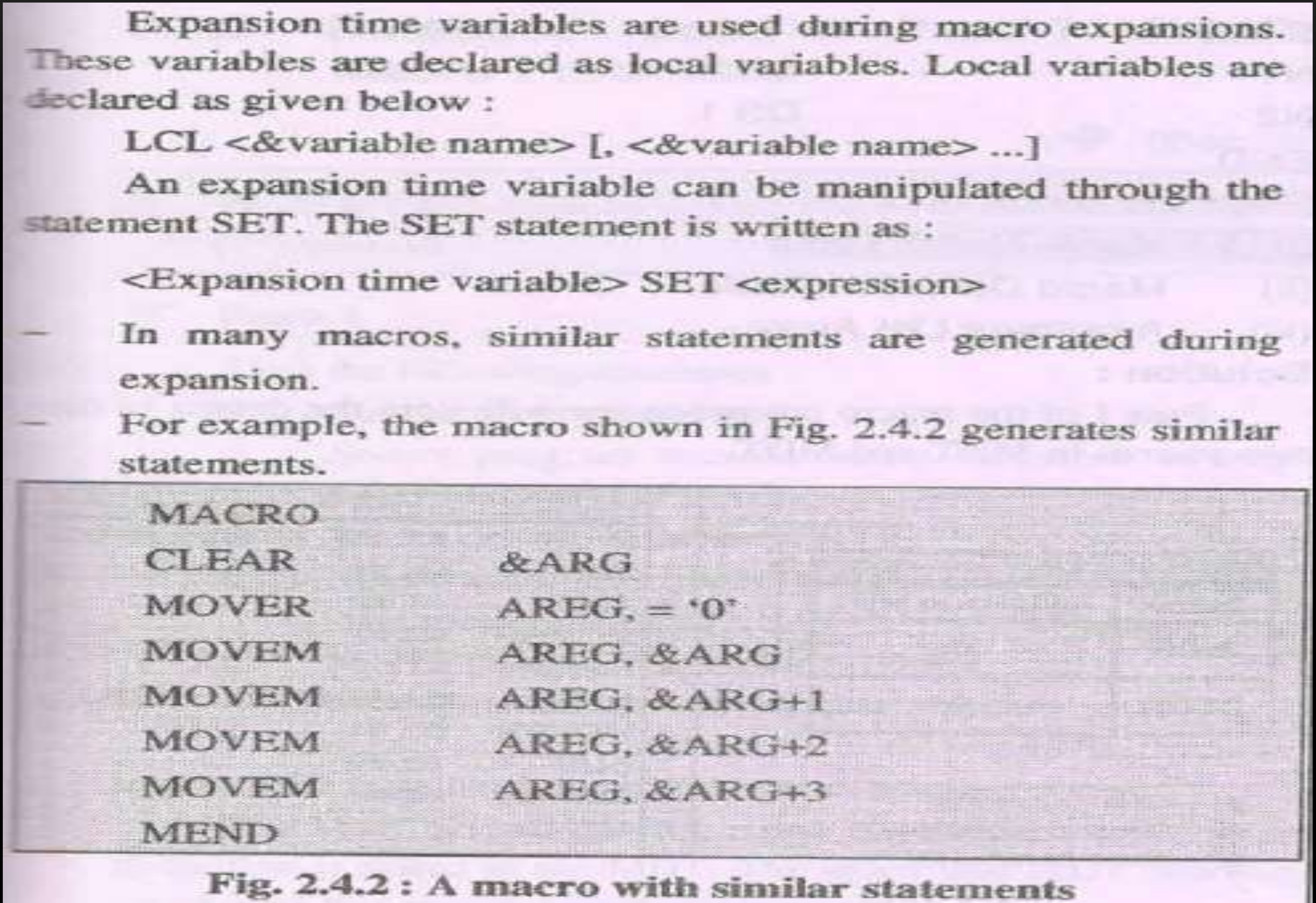
Fig. 2.4.1 : A macro with conditional expansion

- In this macro, the number of instructions generated during expansion will depend on the value of the parameter **&count.**
- ONCE, TWICE and THRICE are sequencing symbol. They help in transfer of control during expansion of the macro.
- Various cases of macro expansions are shown below :

	Macro call	Expanded source	
1.	VARY 1, Y	MOVER ADD	AREG, X AREG, Y
2.	VARY 2, Y	MOVER ADD ADD	AREG, X AREG, Y AREG, Y
3.	VARY 3, Y	MOVER ADD ADD ADD	AREG, X AREG, Y AREG, Y AREG, Y

- AIF and AGO statements do not appear in the expanded source. AIF and AGO statements control the sequence in which the macro processor expands the statements during expansion.
- Sequencing symbols do not appear in the expanded code.

EXPANSION TIME VARIABLE



Expansion time variables are used during macro expansions. These variables are declared as local variables. Local variables are declared as given below :

LCL <&variable name> [, <&variable name> ...]

An expansion time variable can be manipulated through the statement SET. The SET statement is written as :

<Expansion time variable> SET <expression>

- In many macros, similar statements are generated during expansion.
- For example, the macro shown in Fig. 2.4.2 generates similar statements.

```
MACRO
CLEAR           &ARG
MOVER           AREG, = '0'
MOVEM           AREG, &ARG
MOVEM           AREG, &ARG+1
MOVEM           AREG, &ARG+2
MOVEM           AREG, &ARG+3
MEND
```

Fig. 2.4.2 : A macro with similar statements

A call to macro CLEAR with the statement,

CLEAR A

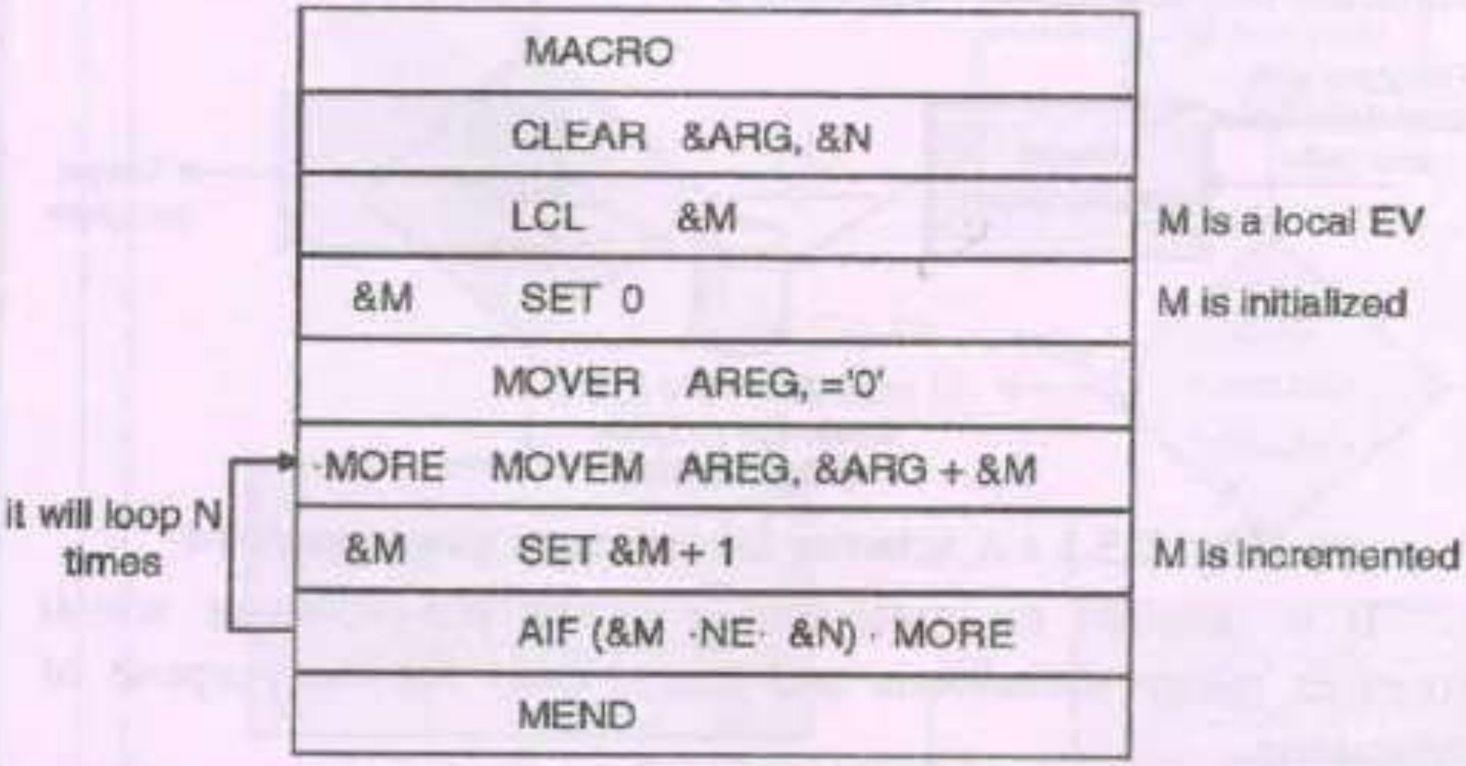
will lead to following expansion

```
MOVER AREG, = '0'
MOVEM AREG, A
MOVEM AREG, A + 1
MOVEM AREG, A + 2
MOVEM AREG, A + 3
```

The above code stores the value '0' in four consecutive locations with the address A, A + 1, A + 2 and A + 3.

- Alternatively, the same effect can be created by implementing loop for expansion. Loop can create the same effect as given in the macro CLEAR. Expansion time loop can be written using expansion time variable.

- The macro given in Fig. 2.4.2 can be re-written as shown in Fig. 2.4.3.



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

(S3.5)**Fig. 2.4.3 : An equivalent macro written using expansion time variable (EV)**

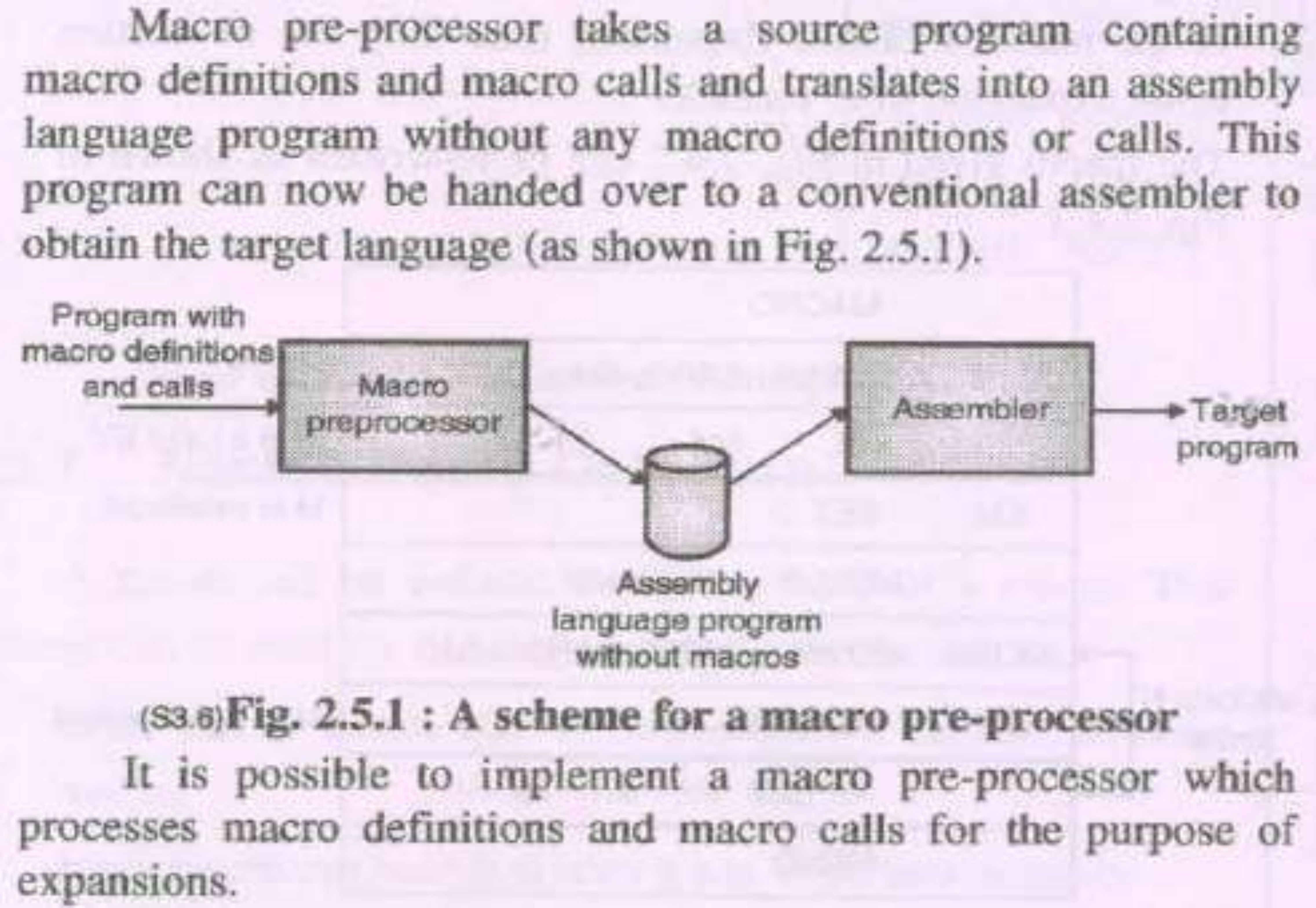
- A call to macro CLEAR (Fig. 2.4.3) with the statement.

CLEAR A, 3

will loop three times for each value of M from 0 to 3 with the following expansion.

```
MOVER AREG, = '0'
MOVEM AREG, A        ——   M = 0
MOVEM AREG, A + 1    ——   M = 1
MOVEM AREG, A + 2    ——   M = 2
MOVEM AREG, A + 3    ——   M = 3
```

DESIGN OF MACROPROCESSOR



Macro pre-processor takes a source program containing macro definitions and macro calls and translates into an assembly language program without any macro definitions or calls. This program can now be handed over to a conventional assembler to obtain the target language (as shown in Fig. 2.5.1).

(S3.6)**Fig. 2.5.1 : A scheme for a macro pre-processor**

It is possible to implement a macro pre-processor which processes macro definitions and macro calls for the purpose of expansions.

2.5.1 Issues Related to the Design of a Simple Macro Preprocessor

We will go for a simple two pass macro pre-processor and then enhance it to handle advance features like :

1. AIF
2. AGO
3. Sequencing symbol
4. Expansion time variable

The macro pre-processor has to perform the four basic tasks :

1. Recognize macro definition
2. Save the macro definition
3. Recognize macro calls
4. Expand macro calls and substitute arguments.

- A macro definition is identified by MACRO and MEND pseudo opcodes.
- A macro definition is saved as it is required during macro expansion.
- A macro call appears as operation mnemonic.
- During macro expansion, the pre-processor must substitute formal parameters with actual parameters.

A macro pre-processor has to do the following :

☞ Pass 1

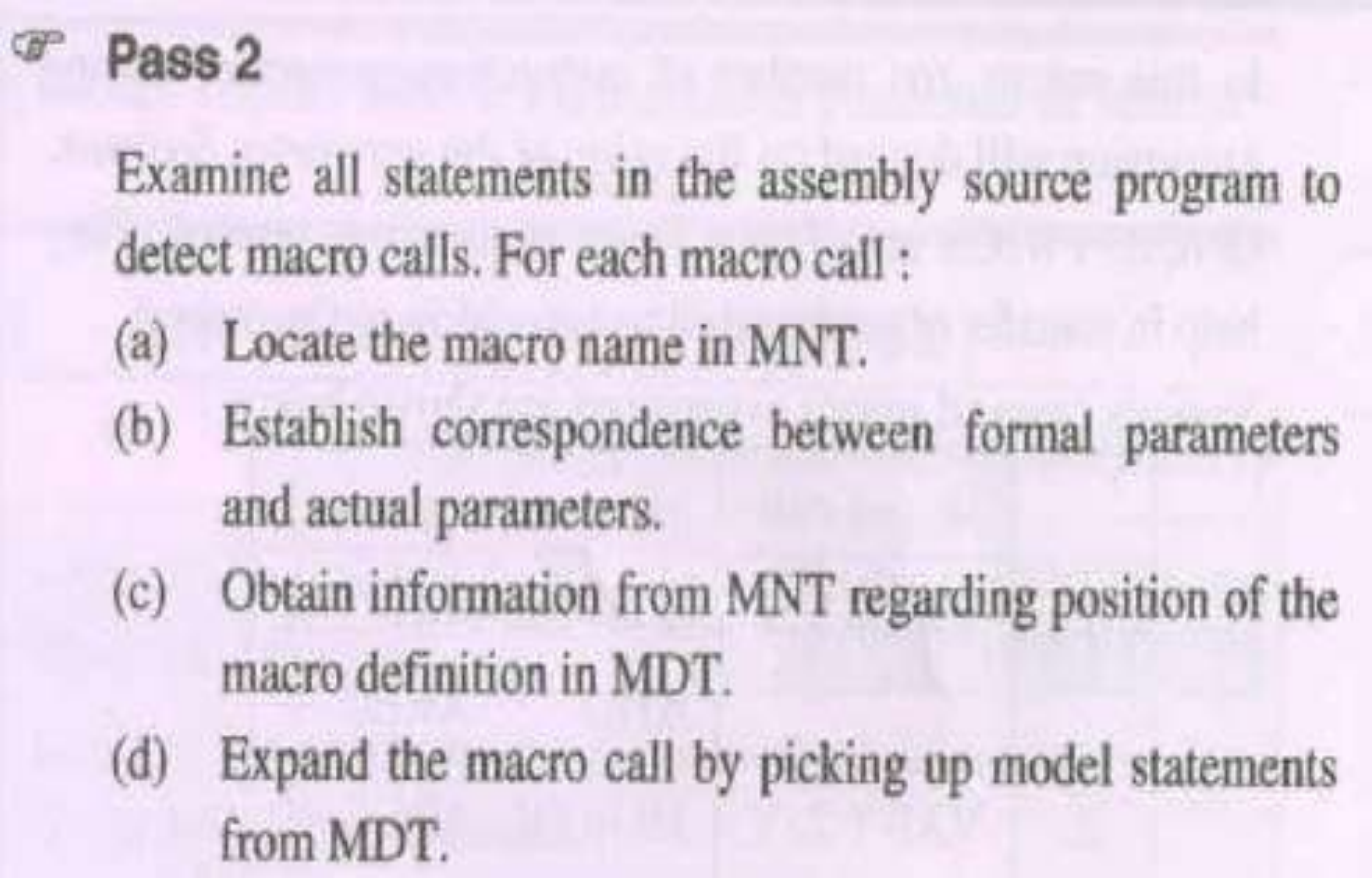
Scan all macro definitions one by one. For each macro:

(a) Enter its name in the Macro Name Table (MNT).

(b) Store the entire macro definition in the Macro Definition Table (MDT).

(c) Add the information to the MNT indicating where the definition of a macro can be found in MDT.

(d) Prepare argument list array.



Pass 2

Examine all statements in the assembly source program to detect macro calls. For each macro call :

(a) Locate the macro name in MNT.

(b) Establish correspondence between formal parameters and actual parameters.

(c) Obtain information from MNT regarding position of the macro definition in MDT.

(d) Expand the macro call by picking up model statements from MDT.

Example 2.5.1

Consider the following code segment.

```
MACRO
INCR                  &X, &Y, &REG = AREG
MOVER                 &REG, &X
ADD                   &REG, &Y
MOVEM                 &REG, &X
MEND
MACRO
DECR                  &A, &B, &REG = BREG
MOVER                 &REG, &A
SUB                   &REG, &B
MOVEM                 &REG, &A
MEND
START                 100
READ N1
READ N2
INCR  N1, N2, REG = CREG
DECR N1, N2
STOP
N1                    DS 1
N2                    DS 1
END
```

Show the contents of

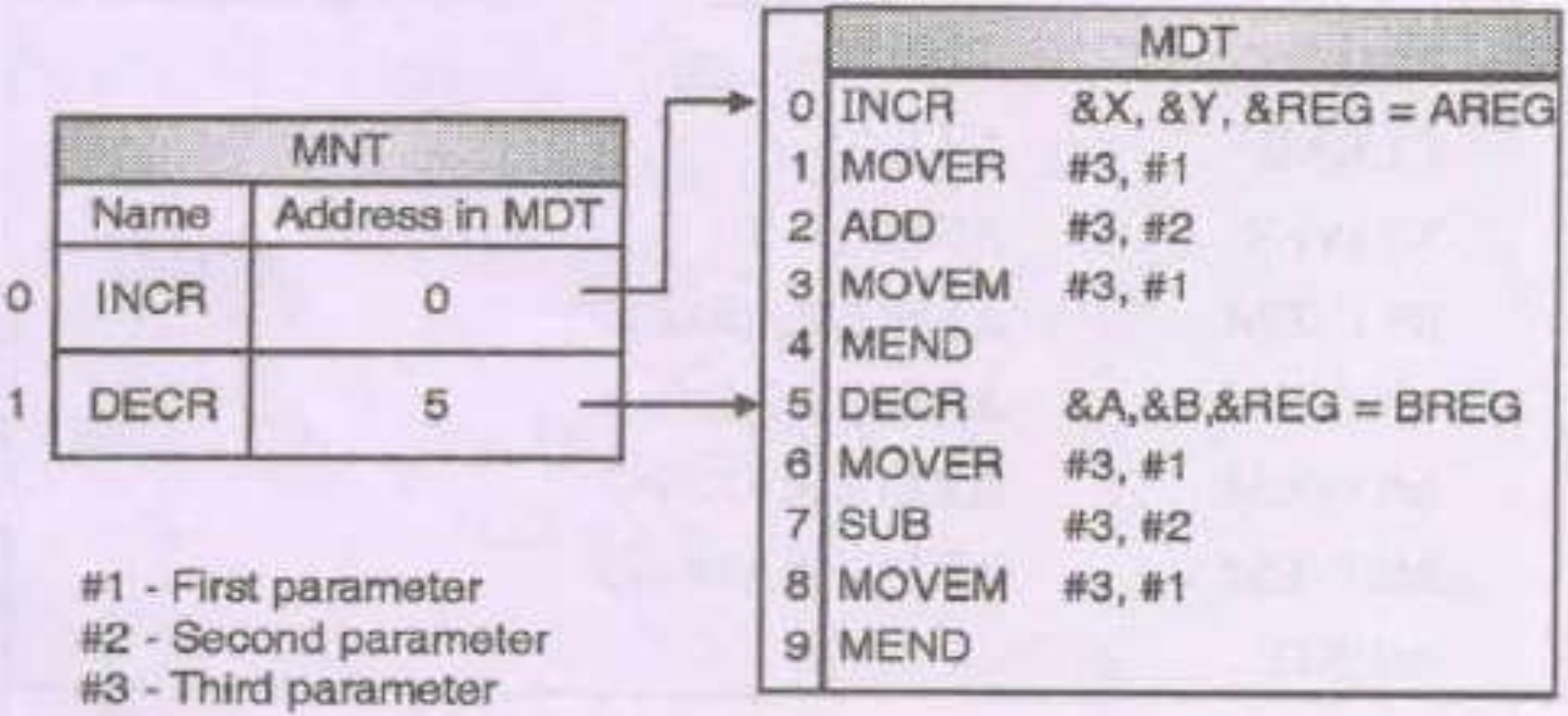
(i) Macro Name Table

(ii) Macro Definition Table

(iii) Argument List Array

Solution :

Pass I of the macro pre-processor will store the details of the two macros in MNT and MDT.



	MNT	
	Name	Address in MDT
0	INCR	0
1	DECR	5

	MDT	
0	INCR	&X, &Y, &REG = AREG
1	MOVER	#3, #1
2	ADD	#3, #2
3	MOVEM	#3, #1
4	MEND	
5	DECR	&A,&B,&REG = BREG
6	MOVER	#3, #1
7	SUB	#3, #2
8	MOVEM	#3, #1
9	MEND	

#1 - First parameter
#2 - Second parameter
#3 - Third parameter

(S3.7)**Fig. Ex. 2.5.1**

Pass II of the macro pre-processor will create the argument list array, every time there is a call to macro, and expand the macro.

(1) Macro call

```
INCR      N1, N2, REG = CREG
```

Argument list array

N1
N2
CREG

Expanded code :

```
MOVER     CREG, N1
ADD       CREG, N2
MOVEM     CREG, N1
```

(2) Macro call

```
DECR    N1, N2
```

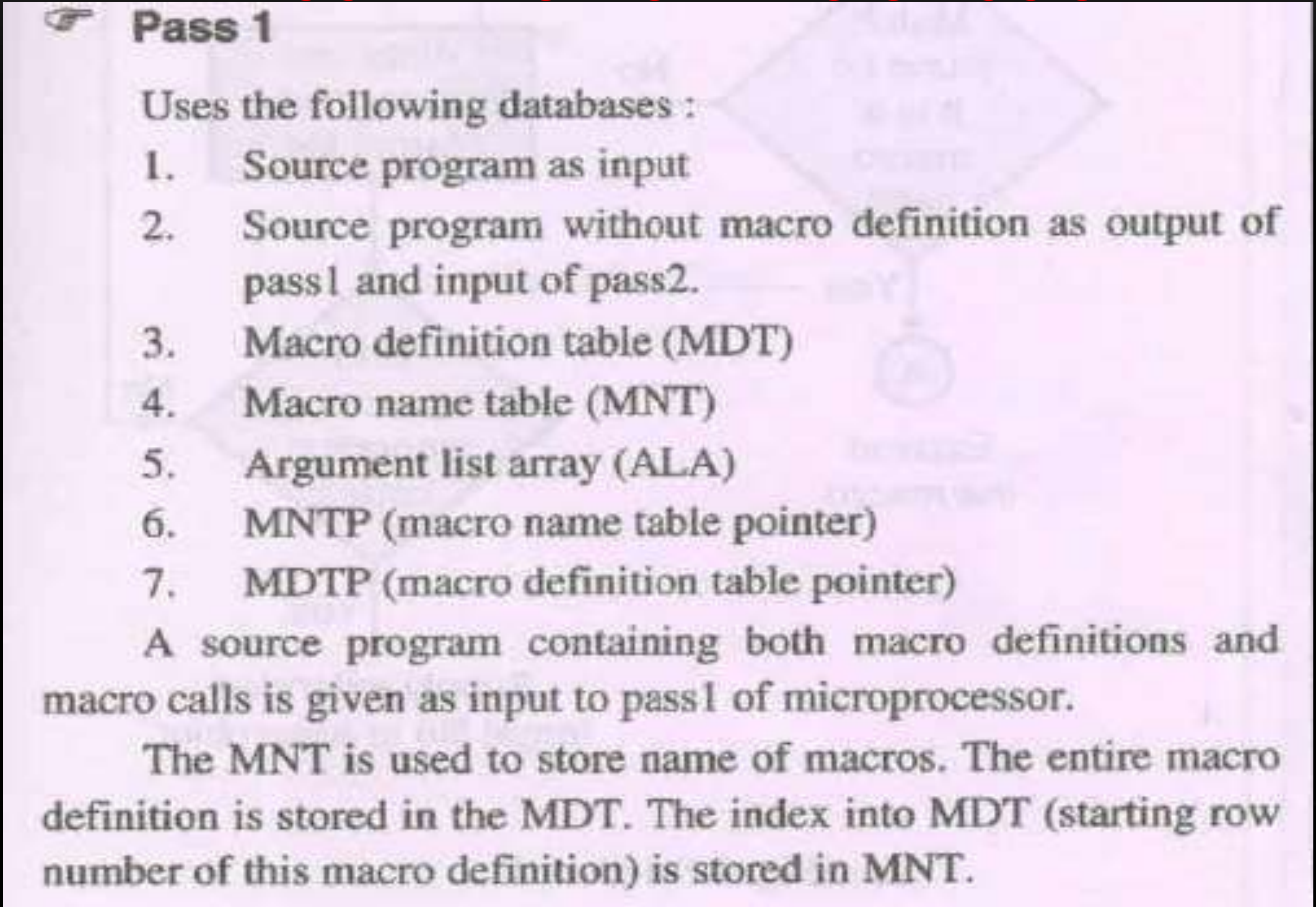
Argument list array

N1	
N2	
AREG	–Default value

Expanded code :

```
MOVER     AREG, N1
SUB       AREG, N2
MOVEM     AREG, N1
```

DATABASES USED IN PASS – I OF TWO PASS MACRO PROCESSOR



☞ Pass 1

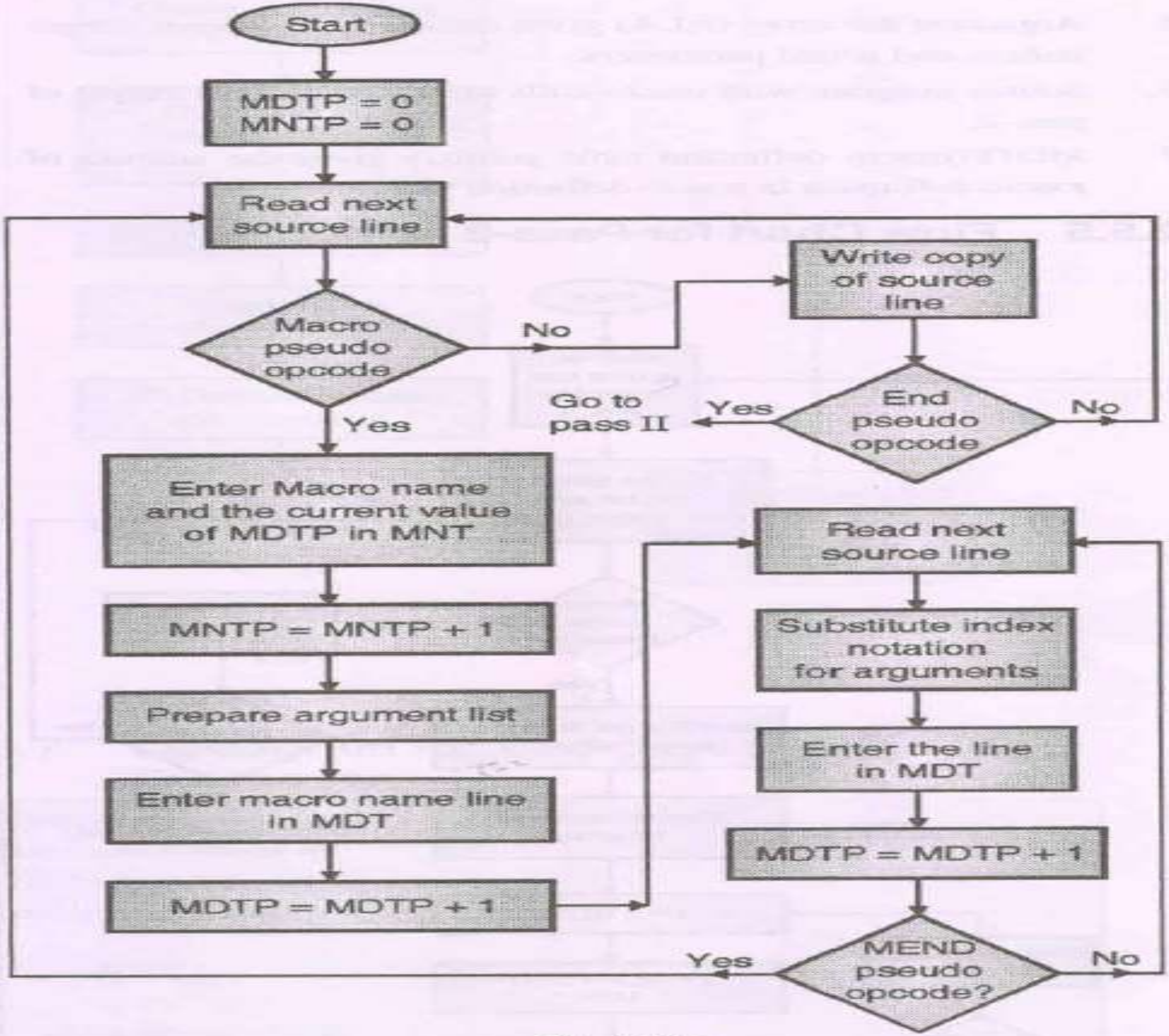
Uses the following databases :

1. Source program as input
2. Source program without macro definition as output of pass1 and input of pass2.
3. Macro definition table (MDT)
4. Macro name table (MNT)
5. Argument list array (ALA)
6. MNTP (macro name table pointer)
7. MDTP (macro definition table pointer)

A source program containing both macro definitions and macro calls is given as input to pass1 of microprocessor.

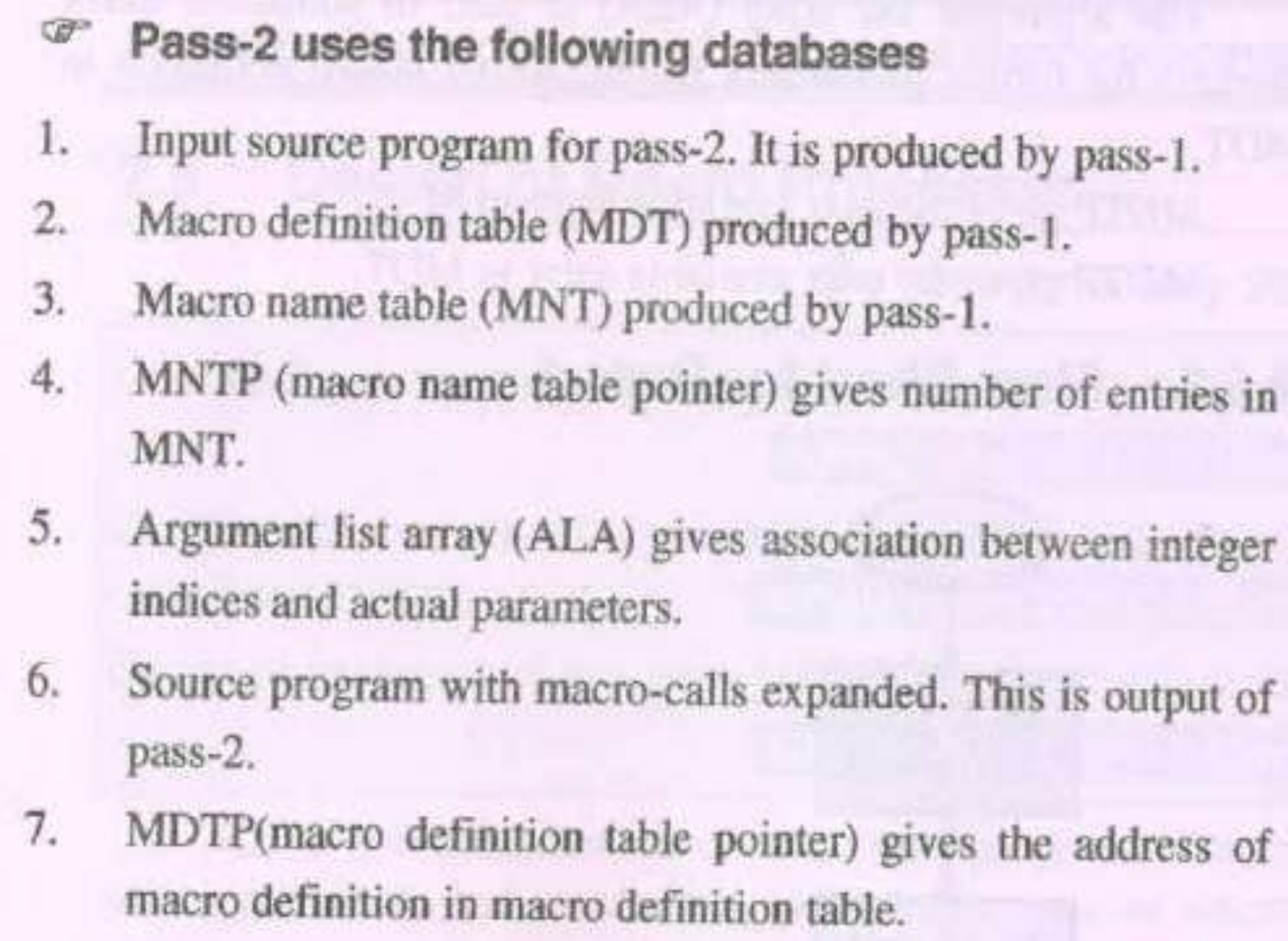
The MNT is used to store name of macros. The entire macro definition is stored in the MDT. The index into MDT (starting row number of this macro definition) is stored in MNT.

2.5.3 Flow Chart for Pass 1



(S3.8)**Fig. 2.5.2**

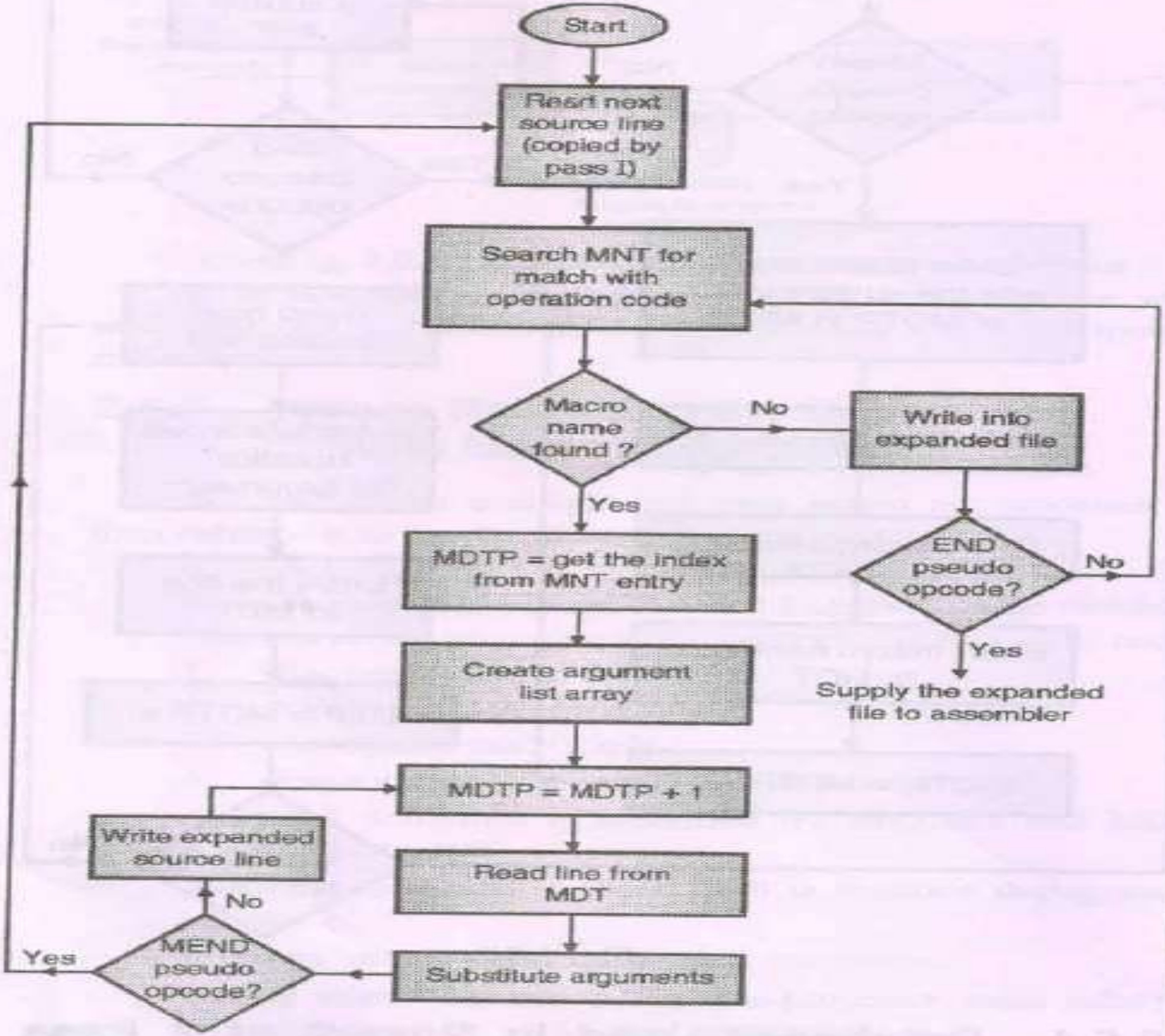
DATABASES USED IN PASS – II OF TWO PASS MACRO PROCESSOR



☞ **Pass-2 uses the following databases**

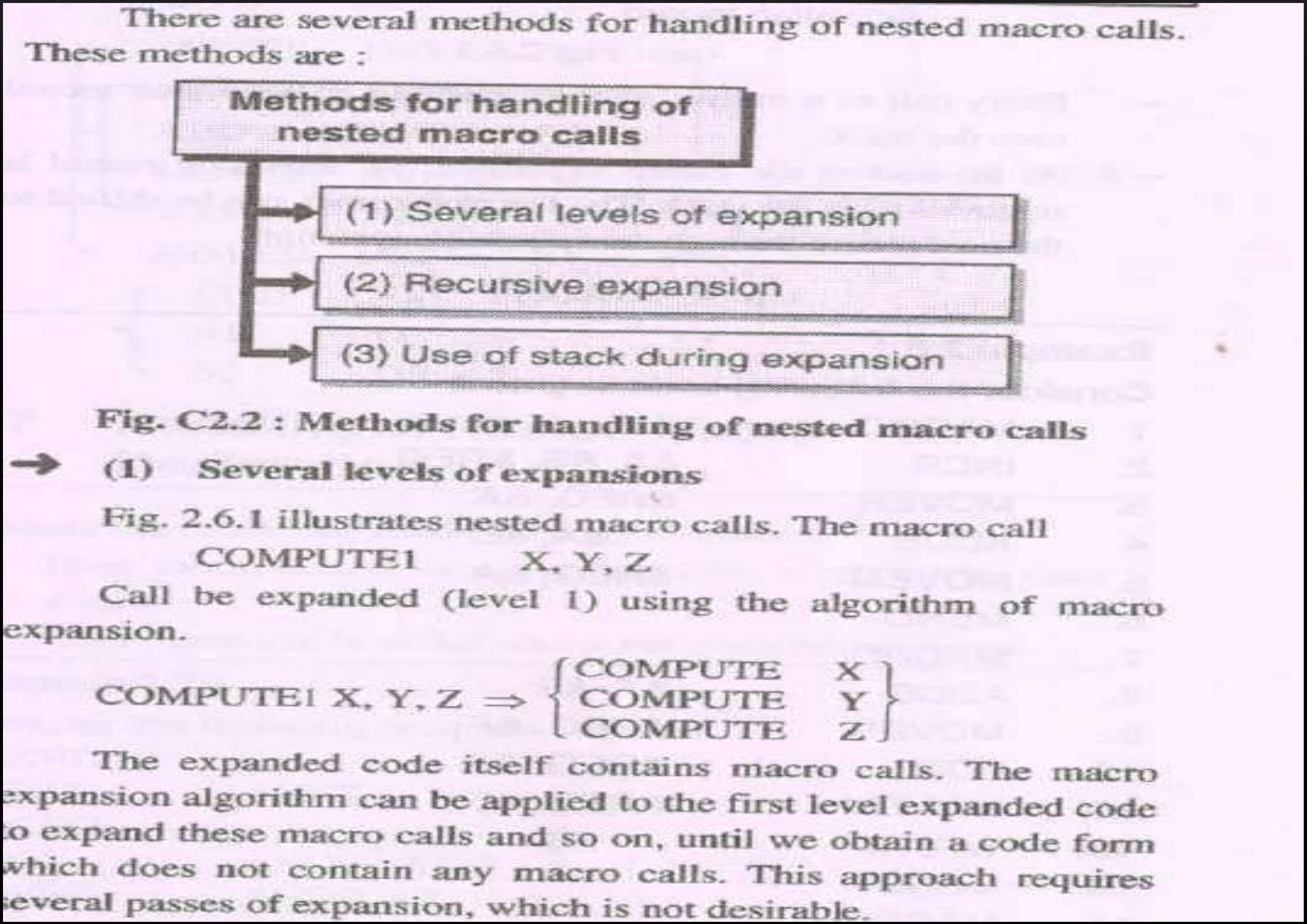
1. Input source program for pass-2. It is produced by pass-1.
2. Macro definition table (MDT) produced by pass-1.
3. Macro name table (MNT) produced by pass-1.
4. MNTP (macro name table pointer) gives number of entries in MNT.
5. Argument list array (ALA) gives association between integer indices and actual parameters.
6. Source program with macro-calls expanded. This is output of pass-2.
7. MDTP(macro definition table pointer) gives the address of macro definition in macro definition table.

2.5.5 Flow Chart for Pass-2



(S3.9)**Fig. 2.5.3**

HANDLING OF NESTED MACRO CALL



There are several methods for handling of nested macro calls. These methods are :

Methods for handling of nested macro calls

- (1) Several levels of expansion
- (2) Recursive expansion
- (3) Use of stack during expansion

Fig. C2.2 : Methods for handling of nested macro calls

→ **(1) Several levels of expansions**

Fig. 2.6.1 illustrates nested macro calls. The macro call

COMPUTE1 X, Y, Z

Call be expanded (level 1) using the algorithm of macro expansion.

$$\text{COMPUTE1 X, Y, Z} \Rightarrow \left\{ \begin{array}{ll} \text{COMPUTE} & \text{X} \\ \text{COMPUTE} & \text{Y} \\ \text{COMPUTE} & \text{Z} \end{array} \right\}$$

The expanded code itself contains macro calls. The macro expansion algorithm can be applied to the first level expanded code o expand these macro calls and so on, until we obtain a code form which does not contain any macro calls. This approach requires several passes of expansion, which is not desirable.

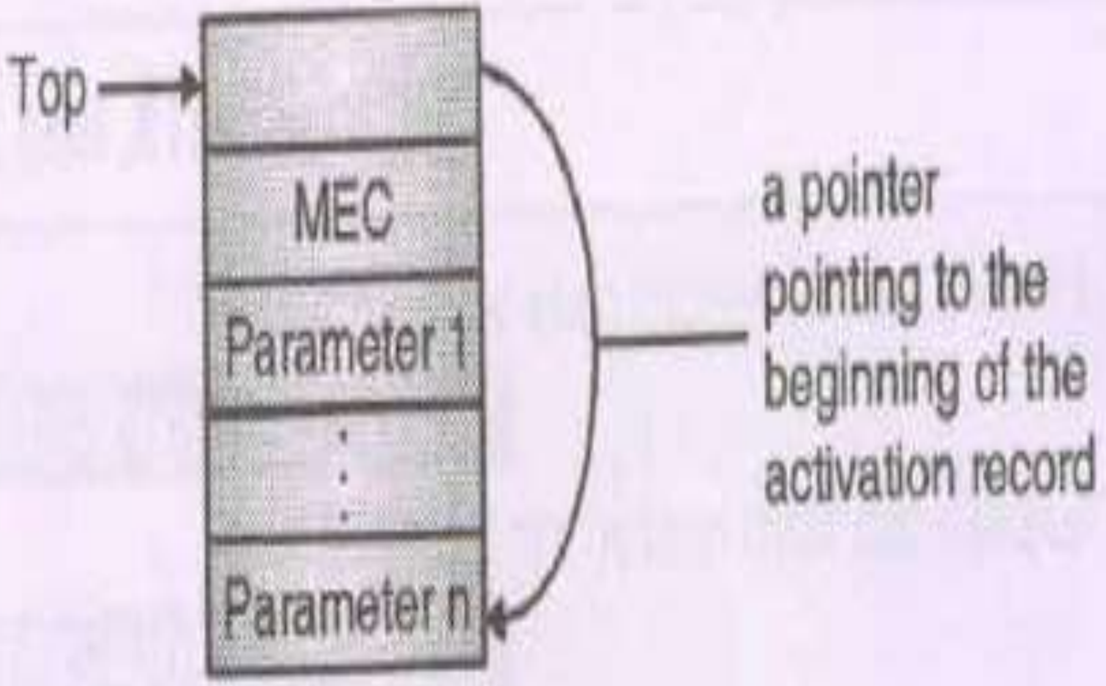
➔ (2) Recursive expansion

To handle nested macro calls, the macro expansion function should be able to work recursively. During recursion, while processing one macro the processing of inner macro can begin and after the expansion of inner macro finishes, the processing of outer macro may continue. A recursion is handled through a stack, where local variables are stored onto the stack before making a recursive call.

➔ (3) Use of stack during expansion

Nested macro calls can be handled with the help of explicit stack.

- Macro calls are handled in LIFO manner.
- Stack can be used to accommodate the expansion time data structure.
- Expansion time data structure include :
  1. MEC – Macro expansion counter
  2. Actual parameter table
- Expansion time data structure is stored in an activation record. The structure of the activation record is given in Fig. 2.6.1.

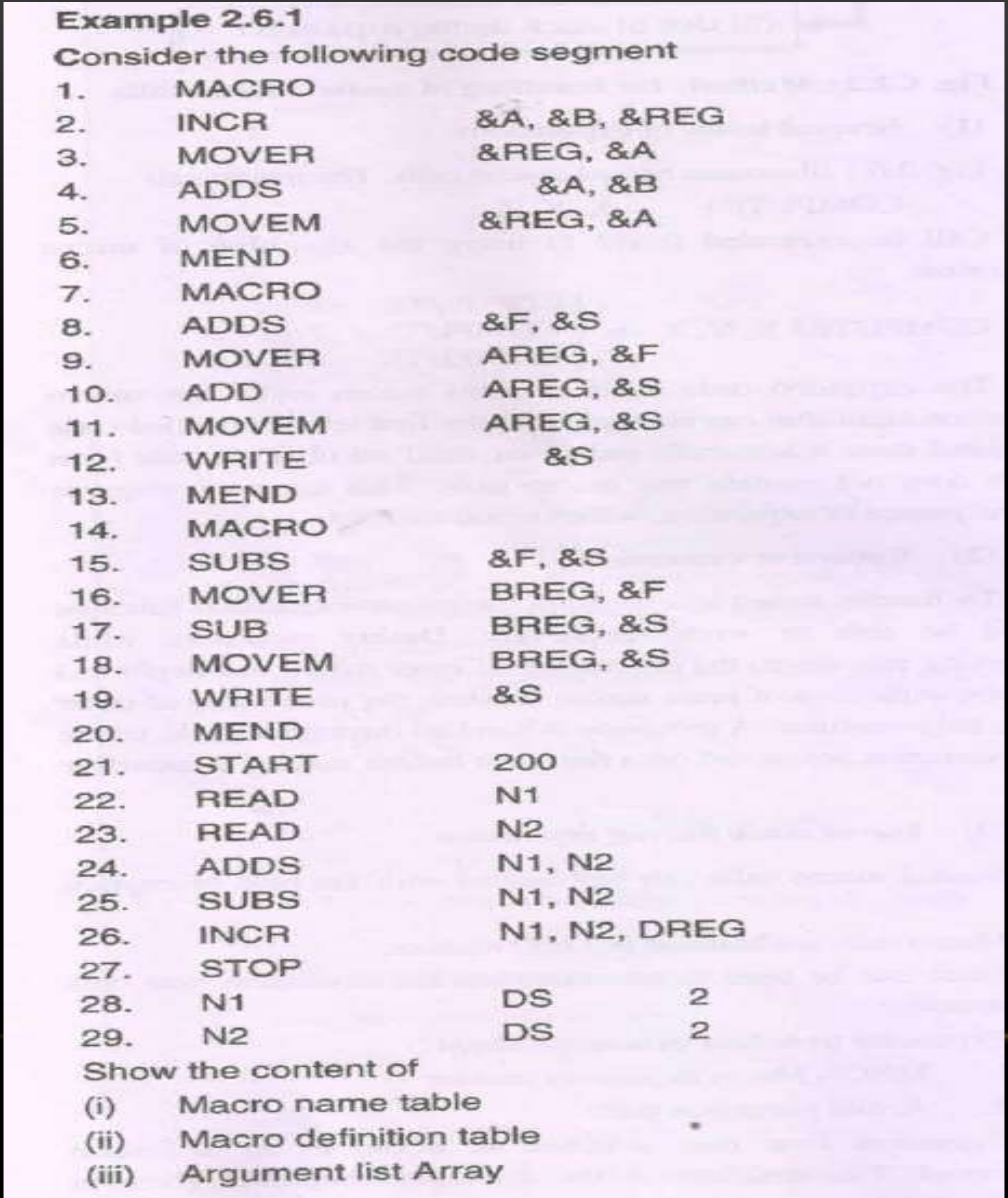


Activation record

(S3.11)Fig. 2.6.1

- Every call to a macro involves pushing an activation record onto the stack.
- At the end of the macro expansion, an activation record is removed from the stack. The top of the stack can be shifted to the next record through the following operation.

```
top = stack[top] - 1;
```



Example 2.6.1

Consider the following code segment

```
1.    MACRO
2.    INCR          &A, &B, &REG
3.    MOVER         &REG, &A
4.    ADDS            &A, &B
5.    MOVEM         &REG, &A
6.    MEND
7.    MACRO
8.    ADDS          &F, &S
9.    MOVER         AREG, &F
10.   ADD           AREG, &S
11.   MOVEM         AREG, &S
12.   WRITE            &S
13.   MEND
14.   MACRO
15.   SUBS          &F, &S
16.   MOVER         BREG, &F
17.   SUB           BREG, &S
18.   MOVEM         BREG, &S
19.   WRITE         &S
20.   MEND
21.   START         200
22.   READ          N1
23.   READ          N2
24.   ADDS          N1, N2
25.   SUBS          N1, N2
26.   INCR          N1, N2, DREG
27.   STOP
28.   N1            DS        2
29.   N2            DS        2
```

Show the content of

(i) Macro name table

(ii) Macro definition table

(iii) Argument list Array

Solution :

Pass I : Contents of MNT and MDT at the end of Pass I.

	MNT	
	Name	**MDTP**
0	INCR	0
1	ADDS	5
2	SUBS	11

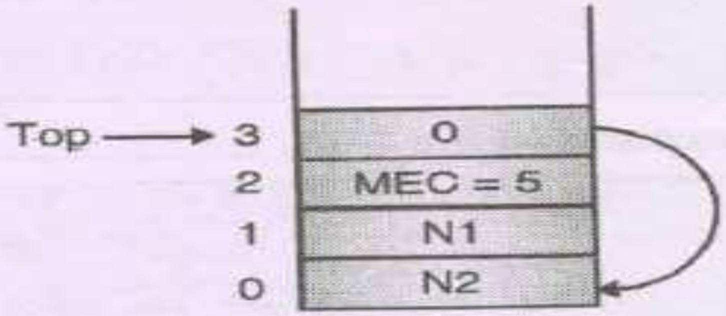
	MDT	
	Opcode	**Rest**
0	INCR	&A, &B, &REG
1	MOVER	#2, #0
2	ADDS	#0, #1
3	MOVEM	#2, #0
4	MEND	
5	ADDS	&F, &S
6	MOVER	AREG, #0
7	ADD	AREG, #1
8	MOVEM	AREG, #1
9	WRITE	#1
10	MEND	
11	SUBS	&F, &S
12	MOVER	BREG, #0
13	SUB	BREG, #1
14	MOVEM	BREG, #1
15	WRITE	#1
16	MEND	

1. **Expansion of line 24 ADDS N1, N2**

Argument list array :

0	N1
1	N2

Activation record on the stack :



Expanded Code :

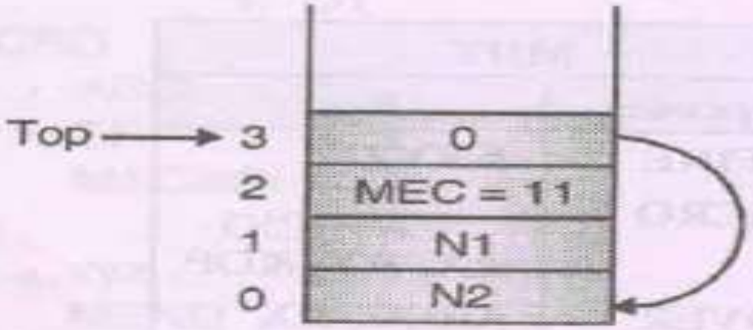
```
MOVER   AREG,  N1
ADD     AREG,  N2
MOVEM   AREG,  N2
WRITE   N2
```

2. **Expansion of line 25 SUBS N1, N2**

Argument list array :

0	N1
1	N2

Activation record on the stack :



Expanded Code :

```
MOVER   BREG,  N1
SUB     BREG,  N2
MOVEM   BREG,  N2
WRITE   N2
```

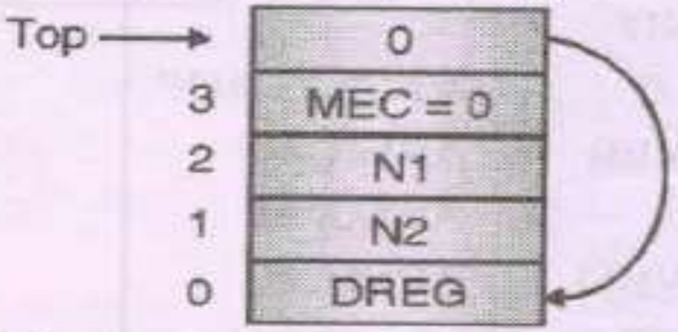
3. **Expansion of line 26 INCR N1, N2, DREG**

Argument list array :

0	N1
1	N2
2	DREG

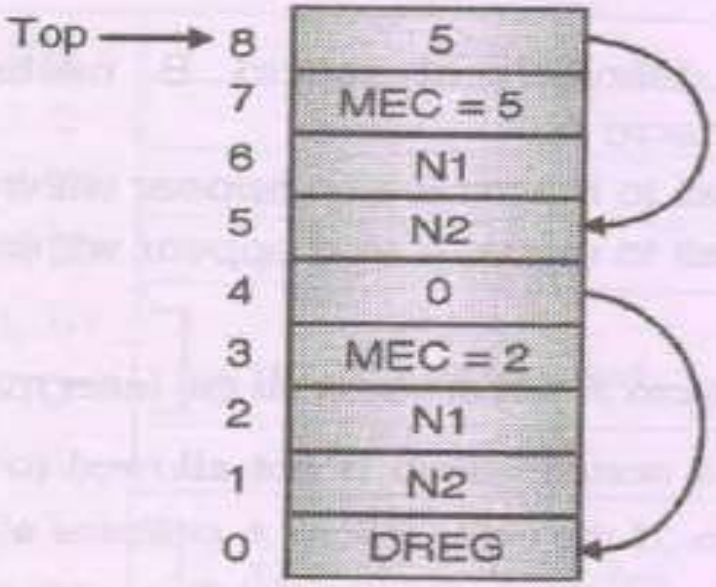
Activation record (stack)

(a) At the beginning



(b) At the time of nested macro call

ADDS #0, #1

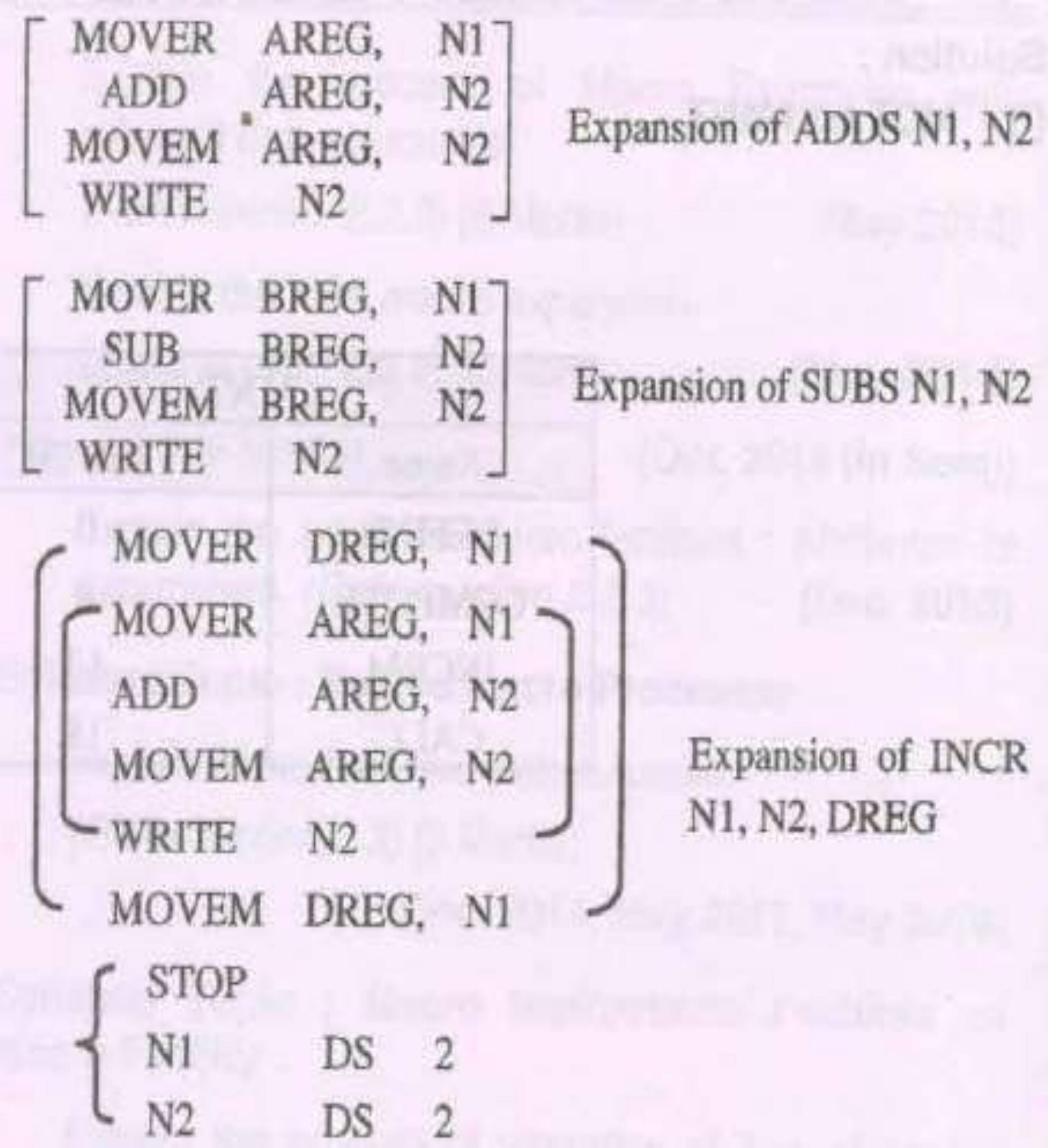


Expanded Code :

```
MOVER   DREG,  N1
MOVER   AREG,  N1
ADD     AREG,  N2
MOVEM   AREG,  N2
WRITE   N2
MOVEM   DREG,  N1
```

Expanded source file at the end of pass II :

```
START   200
READ    N1
READ    N2
```



```
MOVER   AREG,   N1
ADD     AREG,   N2
MOVEM   AREG,   N2
WRITE   N2
```

Expansion of ADDS N1, N2

```
MOVER   BREG,   N1
SUB     BREG,   N2
MOVEM   BREG,   N2
WRITE   N2
```

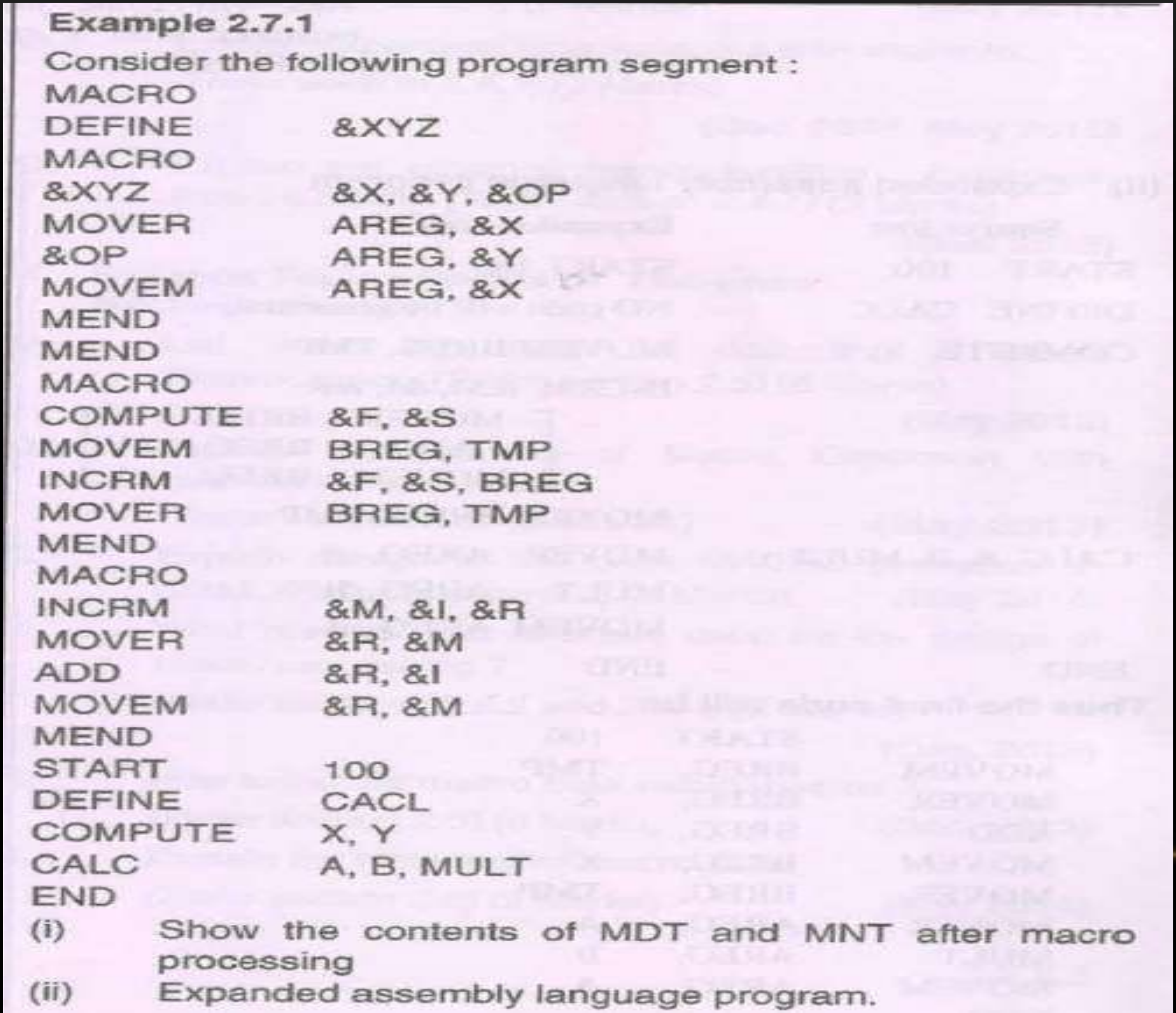
Expansion of SUBS N1, N2

```
MOVER   DREG,   N1
MOVER   AREG,   N1
ADD     AREG,   N2
MOVEM   AREG,   N2
WRITE   N2
MOVEM   DREG,   N1
```

Expansion of INCR N1, N2, DREG

```
STOP
N1      DS   2
N2      DS   2
```

HANDLING OF NESTED MACRO DECLARATION



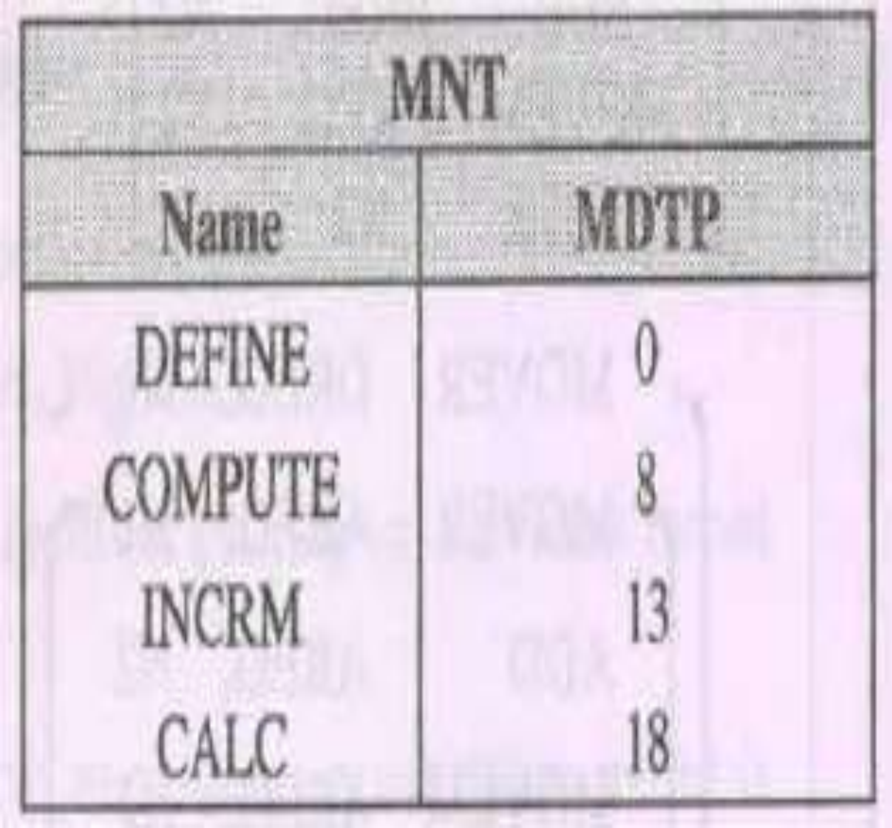
Example 2.7.1

Consider the following program segment :

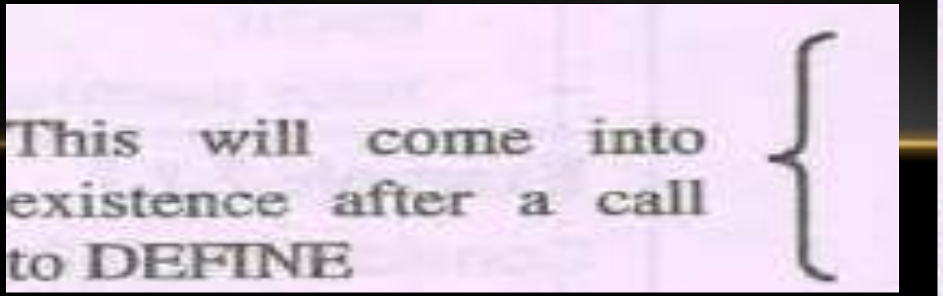
```
MACRO
DEFINE        &XYZ
MACRO
&XYZ          &X, &Y, &OP
MOVER         AREG, &X
&OP           AREG, &Y
MOVEM         AREG, &X
MEND
MEND
MACRO
COMPUTE       &F, &S
MOVEM         BREG, TMP
INCRM         &F, &S, BREG
MOVER         BREG, TMP
MEND
MACRO
INCRM         &M, &I, &R
MOVER         &R, &M
ADD           &R, &I
MOVEM         &R, &M
MEND
START         100
DEFINE        CACL
COMPUTE       X, Y
CALC          A, B, MULT
END
```

(i) Show the contents of MDT and MNT after macro processing

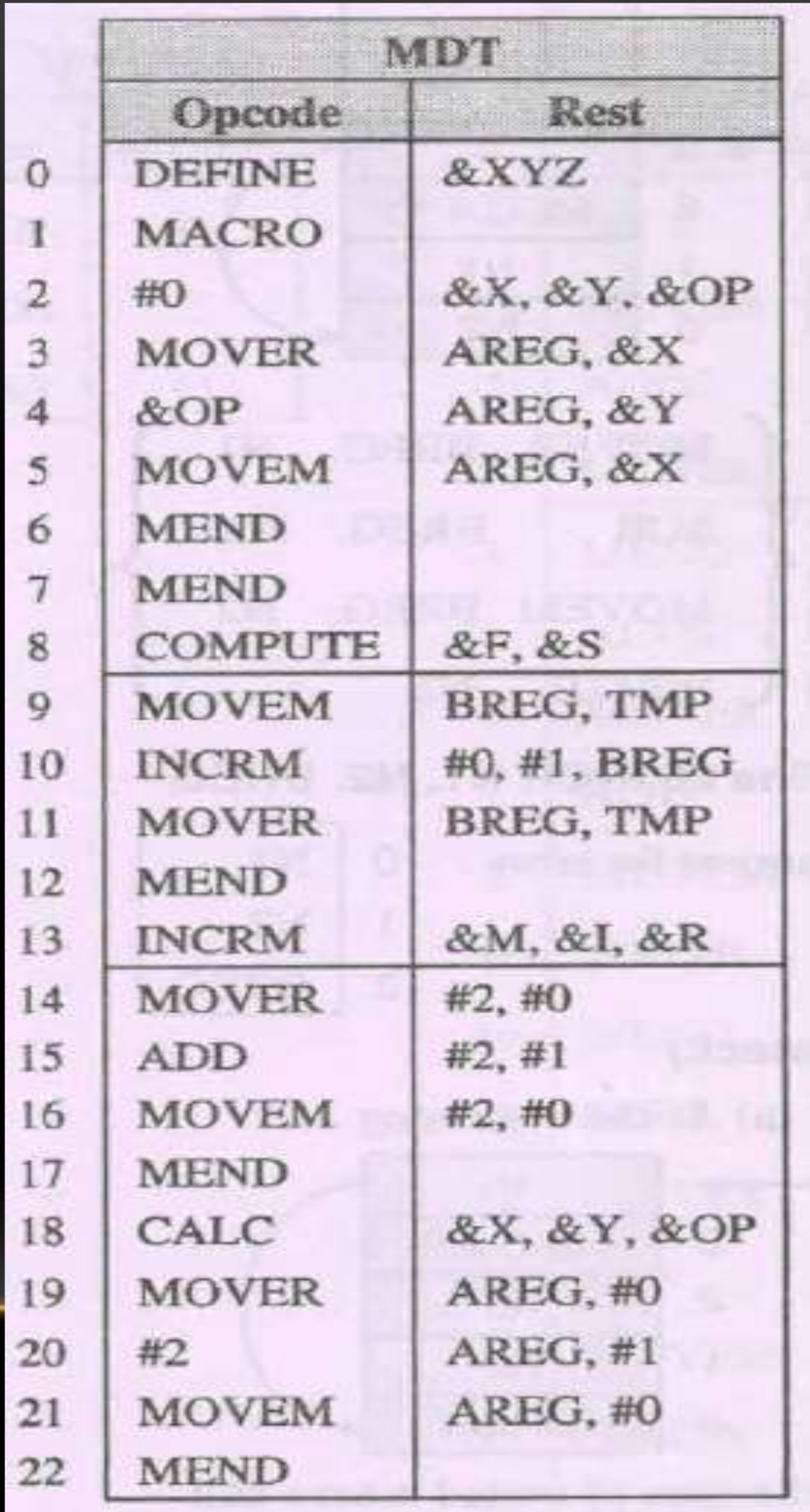
(ii) Expanded assembly language program.



MNT	
Name	**MDTP**
DEFINE	0
COMPUTE	8
INCRM	13
CALC	18



This will come into existence after a call to DEFINE



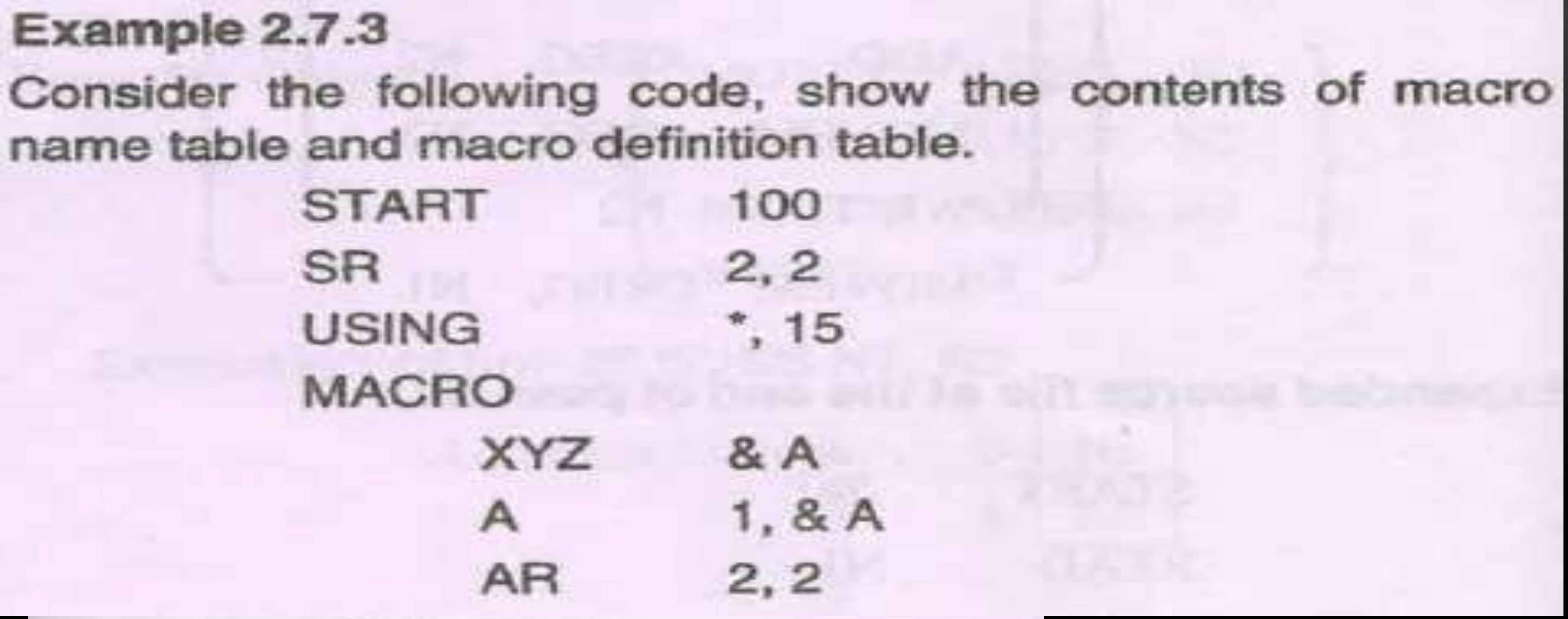
	MDT	
	Opcode	**Rest**
0	DEFINE	&XYZ
1	MACRO	
2	#0	&X, &Y, &OP
3	MOVER	AREG, &X
4	&OP	AREG, &Y
5	MOVEM	AREG, &X
6	MEND	
7	MEND	
8	COMPUTE	&F, &S
9	MOVEM	BREG, TMP
10	INCRM	#0, #1, BREG
11	MOVER	BREG, TMP
12	MEND	
13	INCRM	&M, &I, &R
14	MOVER	#2, #0
15	ADD	#2, #1
16	MOVEM	#2, #0
17	MEND	
18	CALC	&X, &Y, &OP
19	MOVER	AREG, #0
20	#2	AREG, #1
21	MOVEM	AREG, #0
22	MEND	

(ii) Expanded assembly language program

```
   Source line                  Expanded code
 START    100           –       START 100
 DEFINE   CALC          –       NO code will be generated
 COMPUTE  X, Y          –       MOVEM BREG, TMP
                                INCRM  &M, &I, &R
                                       ┌ MOVER    BREG,   X ┐
                                      –│  ADD     BREG,   Y │
                                       └ MOVEM    BREG,   X ┘
                                MOVER  BREG, TMP
 CALC  A, B, MULT       –       MOVER   AREG, A
                                MULT    AREG, B
                                MOVEM  AREG, A
 END                    –       END
```

Thus the final code will be

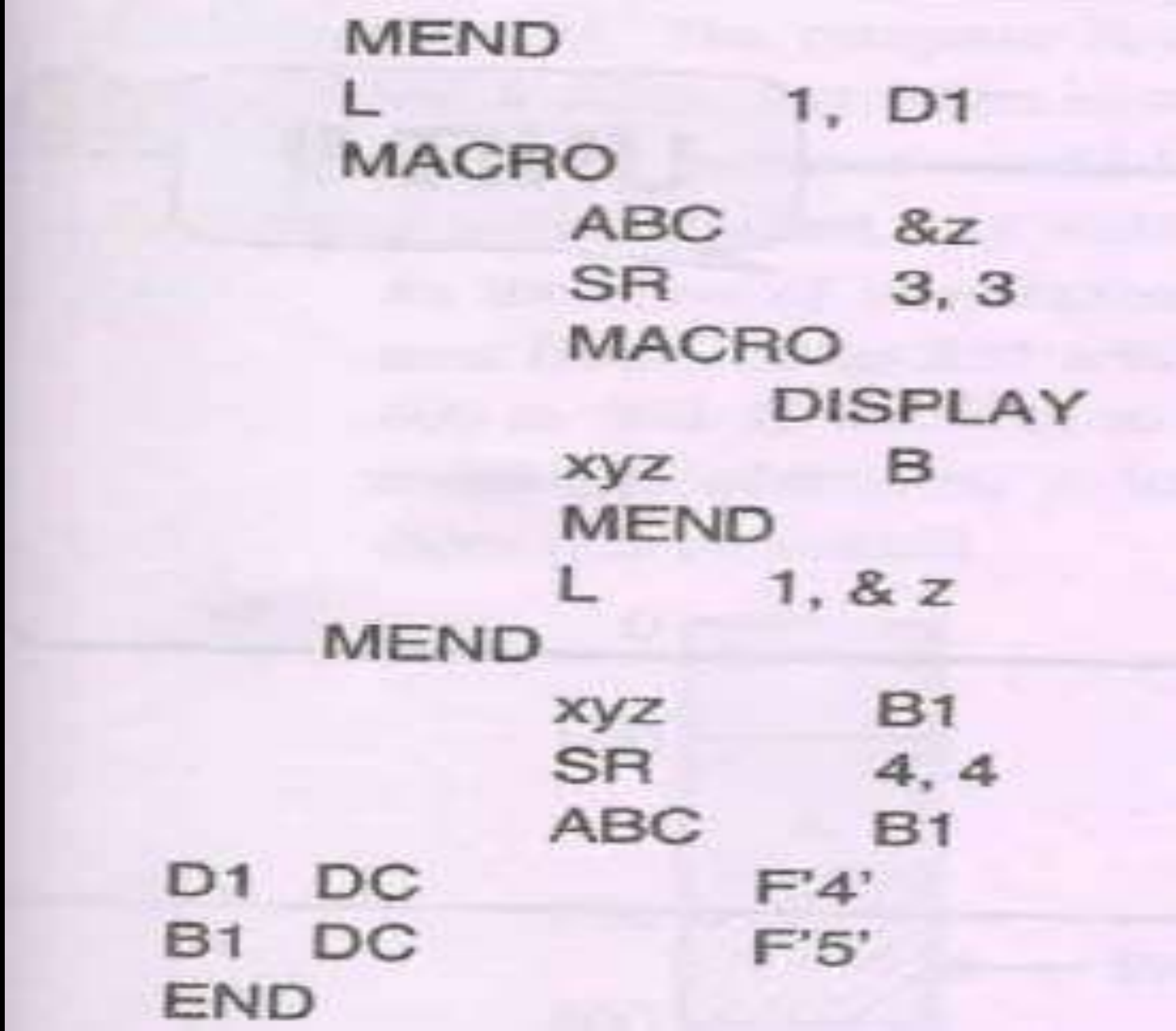
```
                 START    100
     MOVEM       BREG,    TMP
     MOVER       BREG,    X
     ADD         BREG,    Y
     MOVEM       BREG,    X
     MOVER       BREG,    TMP
     MOVER       AREG,    A
     MULT        AREG,    B
     MOVEM       AREG,    A
     END
```



Example 2.7.3

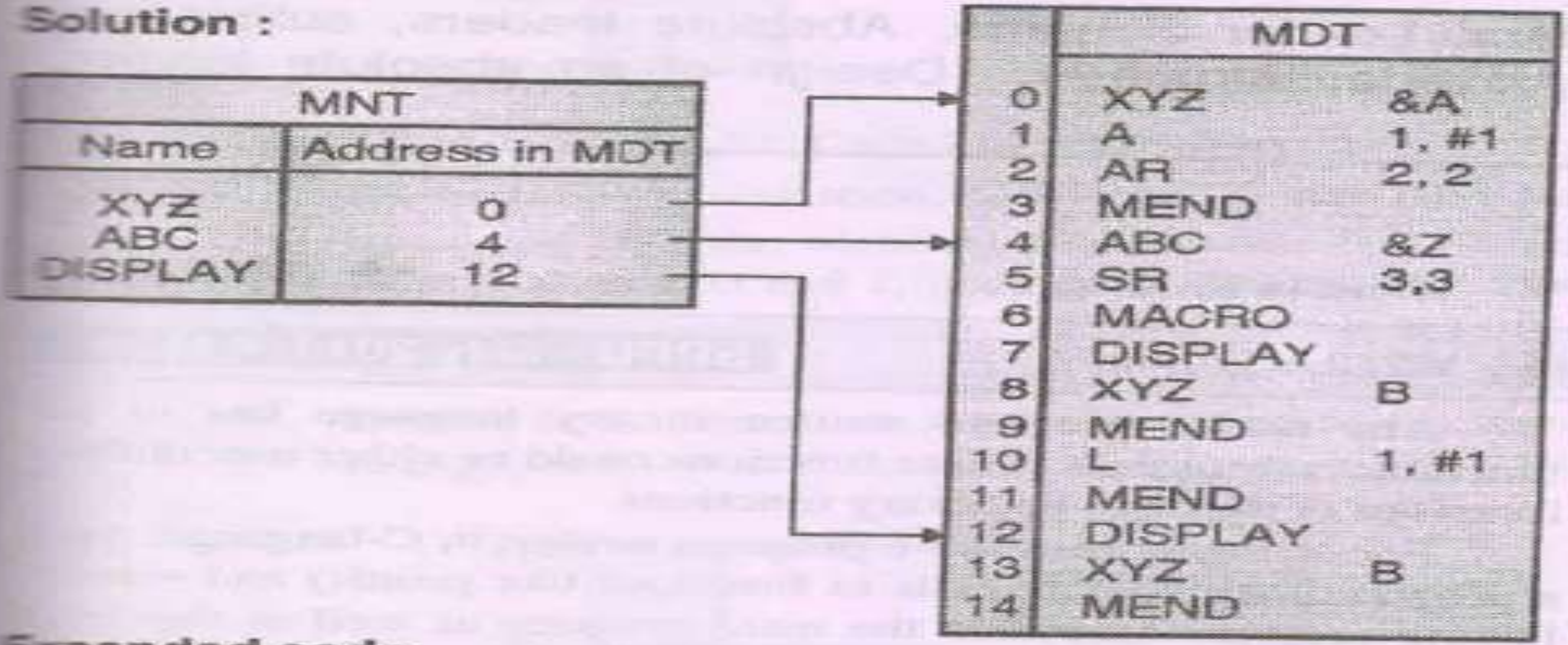
Consider the following code, show the contents of macro name table and macro definition table.

```
        START        100
        SR           2, 2
        USING        *, 15
        MACRO
             XYZ     & A
             A       1, & A
             AR      2, 2
```



```
        MEND
        L                1,  D1
        MACRO
             ABC          &z
             SR           3, 3
             MACRO
                  DISPLAY
             xyz          B
             MEND
             L        1, & z
        MEND
             xyz          B1
             SR           4, 4
             ABC          B1
   D1   DC            F'4'
   B1   DC            F'5'
   END
```

Solution :



MNT	
Name	Address in MDT
XYZ	0
ABC	4
DISPLAY	12

	MDT	
0	XYZ	&A
1	A	1, #1
2	AR	2, 2
3	MEND	
4	ABC	&Z
5	SR	3,3
6	MACRO	
7	DISPLAY	
8	XYZ	B
9	MEND	
10	L	1, #1
11	MEND	
12	DISPLAY	
13	XYZ	B
14	MEND	

Expanded code

	START	100	
	SR	2, 2	
	USING	*, 15	
	L	1, D1	
+	A	1, B1	xyz B1 is expanded
+	AR	2, 2	
	SR	4, 4	
+	SR	3, 3	ABC B1 is expanded
+	L	1, B1	
D1	DC	F '4'	
B1	DC	F '5'	
	END		

Compiler

History of Compiler

- The “compiler” word was first used in the early 1950s by Grace Murray Hopper
- The first compiler was build by John Backum and his group between **1954 and 1957** at **IBM**
- **COBOL** was the first programming language which was compiled on multiple platforms in 1960
- The study of the scanning and parsing issues were pursued in the 1960s and 1970s to provide a complete solution

Compiler

- Compiler is a translator which converts the high level language into low level language.
- Benefits of writing a program in a high level language :
- **Increases productivity:** It is very easy to write a program in a high level language.
- **Machine Independence:** A program written in a high level language is machine independent.

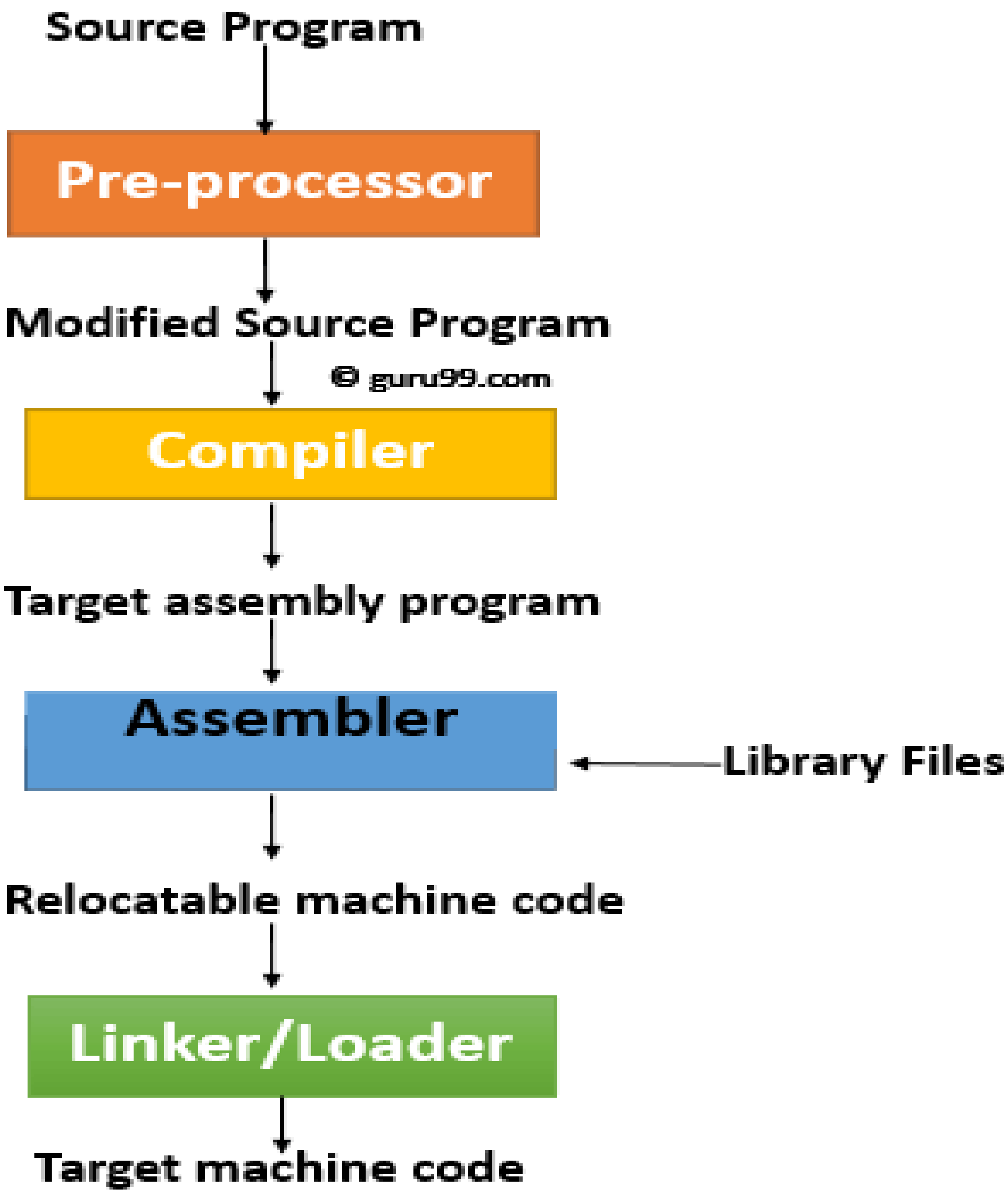
Features of Compiler

- Compilation speed.
- The correctness of machine code.
- The meaning of code should not change.
- Speed of machine code.
- Good error detection.
- Checking the code correctly according to grammar.

Uses / Applications of Compiler

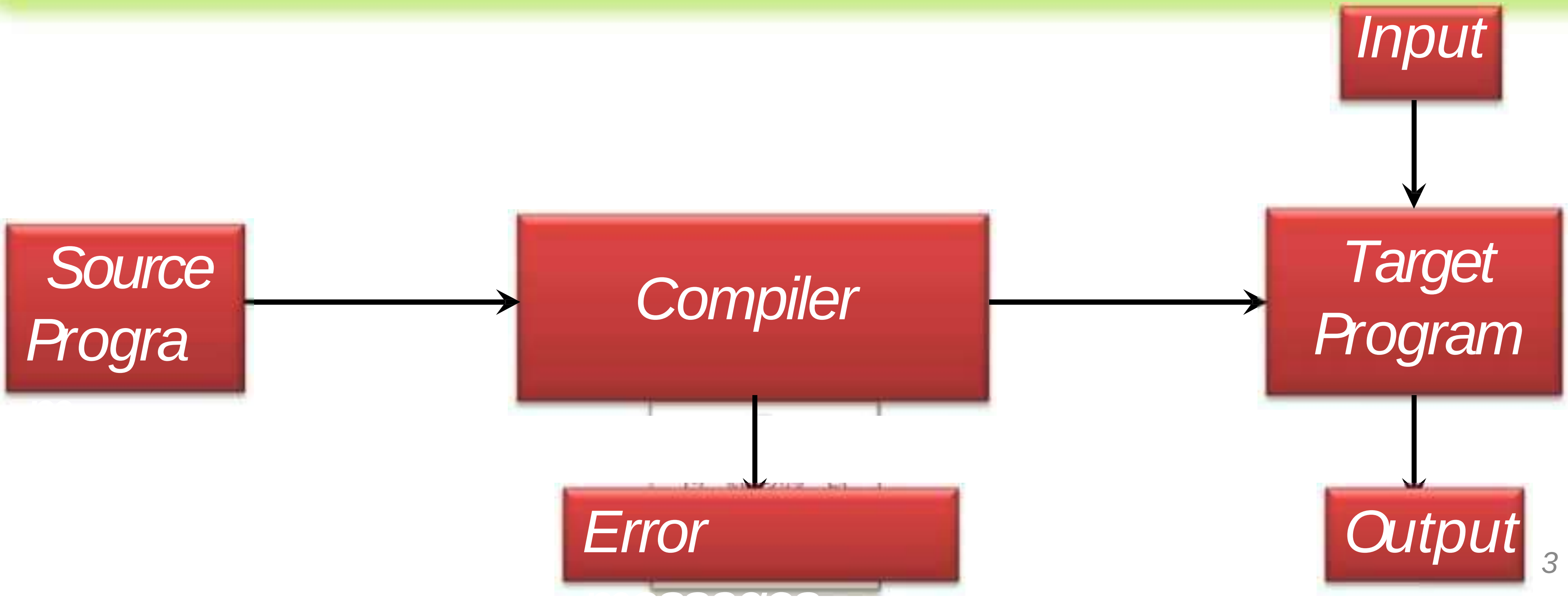
- Helps to make the code independent of the platform.
- Makes the code free of syntax and semantic errors.
- Generate executable files of code.
- Translates the code from one language to another.

Steps in language processing system



COMPILERS

- *"Compilation"*
  - *Translation of a program written in a source language into a semantically equivalent program written in a target language*



Input

Source Progra

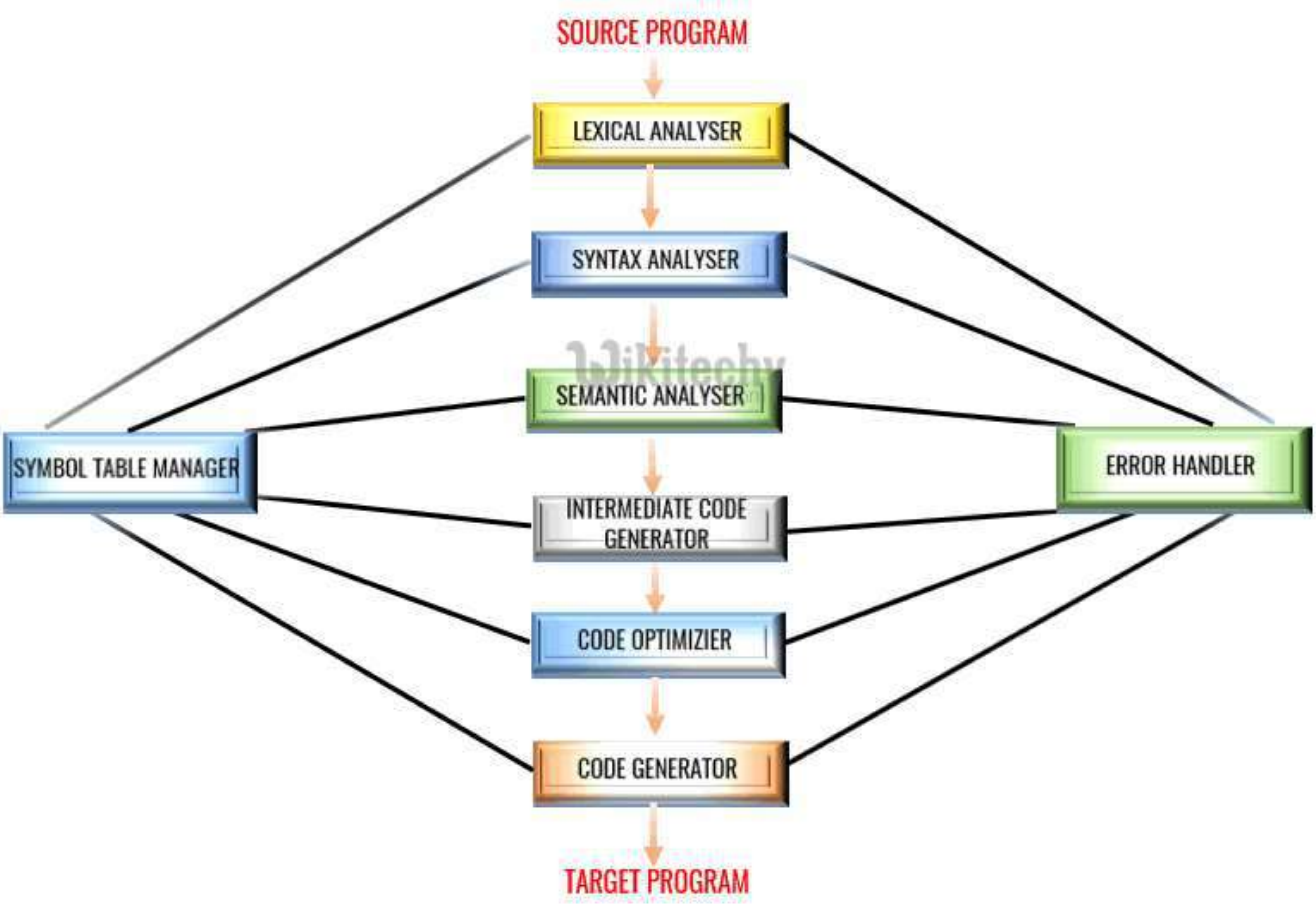
Compiler

Target Program

Error

Output

Phases of Compiler



ANALYSIS – SYNTHESIS MODEL

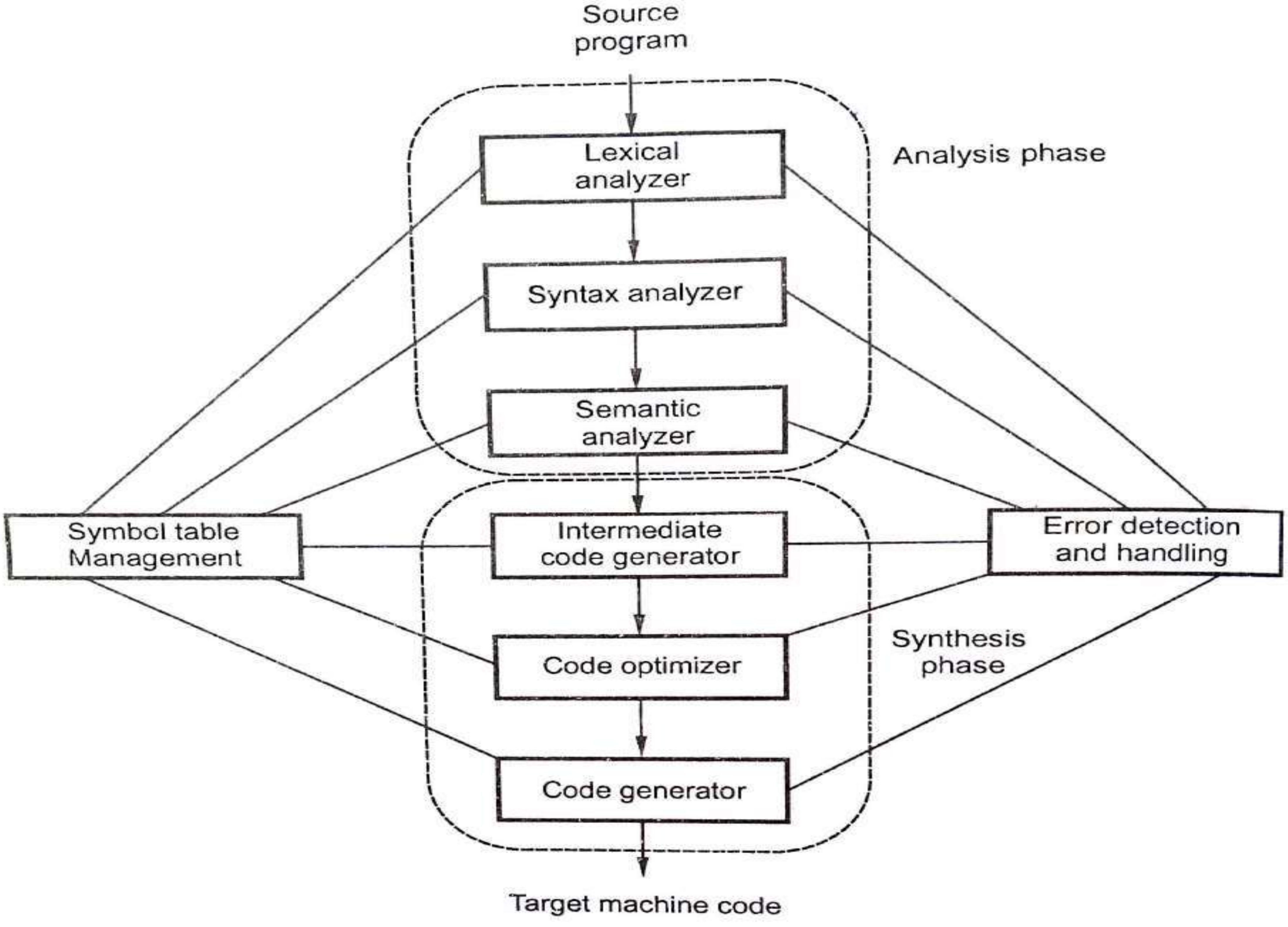
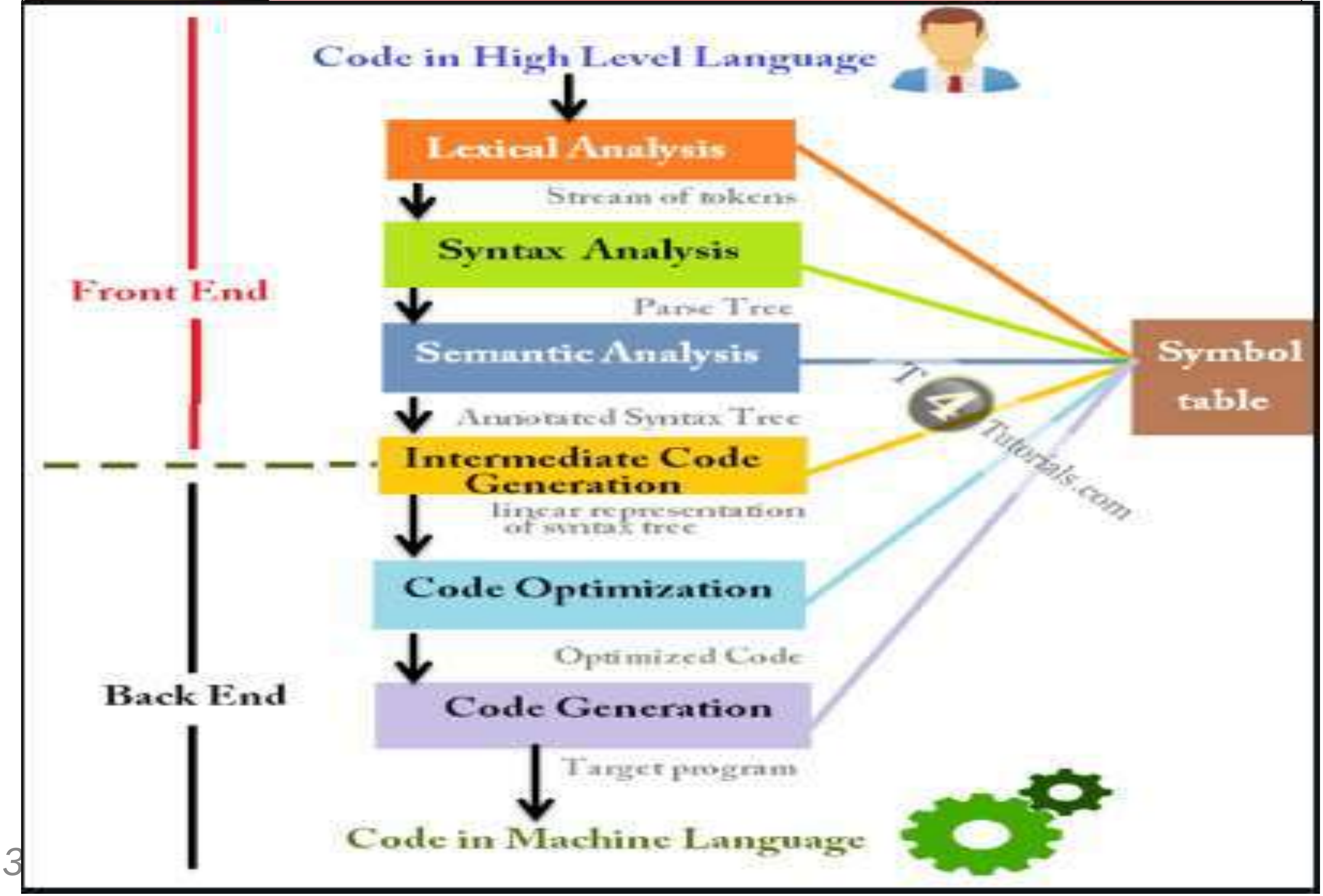


Fig. 1.7 Phases of compiler

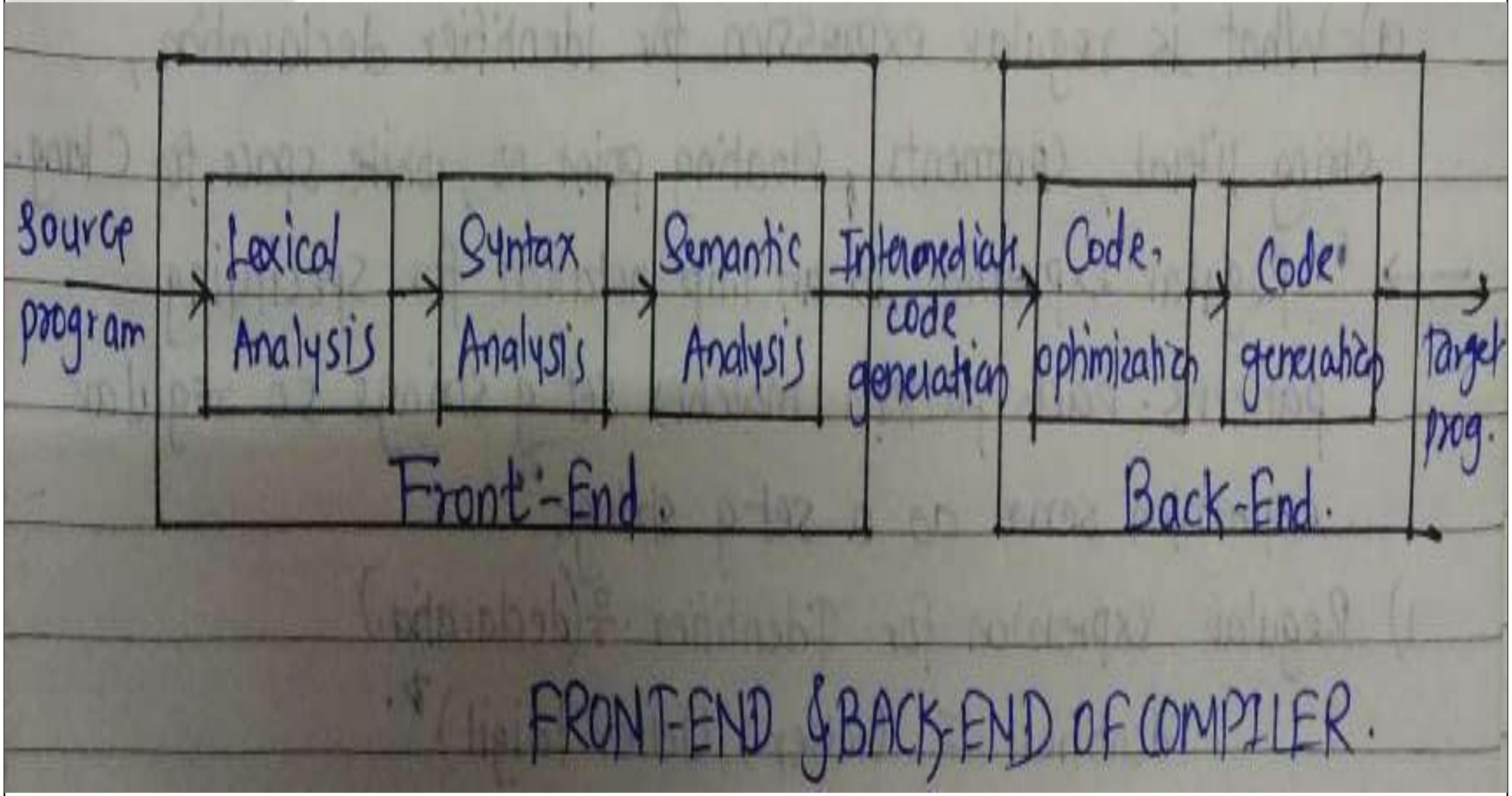
Example : https://youtu.be/P1bQUyl__t0

PHASES OF COMPILER



Example : https://youtu.be/P1bQUyl__t0

FRONT END & BACK END OF COMPILER



Phases of Compiler

- **Lexical Analysis :**

1. It takes the high-level language source code as the input.
2. It scans the characters of source code from left to right. Hence, the name scanner also.
3. It groups the characters into lexemes. Lexemes are a group of characters which has some meaning.
4. Each lexeme corresponds to form a token.
5. It removes white spaces and comments.
6. It checks and removes the lexical errors.

Phases of Compiler

- **Syntax Analysis :**

1. ‘Parser’ is the other name for the syntax analyzer.
2. The output of the lexical analyzer is its input.
3. It checks for syntax errors in the source code.
4. It does this by constructing a parse tree of all the tokens.
5. For the syntax to be correct, the parse tree should be according to the rules of source code grammar.
6. The grammar for such codes is context-free grammar.

Phases of Compiler

- **Semantic Analysis :**

1. It verifies the parse tree of the syntax analyzer.
2. It checks the validity of the code in terms of programming language. Like, compatibility of data types, declaration, and initialization of variables, etc.
3. It also produces a verified parse tree. Furthermore, we also call this tree an annotated parse tree.
4. It also performs flow checking, type checking, etc.

Phases of Compiler

- **Intermediate Code Generation :**

1. It generates an intermediate code.
2. This code is neither in high-level language nor in machine language. It is in an intermediate form.
3. It is converted to machine language but, the last two phases are platform dependent.
4. The intermediate code is the same for all the compilers. Further, we generate the machine code according to the platform.
5. An example of an intermediate code is three address code.

Phases of Compiler

- **Code Optimizer :**

1. It optimizes the intermediate code.
2. Its function is to convert the code so that it executes faster using fewer resources (CPU, memory).
3. It removes any useless lines of code and rearranges the code.
4. The meaning of the source code remains the same.

Phases of Compiler

- **Code Generator :**

1. Finally, it converts the optimized intermediate code into the machine code.
2. This is the final stage of the compilation.
3. The machine code which is produced is relocatable.

Phases of Compiler

- **Lexical Analysis :**

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

- **Syntax Analysis :**

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Phases of Compiler

- **Semantic Analysis :**

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

- **Intermediate Code Generation :**

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Phases of Compiler

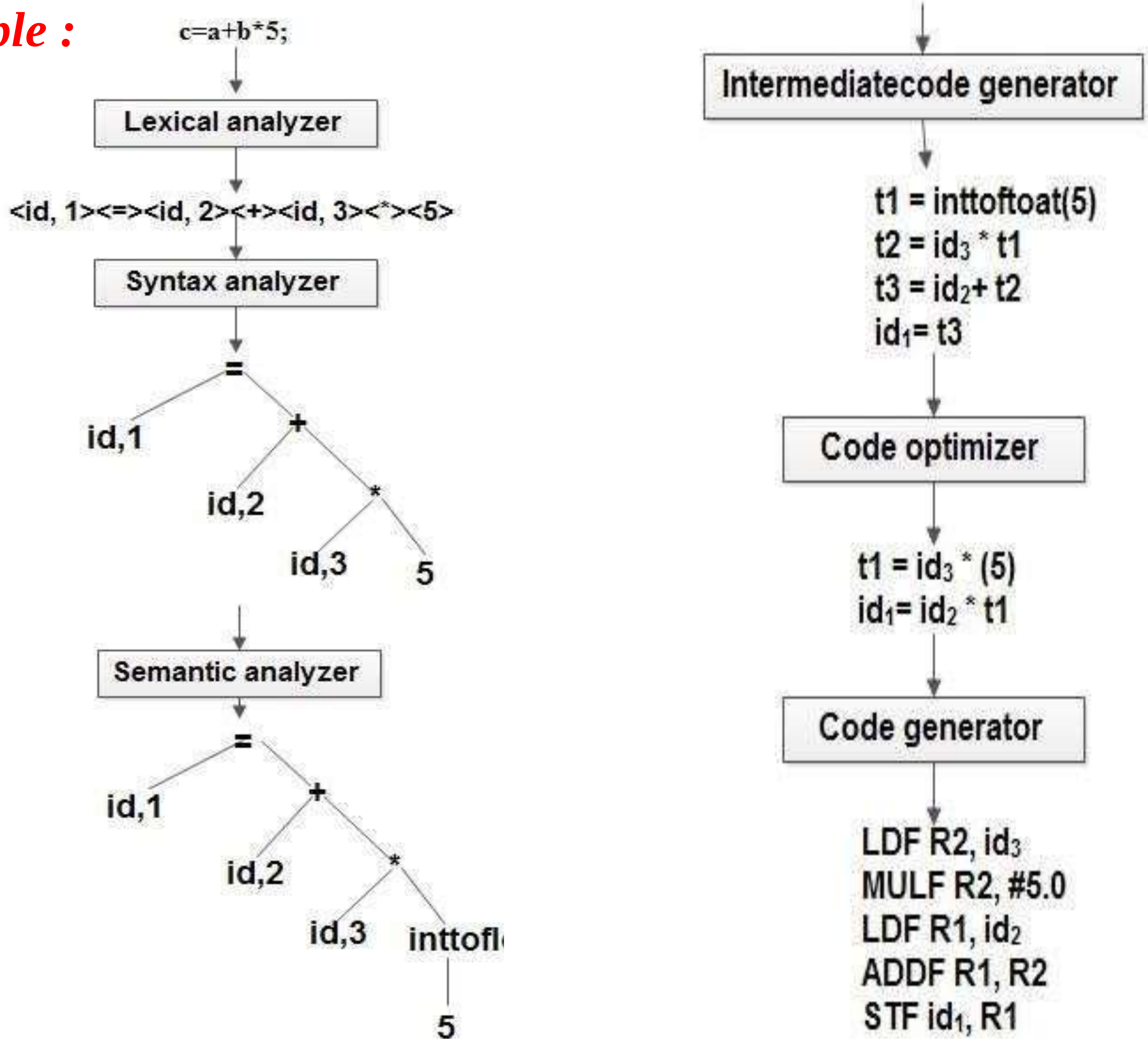
- **Code Optimization :**

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

- **Code Generation :**

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

Example :



c=a+b*5;

Lexical analyzer

<id, 1><=><id, 2><+><id, 3><*><5>

Syntax analyzer

Semantic analyzer

Intermediatecode generator

t1 = inttoftoat(5)
t2 = id_3 * t1
t3 = id_2+ t2
id_1= t3

Code optimizer

t1 = id_3 * (5)
id_1= id_2 * t1

Code generator

LDF R2, id_3
MULF R2, #5.0
LDF R1, id_2
ADDF R1, R2
STF id_1, R1

Types of Compiler

1. Cross Compilers

They produce an executable machine code for a platform but, this platform is not the one on which the compiler is running.

2. Bootstrap Compilers

The process of writing a compiler (or Assembler) in the target programming language which has to be compiled is known as "Bootstrapping“

3. Source to source/transcompiler

These compilers convert the source code of one programming language to the source code of another programming language.

Types of Compiler

4. Incremental compiler :

Incremental Compiler is a compiler, which performs the recompilation of only modified source rather than compiling the whole source program

Decompiler

Basically, it is not a compiler. It is just the reverse of the compiler. It converts the machine code into high-level language.

ISSUES IN COMPILATION

Hierarchy of operations need to be maintained to determine correct order of expression evaluation

Maintain **data type integrity** with automatic type conversions

Handle **user defined data types.**

Develop appropriate **storage mappings**

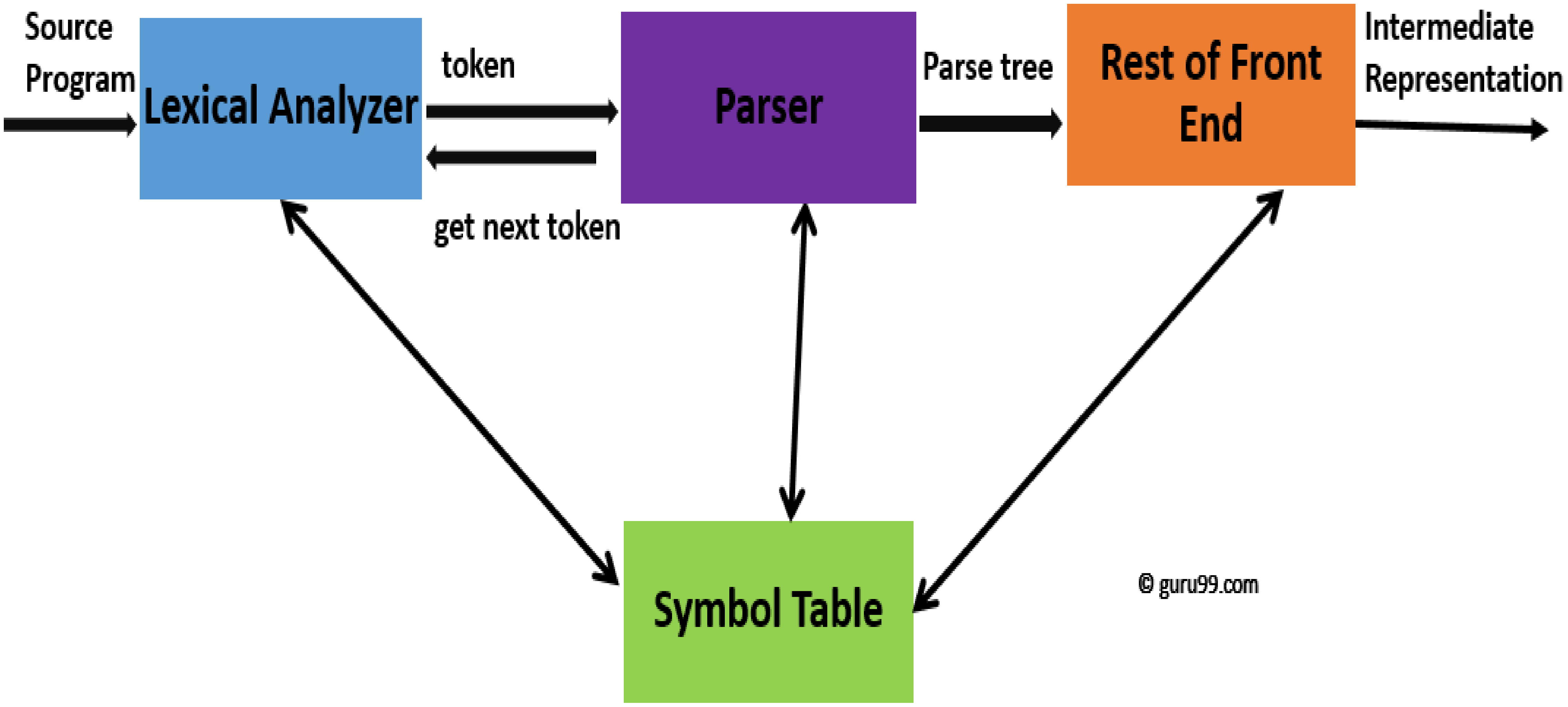
ISSUES IN COMPILATION

Resolve occurrence of each variable name in a program i.e **construct separate symbol tables for different namespaces.**

Handle different control structures.

Perform **optimization**

BLOCK SCHEMATIC OF LEXICAL ANALYZER



© guru99.com

BLOCK SCHEMATIC OF LEXICAL ANALYZER

Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.

A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.

Lexical analysis consists of two stages of processing which are as follows:

- Scanning
- Tokenization

BLOCK SCHEMATIC OF LEXICAL ANALYZER

Roles of the Lexical analyzer

Lexical analyzer performs below given tasks:

1. Helps to identify token into the symbol table
2. Removes white spaces and comments from the source program
3. Correlates error messages with the source program
4. Helps you to expands the macros if it is found in the source program
5. Read input characters from the source program.

BLOCK SCHEMATIC OF LEXICAL ANALYZER

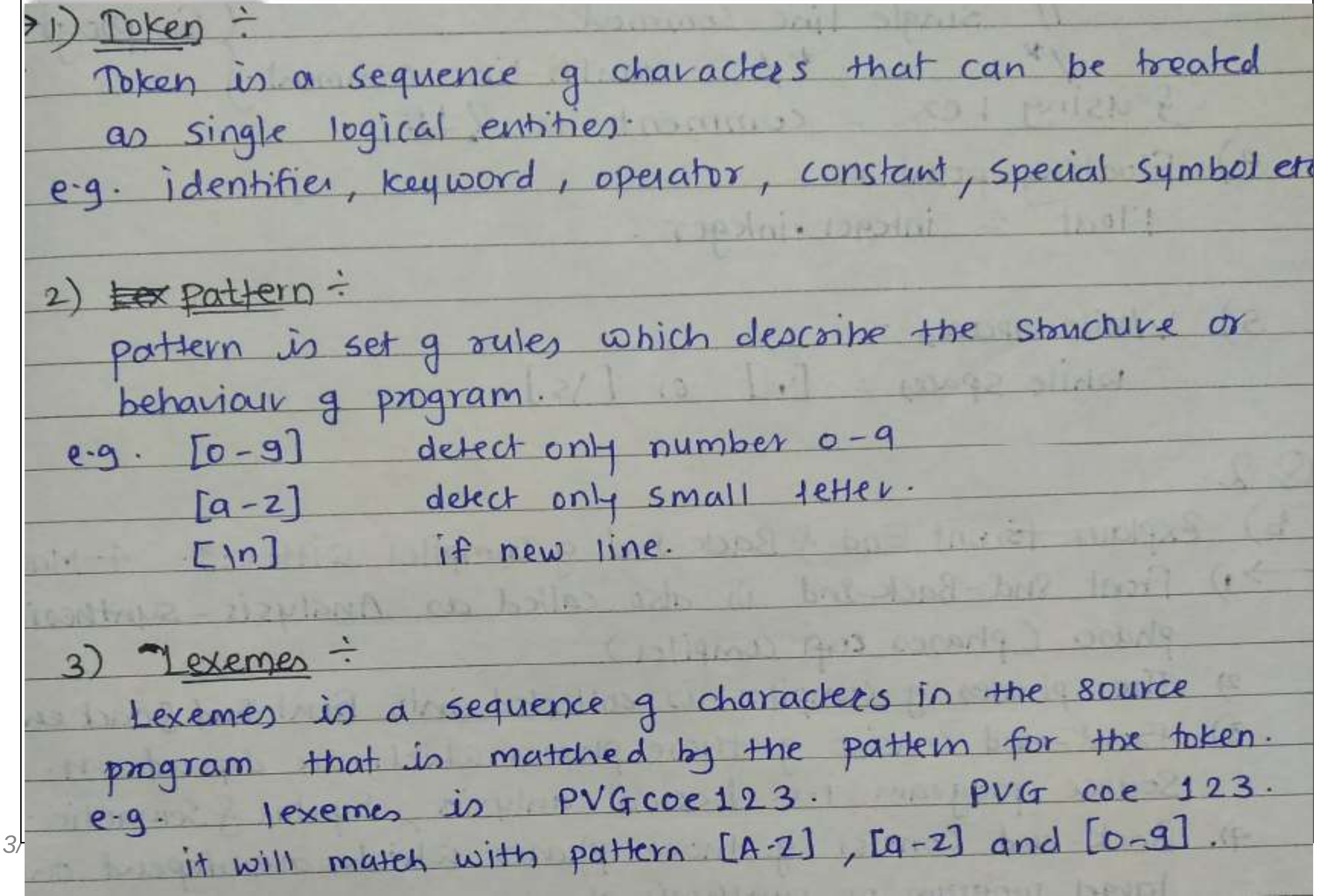
Lexical Analyzer vs. Parser

Lexical Analyser	Parser
Scan Input program	Perform syntax analysis
Identify Tokens	Create an abstract representation of the code
Insert tokens into Symbol Table	Update symbol table entries
It generates lexical errors	It generates a parse tree of the source code

BASIC TERMINOLOGIES OF LEXICAL ANALYSIS

- Major Terms for Lexical Analysis?
  - **TOKEN**
    - A classification for a common set of strings
    - Examples Include <Identifier>, <number>, etc.
  - **PATTERN**
    - The rules which characterize the set of strings for a token
    - Recall File and OS Wildcards ([A-Z]*.*)
  - **LEXEME**
    - Actual sequence of characters that matches pattern and is classified by a token
    - Identifiers: x, count, name, etc…

BASIC TERMINOLOGIES OF LEXICAL ANALYSIS



1) Token :

Token is a sequence of characters that can be treated as single logical entities.

e.g. identifier, keyword, operator, constant, special symbol et

2) ~~Lex~~ Pattern :

pattern is set of rules which describe the structure or behaviour of program.

e.g.

[0-9]	detect only number 0-9
[a-z]	detect only small letter.
[\n]	if new line.

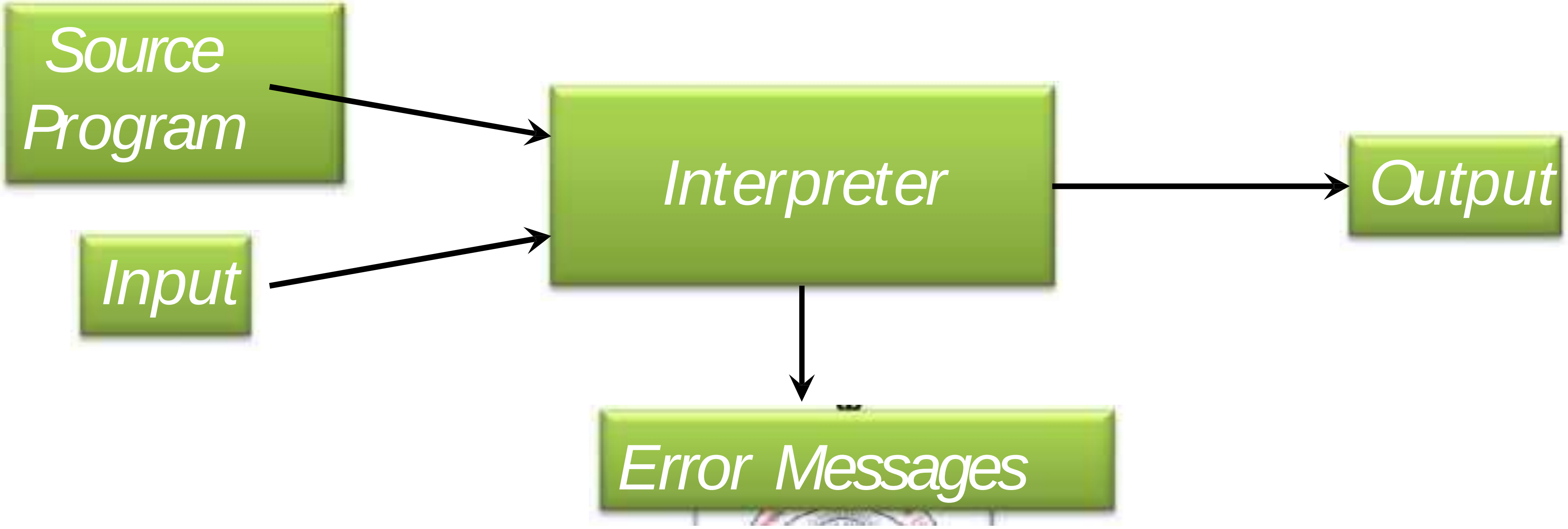
3) Lexemes :

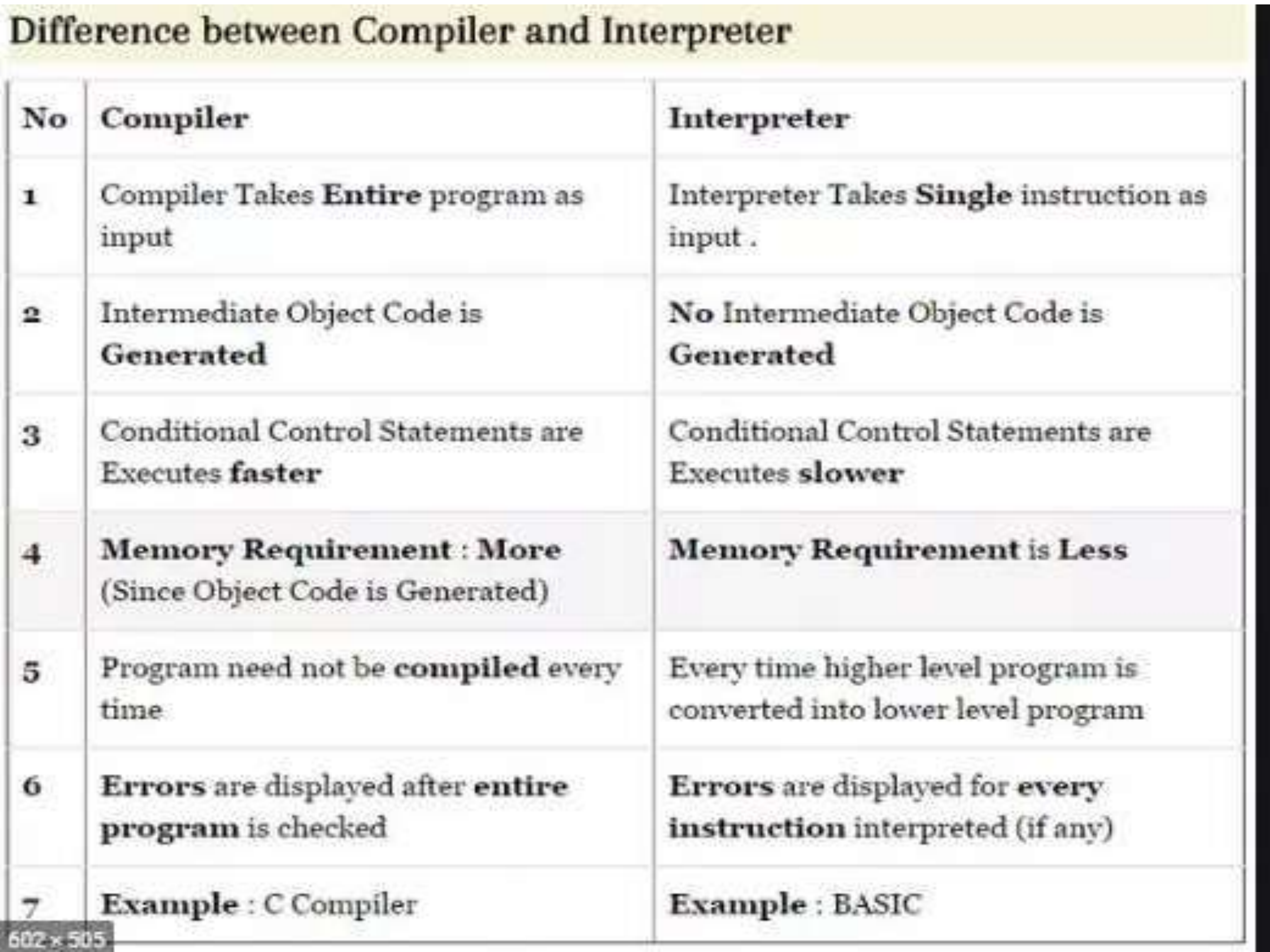
Lexemes is a sequence of characters in the source program that is matched by the pattern for the token.

e.g. lexemes is PVGcoe123. PVG coe 123.
it will match with pattern [A-Z], [a-z] and [0-9].

INTERPRTERS

- "Interpretation"
  - – *Performing the operations implied by the source program*

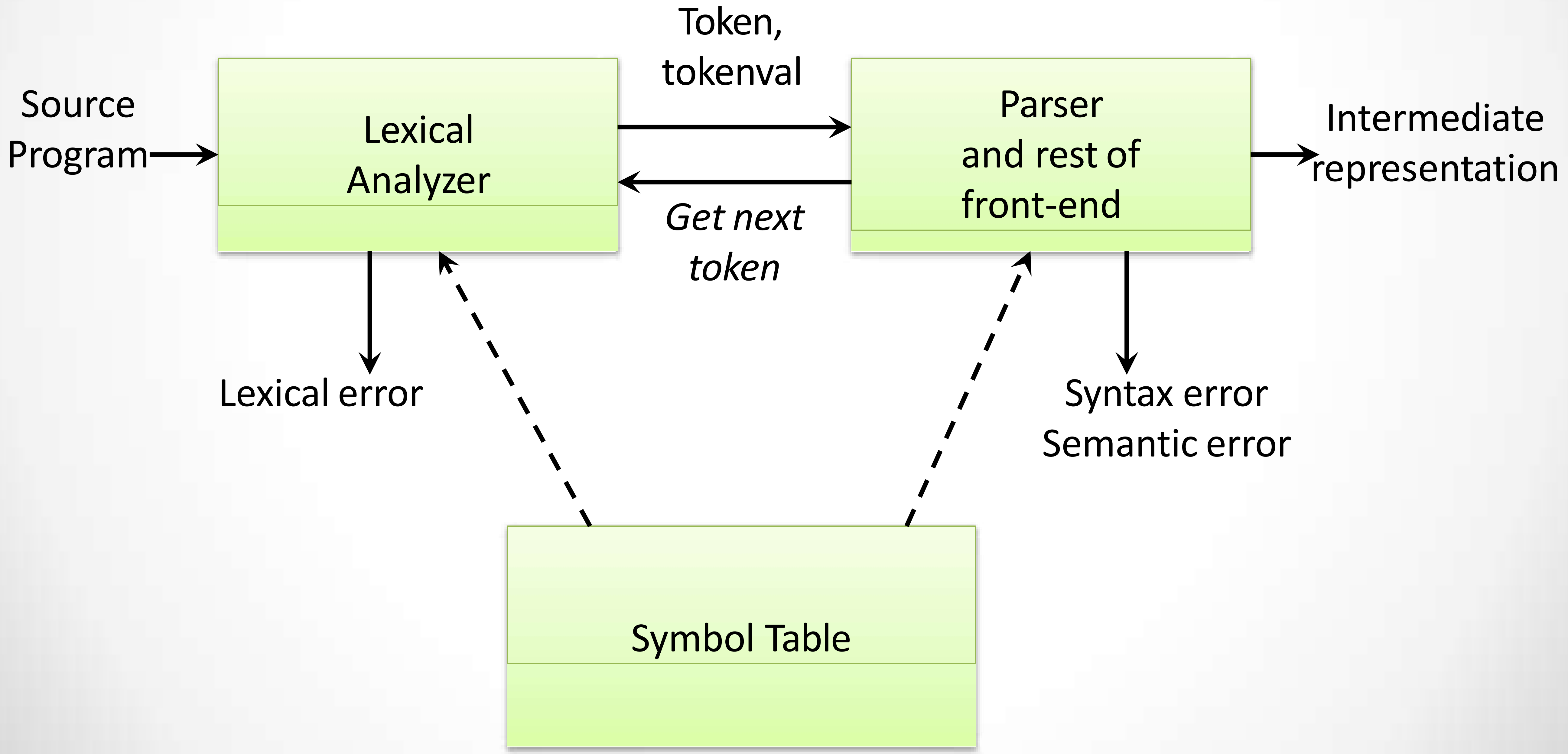




Difference between Compiler and Interpreter

No	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input .
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is **Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement** : **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C Compiler	**Example** : BASIC

Position of a Parser in the Compiler Model



Source Program

Lexical Analyzer

Token, tokenval

Get next token

Parser and rest of front-end

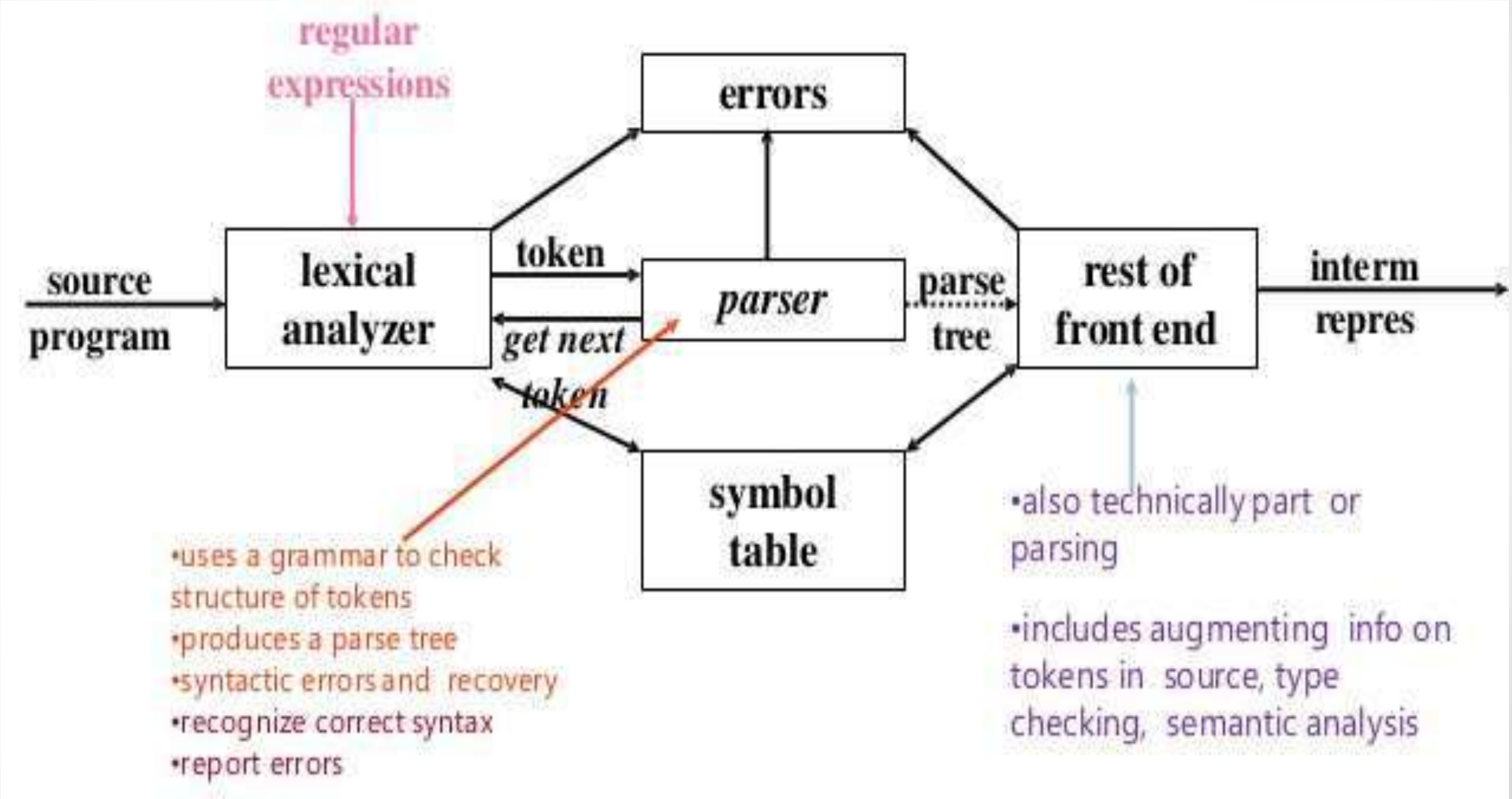
Intermediate representation

Lexical error

Syntax error
Semantic error

Symbol Table

Position of a Parser in the Compiler Model



- uses a grammar to check structure of tokens
- produces a parse tree
- syntactic errors and recovery
- recognize correct syntax
- report errors

- also technically part or parsing
- includes augmenting info on tokens in source, type checking, semantic analysis

The Role Of Parser

- A parser implements a C-F grammar
- The role of the parser is two fold:

1. To check syntax (= string recognizer)
   - And to report syntax errors accurately
2. To invoke semantic actions
   - For static semantics checking, e.g. type checking of expressions, functions, etc.
   - For syntax-directed translation of the source code to an intermediate representation

The Role Of Parser

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues :

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

The Role Of Parser

Syntax error handling :

Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling an identifier, keyword or operator.

2. Syntactic, such as an arithmetic expression with unbalanced parentheses.

3. Semantic, such as an operator applied to an incompatible operand.

4. Logical, such as an infinitely recursive call.

The Role Of Parser

Functions of error handler :

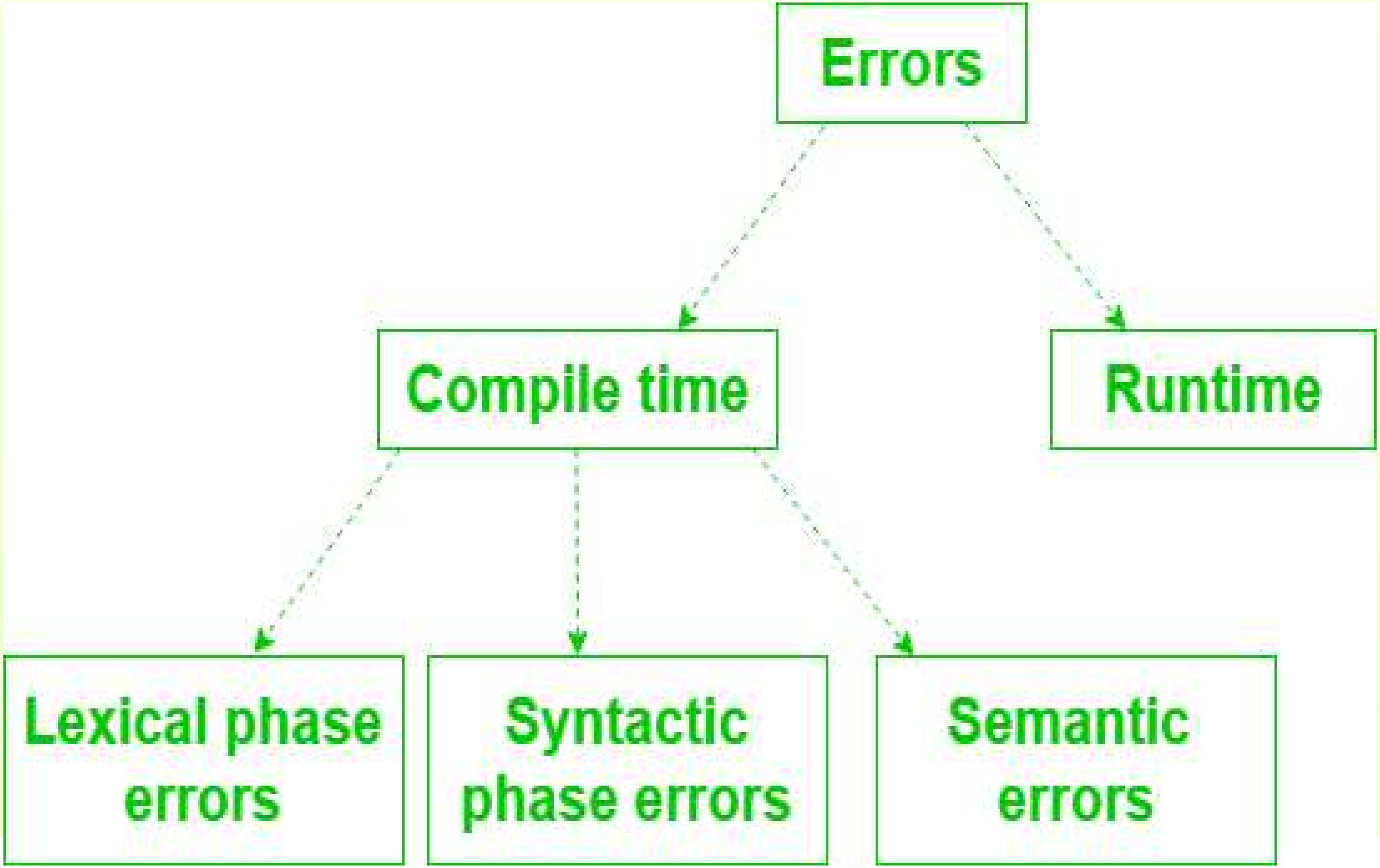
1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

TYPES OF ERRORS

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. **A program may have the following kinds of errors at various stages:**

Lexical error : name of some identifier typed incorrectly
Syntactical error: missing semicolon or unbalanced parenthesis
Semantical error : incompatible value assignment
Logical error: code not reachable, infinite loop
Compile time error.

TYPES OF ERRORS



THANK YOU!!!

My Blog : https://anandgharu.wordpress.com/

Email : gharu.anand@gmail.com