Unit -2

Macro Processor and Compiler

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.

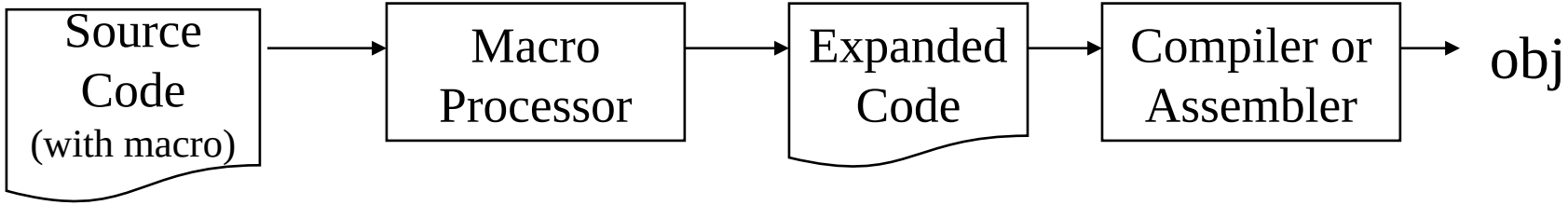
Introduction to Macro Processor

- Macro instructions are considered as the extension of the basic assembler language.
- A macro instruction is convenient for the programmer in terms of notation.
- A macro is a single-line abbreviation used for a group of instructions.
- It allows the programmer to write shorthand version of a program (***module programming***)
- The macro processor replaces each macro call with the corresponding sequence of statements (***expanding***)

Macro Processor

Working of Macro Processor

1. Recognizes macro definitions
2. Saves the macro definition
3. Recognizes macro calls
4. Expands macro calls



Macro Definition

MACRO → Start of Definition

<Macro Name> <List of Parameters [p1, p2, …pn]> →

Macro Name *(with label or argument(s) which is optional)*

→ Sequence to be abbreviated

MEND → End of Definition

Macro Definition (cont..)

Example:

```
MACRO
ADDS    &arg1, &arg2, &arg3
L       1, &arg1
A       1, &arg2
ST      1, &arg3
MEND
```

Macro Call

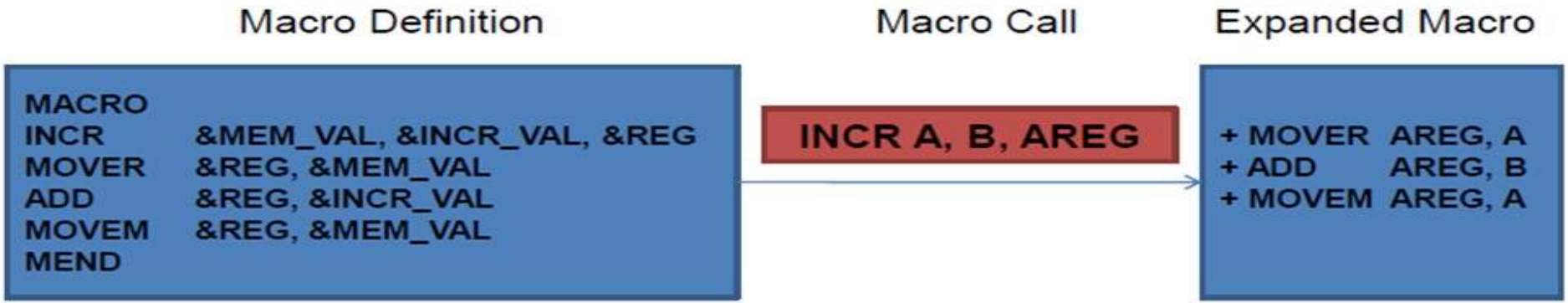
<Macro Name> <List of Actual parameters[a1, a2, …an]>

For Ex.

ADDS DATA1, DATA2, DATA3

Macro Expansion

- Macro calls leads to macro expansion.
- During macro expansion, the macro call is replaced by a sequence of assembly statements.



Macro Expansion/Invocation

//Source Code with Macro definition & macro call

```
MACRO
ADDS    &arg1,&arg2,&arg3
L               1, &arg1
A                       1, &arg2
ST                      1, &arg3
MEND
PROG    START   0
        BALR    15,0
        USING   *,15
        ADDS    DATA1, DATA2, DATA3
        SR      4,4
DATA1   DC      F'3'
DATA2   DC      F'4'
DATA3   DS      1F
        END
```

// Processed code after macro Expansion

```
PROG    START   0
        BALR    15,0
        USING   *,15
L       1, DATA1
A       1, DATA2
ST      1,DATA3
        SR      4,4
DATA1   DC      F'3'
DATA2   DC      F'4'
DATA3   DS      1F
        END
```

Macro Expansion

```
MACRO
ADDS   &arg1,&arg2,&arg3
L                1, &arg1
A                1, &arg2
ST               1, &arg3
MEND
```

Macro Expansion (cont..)

Source program with Macro Definition & Macro call

```
MACRO
ADDS    &arg1,&arg2,&arg3
L       1, &arg1
A               1, &arg2
ST              1, &arg3
MEND
PROG    START   0
        BALR    15,0
        USING   *,15
        ADDS    DATA1, DATA2, DATA3
        SR      4,4
        ADDS    D4, D5, D6
DATA1   DC      F'3'
DATA2   DC      F'4'
DATA3   DS      1F
D4      DC      F'1'
D5      DC      F'2'
D6      DS      1F
        END
```

Source program after macro expansion

```
PROG    START   0
        BALR    15,0
        USING   *,15
        L       1, DATA1
        A       1, DATA2
        ST      1, DATA3
        SR      4, 4
        L       1, D4
        A       1, D5
        ST      1, D6
DATA1   DC      F'3'
DATA2   DC      F'4'
DATA3   DS      1F
D4      DC      F'1'
D5      DC      F'2'
D6      DS      1F
        END
```

1st Macro call

2nd Macro call

Macro expansion

- During macro expansion, the macro name statement in the program is replaced by the sequence of assembly statements.

START	100	
A	DS	1
B	DS	1
INCR	A, B, AREG	
PRINT	A	
STOP		
END		

Assembly Program

	START	100	
	A	DS	1
	B	DS	1
+	MOVER	AREG	A
+	ADD	AREG	B
+	MOVEM	AREG	A
	PRINT	A	
	STOP		
	END		

Extended Assembly Program

Macro

MACRO		
INCR	&MEM_VAL, &INC_VAL, &REG	
MOVER	&REG	&MEM_VAL
ADD	&REG	&INC_VAL
MOVEM	&REG	&MEM_VAL
MEND		

Parameter Types/Substitution

- Positional / Formal Argument
- Keyword Argument
- Default Argument
- Mixed Argument

Types of formal parameters

- Two types of formal parameters are:

1. Positional parameters: Order can not be changed in macro call.

   Example:

   Prototype statement: INCR &MEM_VAL, &INC_VAL, &REG

   Macro call: INCR A, B, AREG

2. Keyword parameters: Order can be changed in macro call.

   Example:

   Prototype statement: INCR &MEM_VAL=, &INC_VAL=, &REG=

   Macro call: INCR INCR_VAL=B, REG=AREG, MEM_VAL=A

Example: Positional parameter

Macro	MACRO		
	INCR	&MEM_VAL, &INC_VAL, &REG	
	MOVER	&REG	&MEM_VAL
	ADD	&REG	&INC_VAL
	MOVEM	&REG	&MEM_VAL
	MEND		

Positional parameters

START	100	
A	DS	1
B	DS	1
INCR	A, B, AREG	
PRINT	A	
STOP		
END		

Assembly Program

	START	100	
	A	DS	1
	B	DS	1
+	MOVER	AREG	A
+	ADD	AREG	B
+	MOVEM	AREG	A
	PRINT	A	
	STOP		
	END		

Extended Assembly Program

Keyword parameter

- Syntax:

  &<parameter name>=

- The <actual parameter specification> is written as <formal parameter name> = <ordinary string>.
- The value of a formal parameter is determined by the rule of keyword association as follows:
  - Find the actual parameter specification which has the form XYZ= <ordinary string>.
  - If the <ordinary string> in the specification is some string ABC, the value of formal parameter XYZ would be ABC.

Example: Keyword parameters

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

Keyword parameters

During a macro call, a keyword parameter is specified by its name.

INCR_M	MEM_VAL=A, INCR_VAL=B, REG=AREG

OR

INCR_M	INCR_VAL=B, REG=AREG, MEM_VAL=A

The order can be changed.

Default Argument

```
//MACRO DEFINITION

MACRO
ADDS    &arg1=, &arg2=, &arg3=4
L               1, &arg1
A               1, &arg2
ST              1, &arg3
MEND


/MACRO CALL

ADDS    &arg1=D1, &arg2= D2
……

ADDS    &arg2= D2, &arg1=D1        //it overrides  default value

ADDS    &arg2= D2, &arg1=D1, &arg3=100
```

Formal Parameter	Value
&arg1	D1
&arg2	D2
&arg3	100

Macro with Mixed Parameter Lists

Macro can use all three types of parameters together

Example:-

```
//MACRO DEFINITION

MACRO
ADDS    &arg1, &arg2=45, &arg3=
L                   1, &arg1
A                   1, &arg2
ST                  1,&arg3
MEND

//MACRO CALL

ADDS    D1, &arg3= D3
……
ADDS    D1, &arg3=D6, &arg2=400
```

Macro vs. Subroutine

Macro:

- Every macro call is replaced by its definition.
- After expansion of program, length increases. So, more memory is required.
- Processing time is less as compared to subroutine processing, because there is no context switching during macro processing.

Subroutine

- Every Subroutine (function) call transfers control to the first instruction of subroutine which is called.
- After processing subroutine call, the program size remains same.
- Processing time is increased due to context switching.

Advanced Macro Facilities

- Macro Calls within Macros (Nested Macro Calls)
- Macro instruction defining Macros
- Conditional Macro Expansion
- Expansion time variables
- Expansion time Loops

Macro Calls within Macros (Nested Macro Calls)

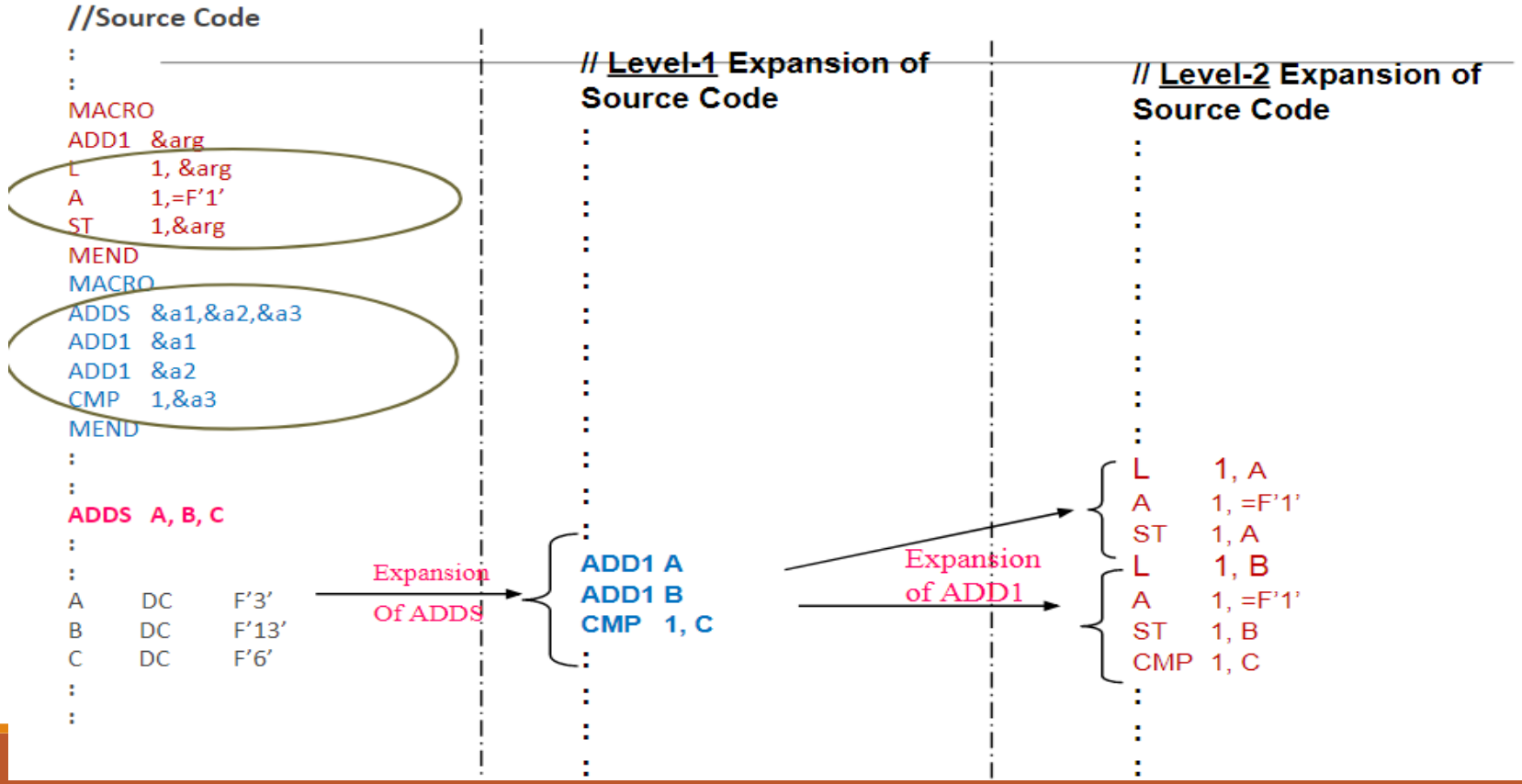
Macro definition of one macro may generate call to another macro is called as nested macro calls.

Example.

```
MACRO
ADD1    &arg
L               1, &arg
A               1, =F'1'
ST              1, &arg
MEND
MACRO
ADDS            &arg1, &arg2 ,&arg3
ADD1            &arg1
ADD1            &arg2
MEND
```

Call to macro

Macro Calls within Macros (Nested Macro Calls) (cont..)



Macro Instruction Defining Macros

Macro definition may defines another macro

Ex.

```
MACRO
DEFINE    &SUB
MACRO
&SUB      &Y
CNOP      0,4
BAL                 1,*+8
DC                  A(&Y)
L                   15,V(&SUB)
BALR      14,15
MEND
MEND
```

Inner Macro

Outer Macro

Expanded Macro

```
CNOP      0,4
BAL                 1,*+8
DC                  A(AR)
L
15,V(COS)
BALR      14,15
```

Macro calls

- DEFINE COS
- COS AR

CONDITIONAL MACRO EXPANSION

- This allows conditional selection of machine instructions that appear in expansion of macro call.
- The sequence of macro expansion can be reordered or change based on some conditions.
- There are 2 Important macro processor pseudo op. they are
  - AIF
  - AGO

Conditional Macro Expansion

- Expansion based on condition.
- 2 macro pseudo ops for conditional macro expansion are
  - AIF
  - AGO
- Syntax of flow of control statements:
  - AIF(< condition>) Conditional Label
    - Ex : AIF(&arg1 = &arg2). FINI
  - AGO Label
    - Ex: AGO .OVER

AIF , .FINI, AGO

.FINI :- it is macro label and don't appear in output of macro processor.

AIF(&count EQ 1).FINI:- The pseuodo op directs the macro processor to skip the statement labeled .FINI, if "&count" value is 1; otherwise , the macro processor is to continue with the statement following AIF pseuodo op.

AGO:- it is unconditional branch pseudo op like "go to".

AIF

- It is a conditional branching pseudo op the format of AIF is

AIF<expression>.<sequencing label>

- Where expression is a relational expression, it involves strings, numbers etc.
- If the expression evaluates to true them the control transferred to the statements containing the sequencing label
- Otherwise, the control transferred to the next statement followed by AIF
- AIF statement does not appears in the expanded source code

AGO

- It is an unconditional branching pseudo op the format of AGO is

  AGO.<Sequencing label>

- It is conditional transfer control to the statement containing sequencing label
- Each and every label must be starting with a .(dot) operator.
- AGO statement does not appear in the expanded source code.

  Ex:

```
Loop1 A 1,data1
A 2, data2
A 3, data3
Loop2 A 1,data3
A 2, data2
Loop3 A 1, data1
```

FINI

- FINI is a macro label.
- Labels starting with a period(.) such as .FINI, are macro labels and don't appear in the output of the macro processor.

AIF (& COUNT EQ 1).FINI

- It directs the macro processor to skip to statement labeled FINI if the parameter corresponding to & COUNT is 1,otherwise the macroprocessor continues with the statement following the AIF pseudo-op

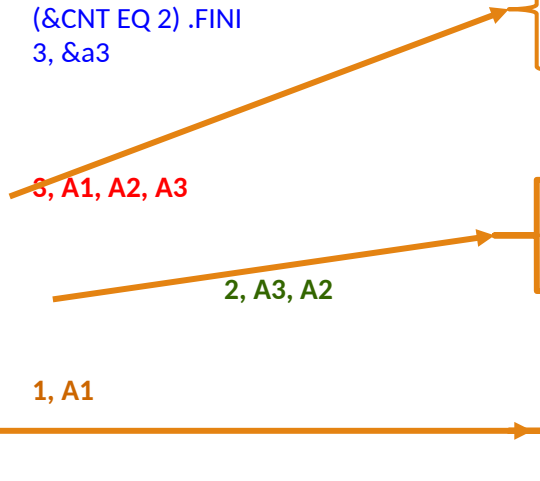
Need of Conditional Macro Expansion

Lets consider example:

```
                :
                :
LOOP1    A      1, A1
         A      2, A2
         A      3, A3
                :
                :
LOOP2    A      1, A3
         A      2, A2
                :
                :
LOOP3    A      1, A1
                :
                :
A1       DC     F'5'
A2       DC     F'15'
A3       DC     F'10'
```

Conditional Macro Expansion

```
        MACRO
&a0     VARY    &CNT, &a1, &a2, &a3
&a0     A       1, &a1
        AIF     (&CNT EQ 1) .FINI
        A       2,& a2
        AIF     (&CNT EQ 2) .FINI
        A       3, &a3
.FINI   MEND
        :
        :
LOOP1   VARY    3, A1, A2, A3
        :
        :
LOOP2   VARY    2, A3, A2
        :
        :
LOOP3   VARY    1, A1
        :
        :
A1      DC      F'5'
A2      DC      F'15'
A3      DC      F'10'
```



// Expanded Source code

```
        :
LOOP1   A       1, A1
        A       2, A2
        A       3, A3
        :
        :
LOOP2   A       1, A1
        A       2, A2
        :
        :
LOOP3   A       1, A1
        :
        :
A1      DC      F'5'
A2      DC      F'15'
A3      DC      F'10'
```

Comparison of Macro Processors Design

- Single pass
  - every macro must be defined before it is called
  - one-pass processor can alternate between macro definition and macro expansion
  - nested macro definitions may be allowed but nested calls are not
- Two pass algorithm
  - Pass1: Recognize macro definitions
  - Pass2: Recognize macro calls
  - nested macro definitions are not allowed

Design of Two pass macro Processor

Pass-1 :

- Recognizes Macro Definition
- Stores Macro Instruction

Pass-2 :

- Recognizes Macro Calls
- Expands calls and substitute actual arguments

Specifications of Data Structures

Pass-1 Data structures/ programs :

- Source program with macro definitions and macro calls
- Output file without macro definitions & with macro calls
- Macro Definition Table (MDT)
- Macro Name Table (MNT)
- Argument List Array (ALA)
- Macro Definition Table Counter (MDTC) : Integer Variable
- Macro Name Table Counter (MNTC): Integer Variable

Pass-2 Data structures/programs :

- Input file without macro definitions & with macro calls
- Expanded output file without macro definitions & macro calls (Free from Macro)
- Refers Macro Definition Table (MDT) created by Pass-1
- Refers Macro Name Table (MNT) created by Pass-1
- Argument List Array (ALA) to map formal parameters with actual
- Macro Definition Table Pointer (MDTP): Integer Variable

Two Pass Macro Processor

Data structures used in two pass macro processor:-

1. Macro Definition Table(MDT) – Store definition of macro
2. Macro Name Table (MNT)– Store name of macro along with address of macro definition.
3. Argument List Array (ALA) – used to substitute index markers for dummy arguments before storing a macro definition.
4. Macro definition table counter (MDTC) – used to indicate the next available entry in the MDT
5. Macro name table counter (MNTC) – used to indicate the next available entry in MNT.
6. Macro definition table pointer (MDTP) – used to indicate the next line of text to be used during macro expansion.

Formats of Data Structures

- Macro Name Table (MNT)

MNTC	Macro Name	MDTC
1	ADDS	1
2		
.		
.		

Formats of Data Structures (Cont..)

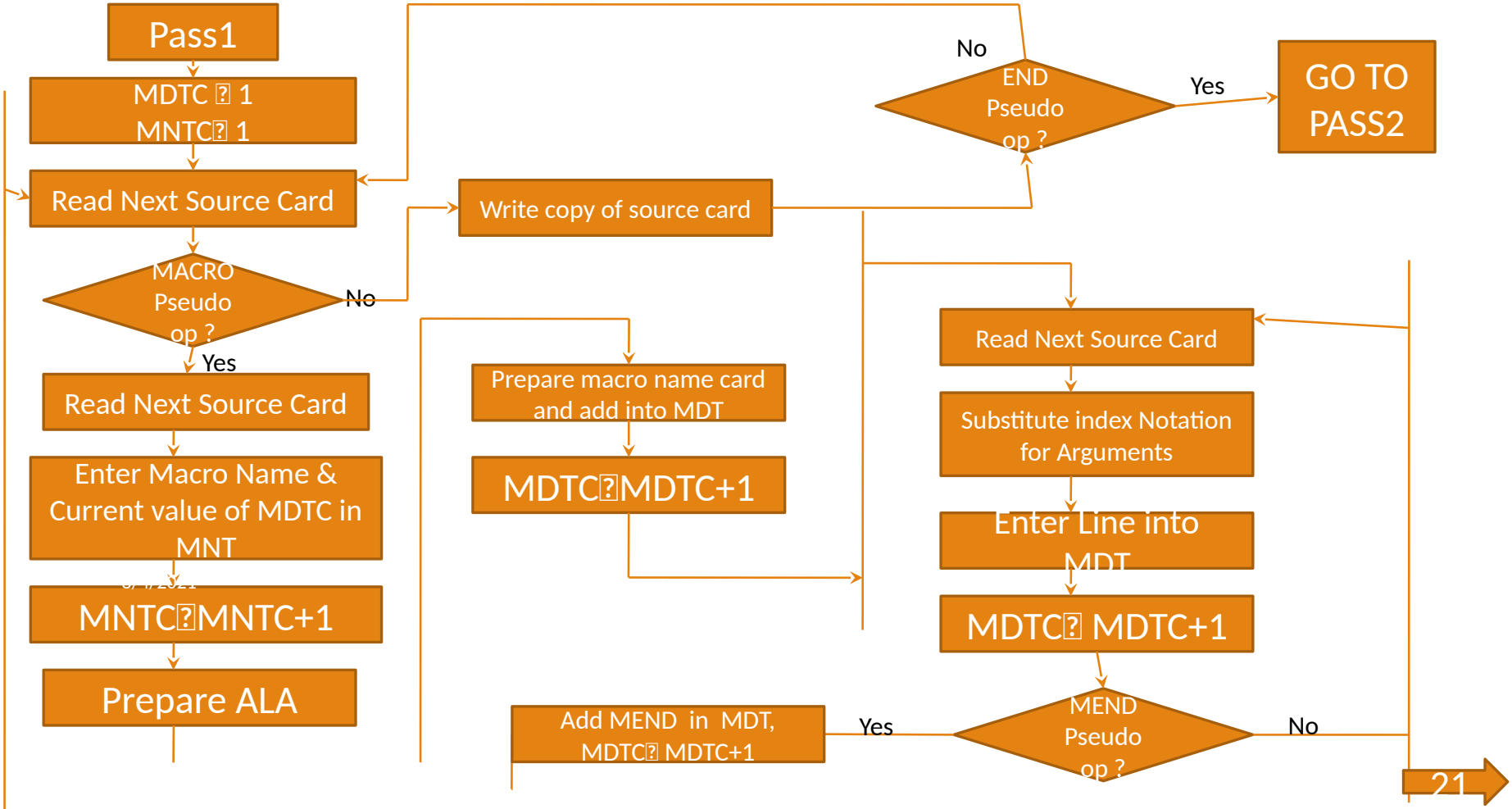
- **Macro Definition Table (MDT)**

MDTC	Macro Definition Instruction Entry (80 bytes per entry)
1	& Lab ADDS &A1, &A2
2	#0 A 1, #1
3	A 1, #2
4	MEND
5	
:	
:	
:	

Formats of Data Structures (Cont..)

- **Arguments List Array (ALA)**

Index	Formal Arguments	Actual Arguments
0	&Lab	-
1	&A1	-
2	&A2	-
3		
:		
:		


Pass1
MDTC ? 1
MNTC? 1
Read Next Source Card
MACRO Pseudo op ?
No
Yes
Read Next Source Card
Enter Macro Name & Current value of MDTC in MNT
MNTC?MNTC+1
Prepare ALA
Write copy of source card
Prepare macro name card and add into MDT
MDTC?MDTC+1
END Pseudo op ?
No
Yes
GO TO PASS2
Read Next Source Card
Substitute index Notation for Arguments
Enter Line into MDT
MDTC? MDTC+1
MEND Pseudo op ?
Yes
No
Add MEND in MDT, MDTC? MDTC+1
21

Source Code :

```
MAC START 100
MACRO
&A0     ADD1    &A1, &A2, &A3
&A0     L       1, &A1
        L       2, &A2
        AR      1, 2
        MUL     1, &A3
        ST      N, 1
        MEND
        MACRO
        SUB     &P1, &P2
        L       1, &P1
        S       1, &P2
        ST      2, 1
        MEND
TOTAL   EQU     5
        L       1,D1
        SR      2,2
        A       1,=F'5'
        ADD1    LOOP1, D1, D2, D3
        ST      2, 1
        AR      TOTAL, 2
        SUB     X, Y
        BR      14
D1      DC      F'3'
D2      DC      F'45'
D3      DC      F'21'
X       DC      F'10'
Y       DC      F'20'
        END
```

Intermediate Code after pass1:

```
MAC     START   100
TOTAL   EQU     5
        L       1, D1
        SR      2, 2
        A       1, =F'5'
        ADD1    LOOP1, D1, D2, D3
        ST      2, 1
        AR      TOTAL, 2
        SUB     X, Y
        BR      14
D1      DC      F'3'
D2      DC      F'45'
D3      DC      F'21'
X       DC      F'10'
Y       DC      F'20'
        END
```

47

35

Macro Name Table :

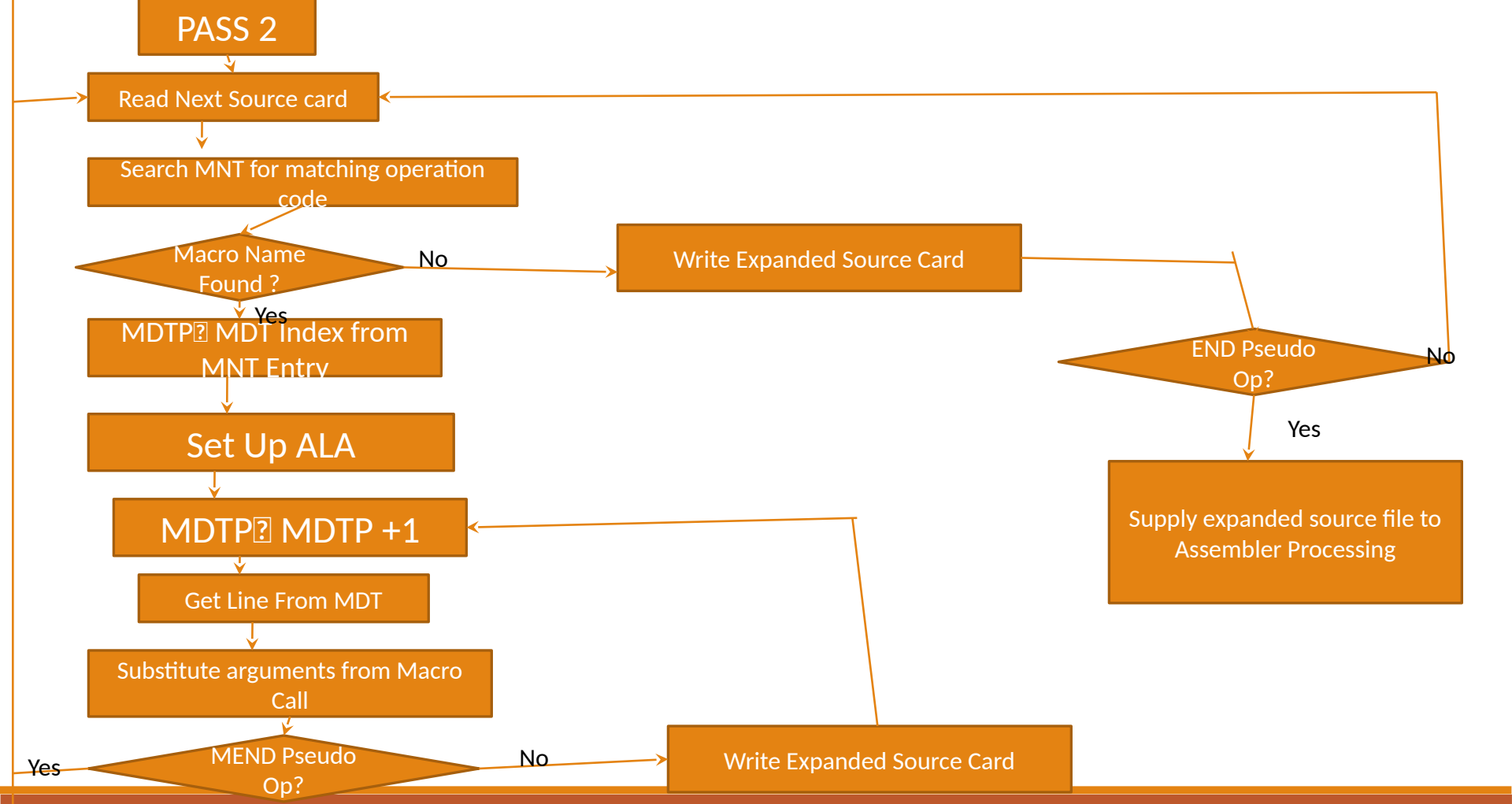
MNTC	Macro Name	MDTC
1		

Argument List Array :

Index	Formal Args	Actual Args
1		
2		
3		
4		
5		
6		

Macro Definition Table :

MDTC	Macro Card
1	


PASS 2
Read Next Source card
Search MNT for matching operation code
Macro Name Found ?
No
Write Expanded Source Card
Yes
MDTP MDT Index from MNT Entry
END Pseudo Op?
No
Yes
Set Up ALA
Supply expanded source file to Assembler Processing
MDTP MDTP +1
Get Line From MDT
Substitute arguments from Macro Call
MEND Pseudo Op?
Yes
No
Write Expanded Source Card

ALA

Index	Formal	Actual
1	&A0	LOOP1
2	&A1	D1
3	&A2	D2
4	&A3	D3
5	&P1	X
6	&P2	Y

, 5, 6, 7 for ADD1
11, 12, 13 for SUB

Intermediate Code after pass1 (input file for pass 2):

```
MAC     START   100
TOTAL   EQU     5
        L       1, D1
        SR      2, 2
        A       1, =F'5'
        ADD1    LOOP1, D1, D2, D3
        ST      2, 1
        AR      TOTAL, 2
        SUB     X, Y
        BR      14
D1      DC      F'3'
D2      DC      F'45'
D3      DC      F'21'
X       DC      F'10'
Y       DC      F'20'
        END
```

Expanded Source Code :

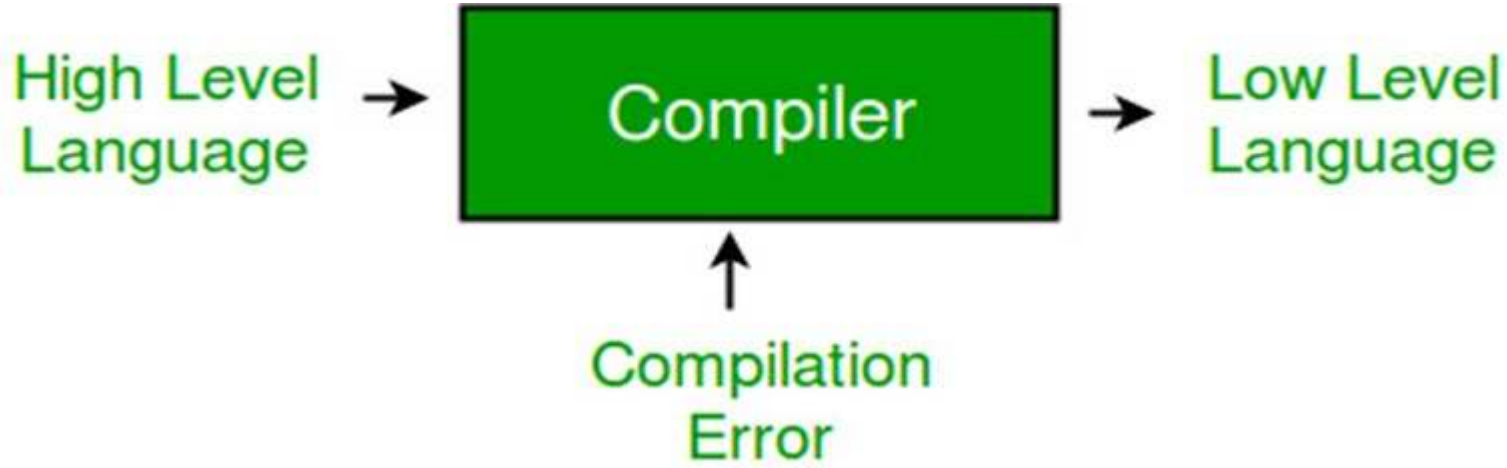
```
MAC     START   100
TOTAL   EQU     5
        L       1, D1
        SR      2, 2
        A       1, =F'5'
LOOP1   L       1, D1
        L       2, D2
        AR      1, 2
        MUL     1, D3
        ST      N, 1
        ST      2, 1
        AR      TOTAL, 2
        L       1, X
        S       1, Y
        ST      2, 1

        BR      14
D1      DC      F'3'
D2      DC      F'45'
D3      DC      F'21'
X       DC      F'10'
Y       DC      F'20'
        END
```

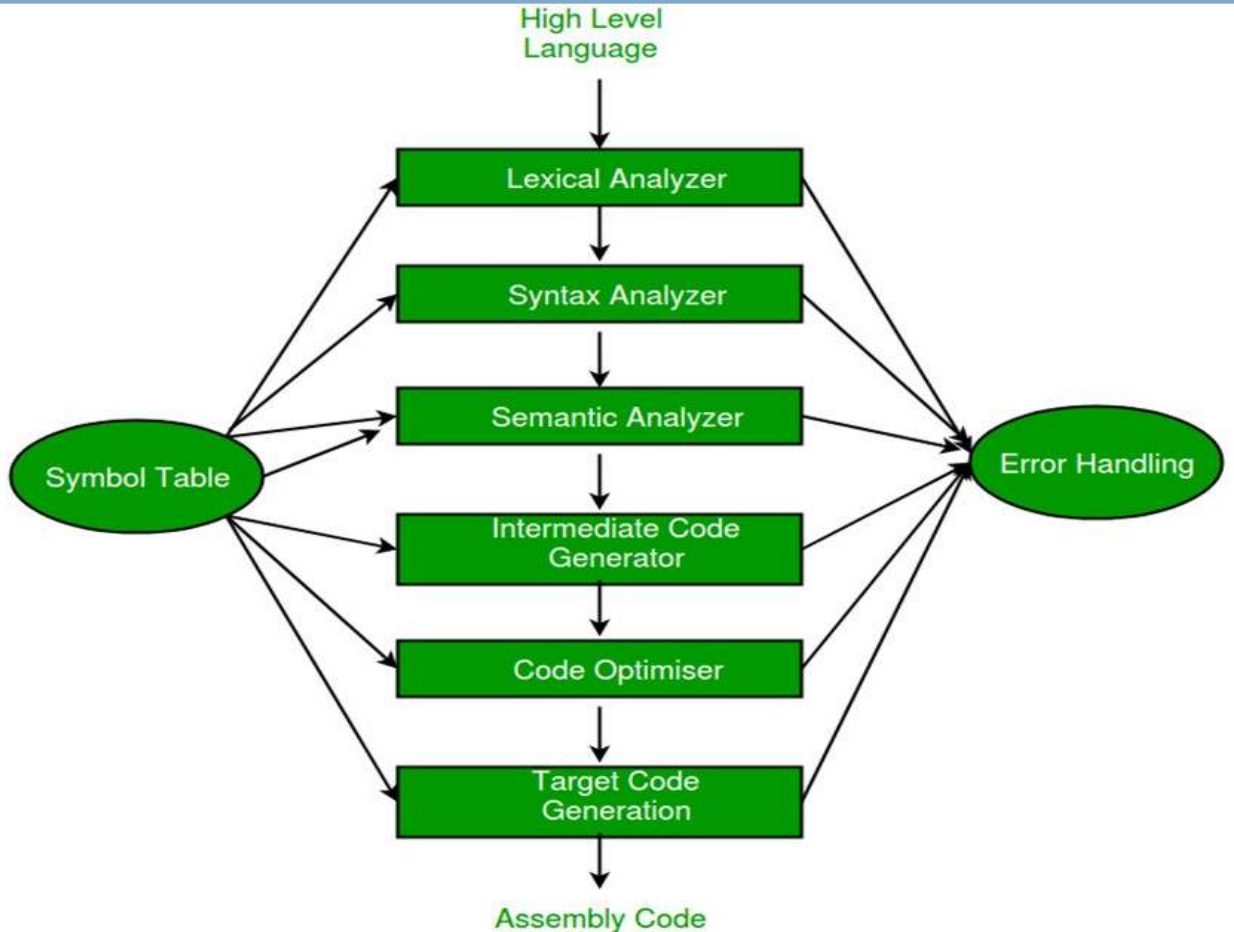
Click on link for Example of Macro :View

What is Compiler?

- **Compiler** is a software which converts a program written in high level language (Source Language) to low level language (Object/Target/Machine Language).



Let's have a look at, what happens inside Black Box



- Structure of compiler has of two parts:

1. Analysis phase(front end)
2. Sysnthesis Phase(back end)

- Front-end constitutes of the Lexical analyzer, semantic analyzer, syntax analyzer and intermediate code generator. And the rest are assembled to form the back end

- **Lexical Analyzer –** It reads the program and converts it into tokens. It converts a stream of lexemes into a stream of tokens. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes white-spaces and comments.
- Example: X:=z+y
- 5 token x,=,z,+,y after this stage id1 assign id2 binop id3

2.Syntax Analyzer

- It is sometimes called as parser. It constructs the parse tree.
- It takes all the tokens one by one and uses Context Free Grammar to construct the parse tree.
- *Why Grammar ?*
  The rules of programming can be entirely represented in some few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
- Syntax error can be detected at this level if the input is not in accordance with the grammar

Grammar is a set of production rules that defines the syntax of a language

Terminals: A set of terminals which we also refer to as tokens, these set of tokens forms strings.
Non-Terminals: CFG has a set of non-terminals (variables). These variables represent the set of strings. Further, they contribute to the formation of a language generated by grammar.

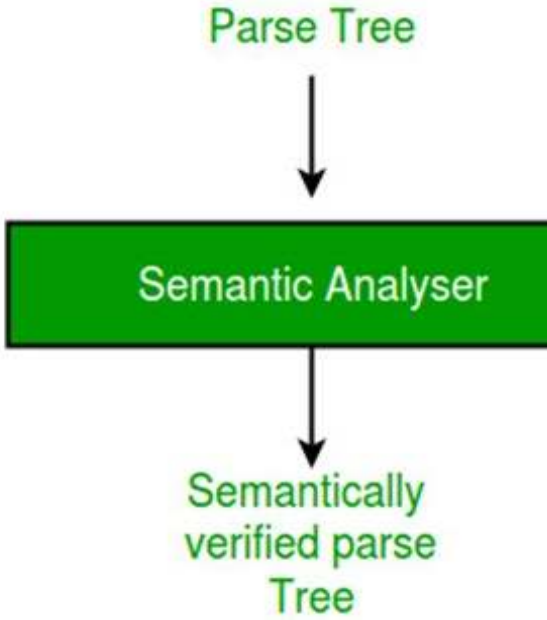
Production: A grammar has a set of production rules. These rules specify how to combine terminals and non-terminals to form a string

Example of Context-free Grammar:
Let us consider a grammar production

stmt -> if (expr) stmt else stmt

3.Semantic Analyzer

- **Semantic Analyzer** – It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree.
- Semantic deals with Type checking constraint with the help of rules.



4.Intermediate Code Generator

- It act as bridge between the analysis phase and synthesis phase of compilation process.
- It generates intermediate code, that is a form which can be readily executed by machine We have many popular intermediate codes.
- Example – ***Three address code(TAC)***, Quadruples, triples,Postfix etc.
- Intermediate code is converted to machine language using the last two phases which are platform dependent. Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

Code Optimizer

- Optimization: Compilers can apply various optimization techniques to the code, such as loop unrolling, dead code elimination, and constant propagation, which can significantly improve the performance of the generated machine code.
- It transforms the code so that it consumes fewer resources and produces more speed.
- The meaning of the code being transformed is not altered.
- Optimization can be categorized into two types: machine-dependent and machine-independent.

6.Target Code Generator

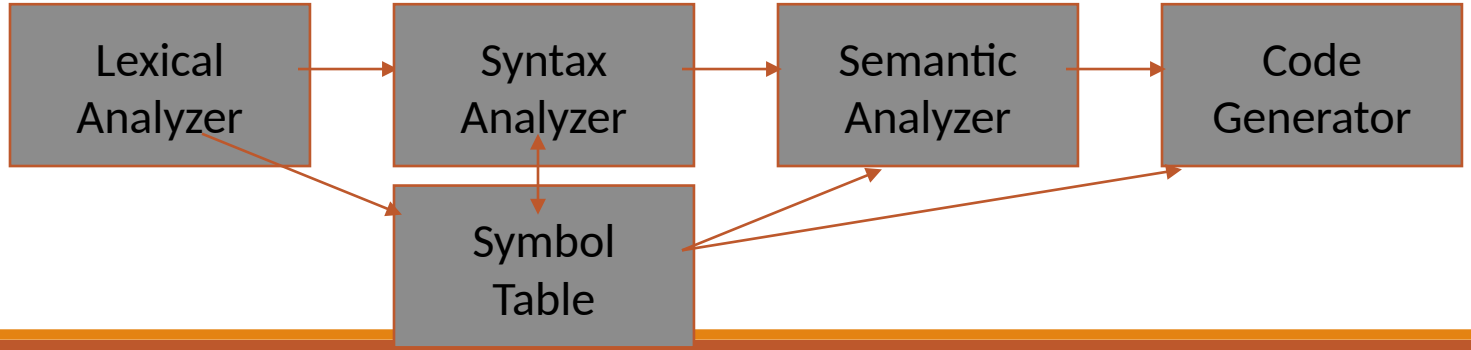
- **Target Code Generator –** The main purpose of Target Code generator is to write a code that the machine can understand.
- The output is dependent on the type of assembler.
- This is the final stage of compilation.

Symbol Table

- Compiler uses symbol table to keep track of scope and binding information about names.
- Symbol table is searched every time when a name is encountered in source text.
- Changes occur in symbol table if a new name or new info about existing name is discovered.
- Compiler should grow S.T. dynamically at compile time or the symbol table can be kept fixed.

The Symbol Table

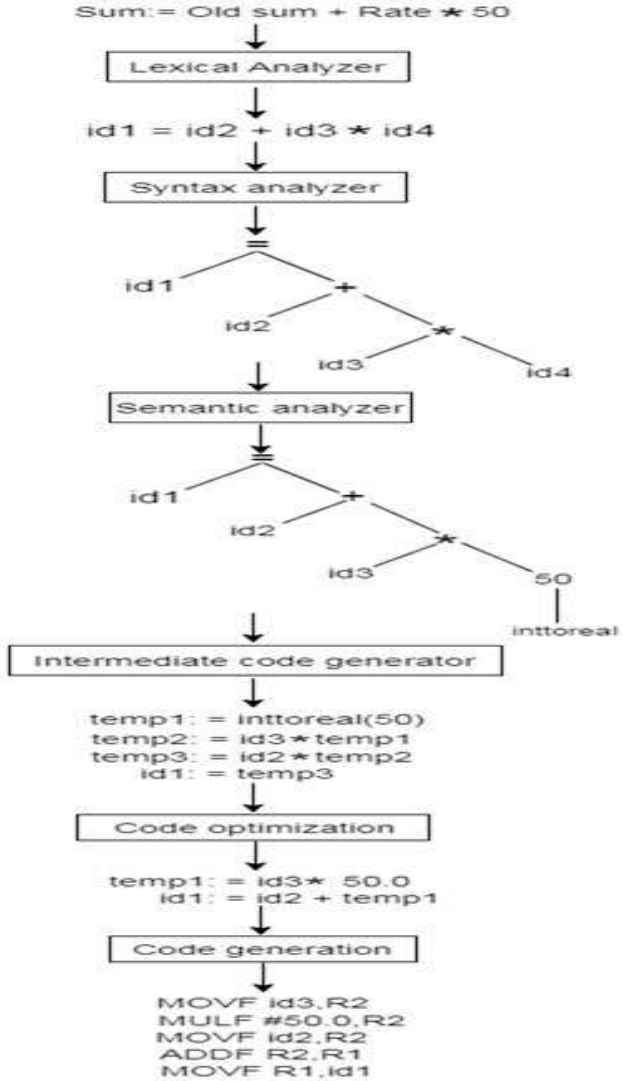
- When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers.
- When the symbol is first encountered by the lexer, we do not yet know the scope.
- That is determined later by the parser.
- This information will be used later by the semantic analyzer and the code generator.



Symbol Table Entries

We will store the following information about identifiers.

- The name (as a string).
- The data type and value.
- The block level.
- Its scope (global, local, or parameter).
- Its offset from the base pointer (for local variables and parameters only).



- https://www.javatpoint.com/compiler-phases

The Structure of a Compiler

```
position := initial + rate * 60
```

Scanner
[Lexical Analyzer]

Tokens

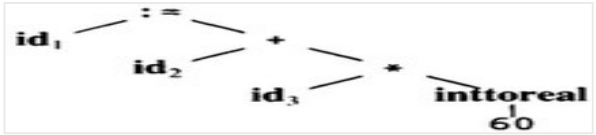
$id_1 := id_2 + id_3 * 60$

Parser
[Syntax Analyzer]

Parse tree

Semantic Process
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes



SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...
4		

Code Generator
[Intermediate Code Generator]

Non-optimized Intermediate Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code Optimizer

Optimized Intermediate Code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

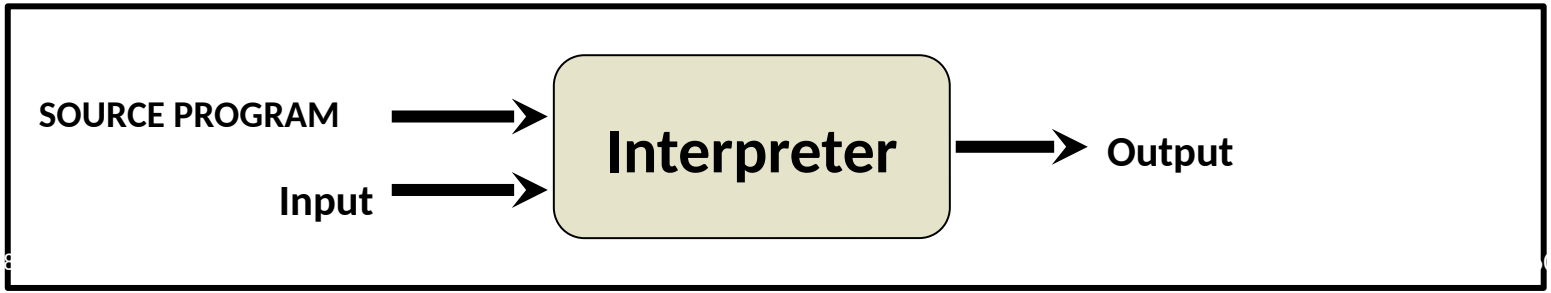
Code Generator

Target machine code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Interpreter

It is a translator (language processor) which directly executes operations specified in source program on input supplied by user.



Interpreter vs Compiler

Interpreter	Compiler
Translates program by one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence less memory are required.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Case Study- GNU m4 Macro Processor

- GNU M4 is an implementation of the traditional Unix macro processor
- m4 is a macro processor, it copies its input to the output, expanding macros as it goes.
- Macros are either built-in or user-defined, and can take any number of arguments.

https://www.gnu.org/software/m4/

Case Study- GNU m4 Macro Processor

1. m4 is a **general-purpose macro processor included** in most Unix-like operating systems,

2. Kernighan and Ritchie developed m4 in 1977,

Case Study- GNU m4 Macro Processor

The format of the m4 command is:

m4 [*option*...] [*macro-definitions*...] [*input-file*...]

m4 --def foo --debug a is equivalent to

m4 --define=foo --debug= -- ./a

Example of Macro in C language

```
#include <stdio.h>
#define NAME "TechOnTheNet.com"
#define AGE 10
int main()
{
   printf("%s is over %d years old.\n", NAME, AGE);
   return 0;
}
```

References

1) John Donovan, “Systems Programming”, McGraw Hill, ISBN 978-0--07-460482-3

2) Dhamdhere D., "Systems Programming and Operating Systems", McGraw Hill, ISBN 0 - 07 - 463579 – 4

3) https://www.gnu.org/software/m4/manual/m4.html#Syntax

Ebooks:

- https://www.elsevier.com/books/systems-programming/anthony/978-0-12-800729-7
- https://www.kobo.com/us/en/ebook/linux-system-programming-1
- https://www.e-booksdirectory.com/details.php?ebook=9907

MOOCs Courses Links:

- nptel video lecture link: https://nptel.ac.in/courses/106/105/106105214/
- https://onlinecourses.nptel.ac.in/noc19_cs50/preview
- https://www.udemy.com/course/system-programming/

Thank You !!