Notes by- Prof. Reshma O. Kapadi
MMCOE, Kaeve Nagar Pune

Unit II

* Macro Processor and Compilers. *

Macro:-

- Macro allows a sequence of source language code to be defined once and then referred many times.
- Each time it is referred by it's name, then the sequence of codes is substituted at that point.

Defining a Macro:-

To define a macro we use following syntax,

1. Macro pseudo code opcode.
2. Macro name.
3. Sequence of statements / instructions.
4. MEND pseudo code opcode.

Syntax:-

~~Macro~~ MACRO mymacro [list of parameters]
(↓ name of macro)

// Body of Macro

MEND.

Example:-

```
MACRO ~~mymacro~~ MYMACRO.
    ADD AREG, ='1'
    SUB BREG, ='2'
    MOVER AREG, ='5'
MEND
```

Calling a Macro:-

A macro is called by writing the macro name with actual parameters.

Syntax:

<macro name> [<list of parameters>]

e.g. ①

MYMACRO X
↑ name of macro
↑ actual parameter.

Example (2)
macro call will be:
MYMACRO.

Macro Expansion:-
Each call to a macro is replaced by its body.
During replacement, actual parameter is used in place of formal parameter.

Example:-
In above example, macro definition is not having any parameter, in that case macro expansion will be.

```
ADD AREG, ='1'
SUB BREG, ='2'
MOVER AREG, ='5'
```

Example (2)
Macro with parameters.

(i) Macro Definition—

```
MACRO  CALCULATE  &ARG
ADD    AREG, &ARG
ADD    BREG, &ARG
MEND.
```

(ii) Macro call:-

```
CALCULATE X
```

(iii) Macro Expansion:-

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

* Assembly Language Program :-

```
MACRO   CALCULATE  &VAR, &REG
        MOVER   &REG, &VAR
        ADD     &REG, &VAR
MEND

START 100
      READ X
      READ Y
      CALCULATE  X, AREG
      PRINT X
      STOP
      X DS 1
      END.
```

In above example, formal parameters are VAR & REG and actual parameters are, X & AREG.

* Types of Parameters :-

There are two types of parameters.

1. Positional Parameter.
2. ~~Actual~~ keyword parameter Parameter.

(1) Positional Parameter :-

A positional parameter is written as &parameter name.

For example,

```
CACLULATE  &VAR1, &VAR2, &VAR3
```

So in above example, &VAR1 & &VAR2, &VAR3 are positional parameter.

During expansion of macro positional parameter values get replaced with actual values based on their position.

example, consider following macro call

```
CALCULATE  X, Y, Z
```

- Value of X will get assigned to VAR1
- Value of Y will get ——"—— VAR2
- Value of Z will ——"—— VAR3.

② Keyword Parameter :-

When we want to assign any default value to parameter at that time we can use keyword parameter.

Syntax:-

<parameter name> = <parameter value>

Example:

```
MACRO
CALCULATE   &VAR1=X, &VAR2=Y, &ANY_REG
                = BREG

MOVER   &ANY_REG , &VAR1
ADD     &ANY-REG , &VAR2
SUB     &ANY_REG , &VAR1
MOVEM   &ANY_REG , &VAR2
MEND.
```

① MACRO call:-

(i) CALCULATE.

Macro Expansion

```
MOVER   BREG, X
ADD     BREG, Y
SUB     BREG, X
MOVEM   BREG, Y
```

(ii) CALCULATE VAR2=B

```
MOVER   BREG, X
ADD     BREG, B
SUB     BREG, X
MOVEM   BREG, B
```

(iii) CALCULATE VAR2 = B, ANY_REG = AREG
VAR1 = A

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

(iv) CALCULATE VAR1 = A, VAR2 = B,
ANY_REG = CREG

```
MOVER  CREG, A
ADD    CREG, B
SUB    CREG, A
MOVEM  CREG, B.
```

(v) CALCULATE ANY_REG = BREG

```
MOVER  BREG, X
ADD    BREG, Y
SUB    BREG, X
MOVEM  BREG, Y.
```

* Macro with Mixed Parameters:-

A macro can have both parameters.

1. Positional
2. Keyword

- When we are defining any macro with both parameters (positional & keyword), then we must follow one Rule.

1. Positional parameters should be written before keyword parameters.

Macro Definition:-

```
MACRO  FIND_VALUE  &VAR1, &VAR2, &USE_REG=
                                AREG
       MOVEM  &USEREG, &VAR1
       ADD    &USE_REG, &VAR2
MEND.
```

Macro call:-

```
FIND_VALUE  X, Y, USE_REG = CREG.
```

Macro Expansion:-

```
MOVEM  CREG, X
ADD    CREG, Y
```

*Other uses of parameters:-
We can use formal parameters in any field of statement inside the body of macro.

1. Label
2. Opcode.
3. Operand.

* Advanced Macro Facilities :-

- Advanced macro facilities allows conditional & re-ordering of the statements of macro expansion.
- Flow of control during macro expansion can be altered using:

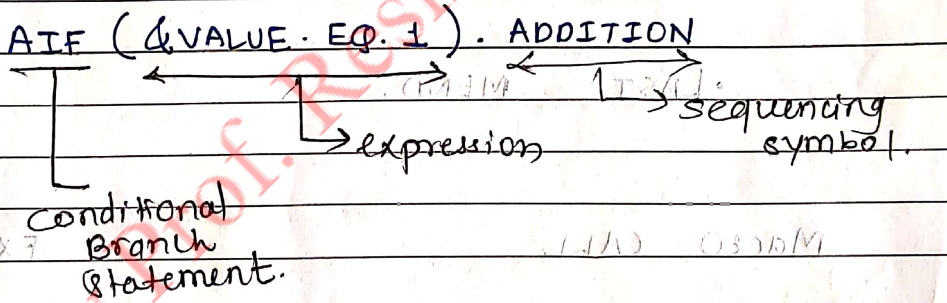
1. conditional branch pseudo-opcode AIF.
2. Unconditional branch pseudo-opcode AGO.

1. AIF :-

- AIF is similar to IF statement.
- The label used for branching is known as sequencing symbol.

Syntax:-

AIF (<expression>).<sequencing symbol>



2. AGO :-

- AGO is unconditional branch pseudo-opcode
- AGO is similar to GOTO statement.

Syntax:

AGO <sequencing symbol>

AGO .LAST

```
MACRO
      CALCULATOR  &VALUE , &ARG1
          AIF  (&VALUE. EQ. 1). ADDITION
          AIF  (&VALUE. EQ. 2). SUBTRACTION.
          AIF  (&VALUE. EQ. 3). MULTIPLICATION
          AGO  .LAST.

.ADDITION     MOVER  BREG, X
              ADD    BREG, &ARG1
              AGO .LAST

.SUBTRACTION  MOVER  BREG, X
              SUB    BREG, &ARG1
              AGO  .LAST

.MULTIPLICATION  MOVER  BREG, X
                 MULT   BREG, &ARG1
                 AGO  .LAST.

.LAST   MEND.
```

Macro CALL

Expanded source

① CALCULATOR 1, Y

② CALCULATOR 2, Z

③ CALCULATOR 3, X

* Design of two pass Macro processor :-

- The macro 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 & substitute arguments.

- The macro processor has two passes.

1) Pass 1.
2) Pass 2.

1) Pass 1 :-

Scan all macro definitions one by one.
For each macro

a) Enter its name in 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.

2) 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 correspondance bet? formal parameters & 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:-
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.

Step 1:- create Macro Name Table.

⇒ Macro Name Table,

MNTP	Name of Macro	Address in MDT
0	INCR	0
1	DECR	5

Step 2:- Create MDT

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

Step3: Prepare argument list array.

1. Macro call:-

INCR N1, N2, REG=CREG

N1
N2
CREG

Argument list array.

2. Expanded code:-

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

② Macro call:-

DECR N1, N2

N1
N2
AREG ← default value.

Argument list array.

Expanded code:-

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

* Macro calls within macros :-
- macro calls within macros is called as Nested macro calls.
- 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.

Consider an example,

```
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.
```

- In above example, 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.

Source line	Expanded sources (level 1)	Expanded source (level 2)

Consider macro call:

COMPUTE1 X, Y, Z

Source line	Expanded source (level 1)	Expanded source (level 2)
~~COMPUTE1 X,Y,Z~~		

COMPUTE1 X, Y, Z	COMPUTE X	MOVER AREG, X
		ADD AREG, ='1'
		MOVEM AREG, X

	COMPUTE Y	MOVER AREG, Y
		ADD AREG, ='1'
		MOVEM AREG, Y

	COMPUTE Z	MOVER AREG, Z
		ADD AREG, ='1'
		MOVEM AREG, Z

* 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 as follows:

```
MACRO DEFINE &VALUE
 MACRO &VALUE &Y
MOVER AREG, &Y
 ADD AREG, ='5'
MOVEM AREG, &Y
  MEND
MEND.
```

MACRO call:-

DEFINE NESTED.

Expansion of DEFINE MACRO:-

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

* Handling of Nested Macro calls:-

There are several methods for handling of nested macro calls.

These methods are:

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

1) several levels of expansion

The macro expansion algorithm can be applied to the first level expanded code to expand these macro calls. & so on, until we obtain a code 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.

In Recursive expansion stack is used.

The local variables are stored onto th stack before making a recursive call.

In recursive expansion, during processin of one macro(outer) the processing of inner macro begins and after the expansion of inner macro finishes, the processing of outer macro may continue.

*Use of stack during expansion:-

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

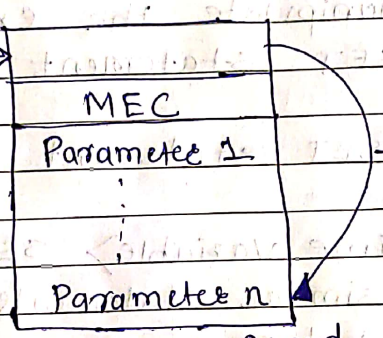
Macro calls are handled in LIFO Manner.

Stack can be used to accommodate the expansion time data structure.

Expansion time data structure is stored in an activation record

The structure of activation record is

Expansion time data structure include:
1. MEC
2. Actual Parameter Table



- 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;

Here, MEC - Macro Expansion Counter.

Expansion Time Variable:-

Expansion Time variables are used in during macro expansions.

These variables are declared as local variables.

Syntax:-

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

- We can manipulate the expansion time variable using SET statement

Syntax for SET statement

<Expansion Time Variable> SET <expression>

* In many macro's, similar statements are generated during expansion, in that case we can use expansion time variable. *

Example,

LCL &A

Here A is a expansion time variable & we can manipulate the value of A using SET statement.

&A SET 0

Here we have assigned value 0 (zero) to expansion time variable A.

Example with macro:-

consider a macro

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

```
MOVEM   AREG, &AREG+2
MOVEM   AREG, &AREG+3
MEND.
```

In above macro, the value 0(zero) is transfer to AREG register and same value 0(zero) is stored in four consecutive locations with the help of MOVEM statement.

If we call the macro, consider following macro call.

CLEAR A then expansion will be

```
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 address A, A+1, A+2 & A+3.

We can achieve above effect by implementing loop for expansion.

Expansion time loop can be written using expansion time variable.

A macro written using expansion time variable (EV).

```
MACRO
CLEAR &ARG, &N
LCL &M          ---- declaration of Expansion time variable.
&M SET 0        ---- assign value 0 to M.
MOVER AREG, ='0'
MORE
MORE MOVEM AREG, &ARG+&M
&M SET &M+1
AIF (&M .NE. &N) .MORE
MEND
```

Call to macro clear.

```
CLEAR A, 3
```

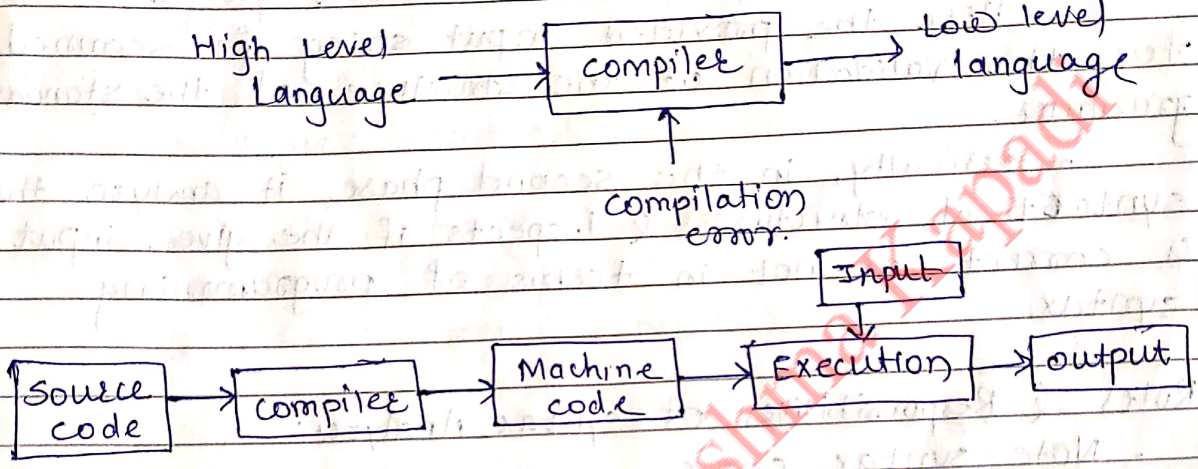
Expansion :-

```
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
```

*Phases of compiler:-

A compiler is a computer program that decodes computer code composed in one programming language into another language.

Compiler helps in translating the source code composed in high level programming language into the machine code.



Phases of compiler:-

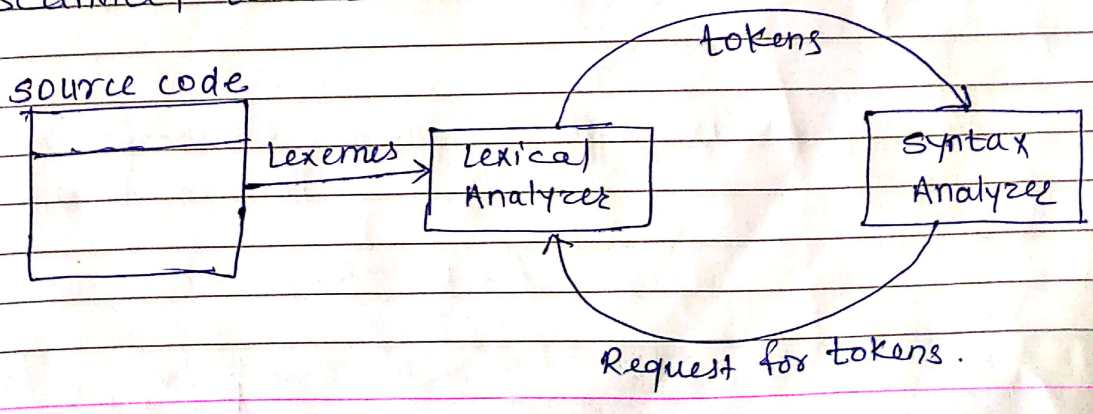
1. Lexical Analysis:-

Lexical analysis or lexical analyzer is the initial stage or phase of the compiler.

This phase scans the source code & transform the input program into series of token.

A token is basically the arrangement of characters that defines a unit of information in the source code.

In computer science, a program that executes the process of lexical analysis is called as scanner, tokenizer or lexer.



* Roles and Responsibilities of Lexical Analyzer:-
  - It helps in identifying the tokens.
  - It is accountable for terminating the comment and white spaces from the source program.

2. Syntax Analysis:-
  - Syntax analysis is the second stage.
  - Here the provided input string is scanned for the validation of the structure of the standard grammer.
  - Basically, in the second phase, it analyse the syntactical structure & inspects if the given input is correct or not in terms of programming syntax.

Roles & Responsibilities of syntax Analyzer:
  - Note syntax error.
  - Helps in building a parse tree.
  - Acquire tokens from the lexical analyzer.

3. Semantic Analysis:-
  In the process of compilation, semantic analysis is the third phase.
  It scans whether the parse tree follows the guidelines of language.

① Consider a statement

$$x = y + z * 30;$$

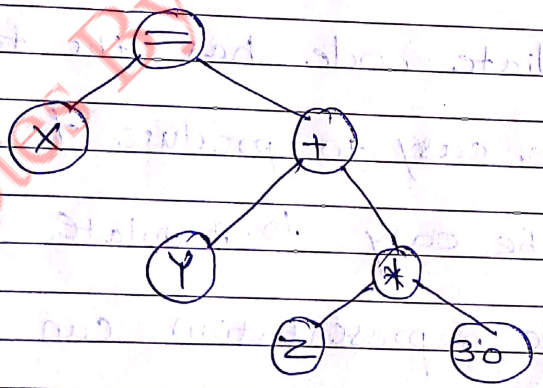
If we perform lexical analysis on above statement, it will generate following tokens.

x is an identifier
= is an terminal symbol
y is an identifier
+ is an terminal symbol
z is an identifier
* is an terminal symbol
30 is a literal
; is a terminal symbol

② Syntax analysis.

A syntax tree of
$x = y + z * 30;$

In syntax tree, operators appear as internal nodes and operand as leaf nodes.

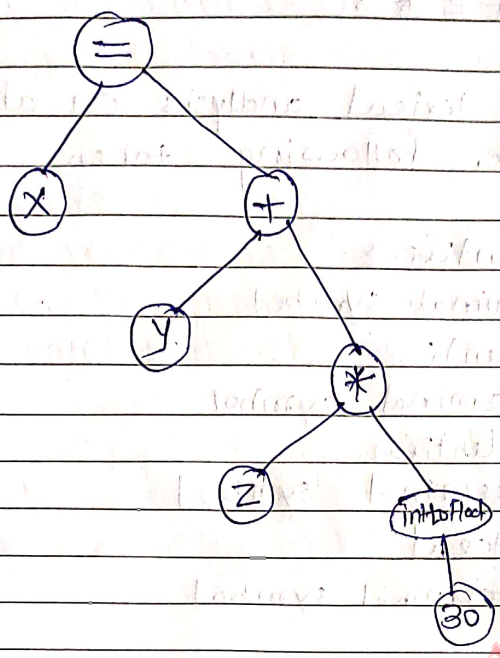


③ In semantic analysis type checking is an important aspect.

In $x = y + z * 30$

if x & y & z are of type float

then literal 30 should be converted into float.



Syntax tree for semantic analysis.

(4) Intermediate code generation.

Before generation of machine code, the compiler creates an intermediate form of the source program.

An intermediate code has the following properties.

1) It should be easy to produce from the Syntax tree.

2) It should be easy to translate into machine code.

The intermediate representation can have a variety of form.

1. Three address code.
2. Quadruple
3. Triple.
4. Postfix Notation
5. Syntax tree
6. DAG representin

1. Three address code:-
Each instruction has maximum of three operands.

x = y + z * 30;

i) z * 30 should be evaluated first & result can be saved in temp 1

temp1 = z * 30.

ii) temp1 must be added to y & store result in temp 2.

temp2 = y + temp1

iii) temp2 should be assigned to x

x = temp2;

Thus, the three address code for x = y + z * 30 is

a) temp 1 = z * 30
b) temp 2 = y + temp 1
c) x = temp2.

2. Qudruple representation:-
It has following form.

Operator	Operand 1	Operand 2	Result
*	z	30	temp 1
+	y	temp 1	temp 2
=	temp2	—	x

⑥ DAG Representation:-

⑤ Code Optimization:-

In this phase compiler tries to improve the intermediate code so that a smaller & faster running machine code can be derived.

For example,

```
temp1 = z * 30
temp2 = y + temp1
x = temp2
```

So, in code optimization,

```
temp1 = z * 30
x = y + temp1
```

④ code Generation:-

```
temp1 = z * 30

MOVER  AREG, Z
MUL    AREG, ='30'
MOVEM  AREG, temp1

X = y + temp1
MOVER  AREG, y
ADD    AREG, temp1
MOVEM  AREG, X
```