Times asked: 6 times
5 times
4 times
3 times
2 times
1 time

# indicates 5-mark question

SPCC Question Bank

1. Introduction to System Software

1. Differentiate between System Software and Application software. #
2. Compare Compiler and Interpreter. #

2. Assemblers

1. Explain forward reference problem with suitable example. #
2. Explain with flowchart design of two pass assembler.
3. Draw and explain the flowchart of Pass 1 of two pass assembler with suitable example.
4. State and explain the types of assembly language statements with examples.
5. Consider the following assembly program:

```
START   501
        A DS 1
        B DS 1
        C DS 1
READ A
READ B
MOVER       AREG, A
ADD         AREG, B
MOVEM       AREG, C
PRINT C
END
```

- Generate Pass-1 and Pass-2 and also show the content of database table involved in it.

3. Macros and Macro Processor

1. Explain the working of a Single pass macro processor with neat flowchart.
2. Explain and draw a neat flowchart of two pass macro processor.
3. Explain advanced macro facilities with suitable examples.
4. Write a short note on: Parameterized macros. #
5. With reference to MACRO, explain the following tables with suitable example:
   I) MNT ii) MDT iii) ALA
6. Explain conditional macro with suitable example. #

7. Construct the necessary data structure after compiling the following code by Pass-1 of two-pass macro processor:

```
1.  MACRO
2.  COMPUTE   &x, &a, &p
3.  MOVER     &a, &x
4.  MULT      &a, ='4'
5.  MOVEM     &a, &p
6.  MEND
7.  MACRO     &g, &k, &r
8.  MOVER     &r, &k
9.  SUB       &r, ='4'
10. MEND
```

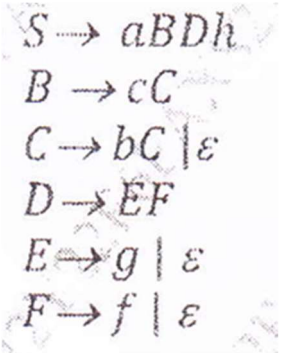
8. Write a short note on: Macro facilities. #
9. Explain Macro and Macro expansion with example. #
10. Explain different features of macros with suitable example.

4. Loaders and Linkers

1. What are the functions of a Loader. #
2. Explain design of Direct Linking Loader with suitable example. Discuss the databases used.
3. Explain absolute loader. State its advantages and disadvantages. #
4. Explain Dynamic Linking Loader in detail.
5. What is relocation and linking concept in Loaders. #

5. Compilers: Analysis Phase

1. Explain the different phases of compiler with suitable example; or a given statement. (Previously asked statements)
   i. a = a * b – 5 * 3 / c
   ii. P = Q + R – S * 3
   iii. a=b*c+10
2. Construct LL(1) parsing table for the given grammar and state whether the given grammar is LL(1) or not. (Previously asked questions)


$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \mid \varepsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \varepsilon$
$F \rightarrow f \mid \varepsilon$

S → iCtSE | a
E → eS | ε
C → b

S→ Ad
A→ aB | BC
B→ b
C→ e | ε

S→ 1AB | ε
A→ 1AC | 0C
B→ 0S
C→ 1

S→ AB | gDa
A→ ab | c
B→ dC
C→ gC | g
D→ fD | g

3. Write a short note on: Syntax directed translation. #
4. Construct operator precedence parser for the grammar:
   E → E+E | E*E | a
   Parse the string “a + a * a” using the same parser. #

5. What is Left recursion? Check if the following grammar is left recursive. and take necessary action if it exists:

   S → SS + | SS*| a #

6. Write a short note on: YACC
7. Construct LR(0) parsing table for the following grammar and analyse the contents of stack and input buffer and action taken after each step while parsing the input string "abbcbcde".:

   $S \rightarrow aCDe$
   $C \rightarrow Cbc$
   $C \rightarrow b$
   $D \rightarrow d$

8. Construct SLR parser for the following grammar and parse the input "()()" :

   S→ (S)S | ε

9. Compare Bottom-Up and Top-Down parser. #
10. Compare Pattern, Lexeme, and token with example. #

6. Compilers: Synthesis phase

1. Explain with suitable example code optimization techniques.
2. Explain different issues in code generation phase of a compiler.
3. Construct three-address code for the following program: (PYQs)

```
For(i=0;i<10;i++)
{
If (i<5)
a=b+c*3;
else
x=y+z;
}
```

```
while (a<b) do
    if(c<d) then
        x:=y+z
    else
        x:=y-z
```

```
i= 1;
x = 0;
while (i <= n)
{
        x = x + 1;
        i = i + 1;
}
```

4. What are the different ways for Intermediate code representation? Explain with example.
5. Generate 3 address code for the following C program and construct flow graph with the help of basic blocks(PYQs):

```
i=1; j=1;x=5;
while(i<3)
{
    switch(i)   {
      case 1:   a[j++]=i+x;
                break;
      case 2:   a[j++]=i-x;
                break;    }
i++;
                }
```

```
min=a[0];
for (i=1;i<n;i++)
  if(a[i]>max)
    max=a[i];
flag=1;
```

6. Explain DAG and construct DAG for the following expression:
   x = m + p / q - t + p / q * y
7. Write a short note on Peephole Optimization. #
8. Explain the concept of basic blocks and flow graph with example of the three-address code.

	1	2	3	4	5	6
2024 Dec	5	15	20	15	25	40
2024 May	5	15	15	15	30	35
2023 Dec	5	25	15	20	25	30
2023 May	5	25	15	15	35	25
2022 Dec	5	10	25	15	25	40
Last 5 Avg	**5**	**15-25**	**15**	**15**	**25**	**30**
***2022 May**	5	15	10	20	20	25
Total	30	105	100	100	160	185

**20-marks MCQs asked in 2022 May*

SPCC Answer Bank

multiple times asked questions highlighted
question asked once with red font
# indicates 5-mark question

1. Introduction to System Software

1. Differentiate between System Software and Application software. #

Parameter	Application Software	System Software
Definition	Software designed to help the user perform specific tasks.	Software designed to operate hardware and provide a platform for other software.
Purpose	Used for specific purposes such as word processing, browsing, etc.	General-purpose software that supports the functioning of the entire system.
Environment	Runs within an environment created by the operating system.	Creates and manages its own environment to run itself and other applications.
Execution Time	Executes only when required by the user.	May run continuously or be required to run as long as the system is active.
Essentiality	Not essential for basic computer operation.	Essential for running and managing computer hardware and system operations.
Number	Greater in number; many application software types exist.	Fewer in number compared to application software.
Machine Dependent/Independent	Generally machine-independent, focusing on solving user problems.	Usually machine-dependent as it interacts closely with hardware.

2. Compare Compiler and Interpreter.

Category	Interpreter	Compiler
Working	Translates the high-level language into intermediate form line-by-line.	Compiles the whole program at once directly into machine language.
Speed	Interpreters are slow in comparison to compilers.	Compiled programs run faster than interpreted programs.
Memory Efficiency	Less compact.	More compact.
Suitability for Applications	Good for simple applications.	Good for complex applications.
Handling Code Changes	Does not require retranslation of the entire code for changes.	Requires recompilation of the entire code if any changes are made.
Error Display	Errors are displayed in every single line during execution.	Errors are displayed after compiling the whole code together.
Storage of Machine Code	Does not save the Machine Language.	Saves the Machine Language in form of Machine Code on disks.
Working Model	Interpretation Model is the basic working model of the Interpreter.	Linking-Loading Model is the basic working model of the Compiler.
Examples	JAVA SCRIPT, LISP, FORTH, BASIC.	VISUAL BASIC, C, C++, C#, COBOL, etc.

2. Assemblers

3. Explain forward reference problem with suitable example. #

A forward reference occurs when a symbol (like a label or variable) is used in the code before it is defined. This is a problem in single-pass assemblers, where the assembler reads the code only once meaning it might encounter an undefined symbol with no prior knowledge of its address.

Example:

```
JMP  LOOP          ; Forward reference to label "LOOP"
MOV  A, B
LOOP: ADD  A, C    ; Definition of the label
```

The assembler encounters JMP LOOP, but at that point, it doesn't know where LOOP is in memory because it's defined later.

How Forward Reference is Solved using Forward Reference Table:

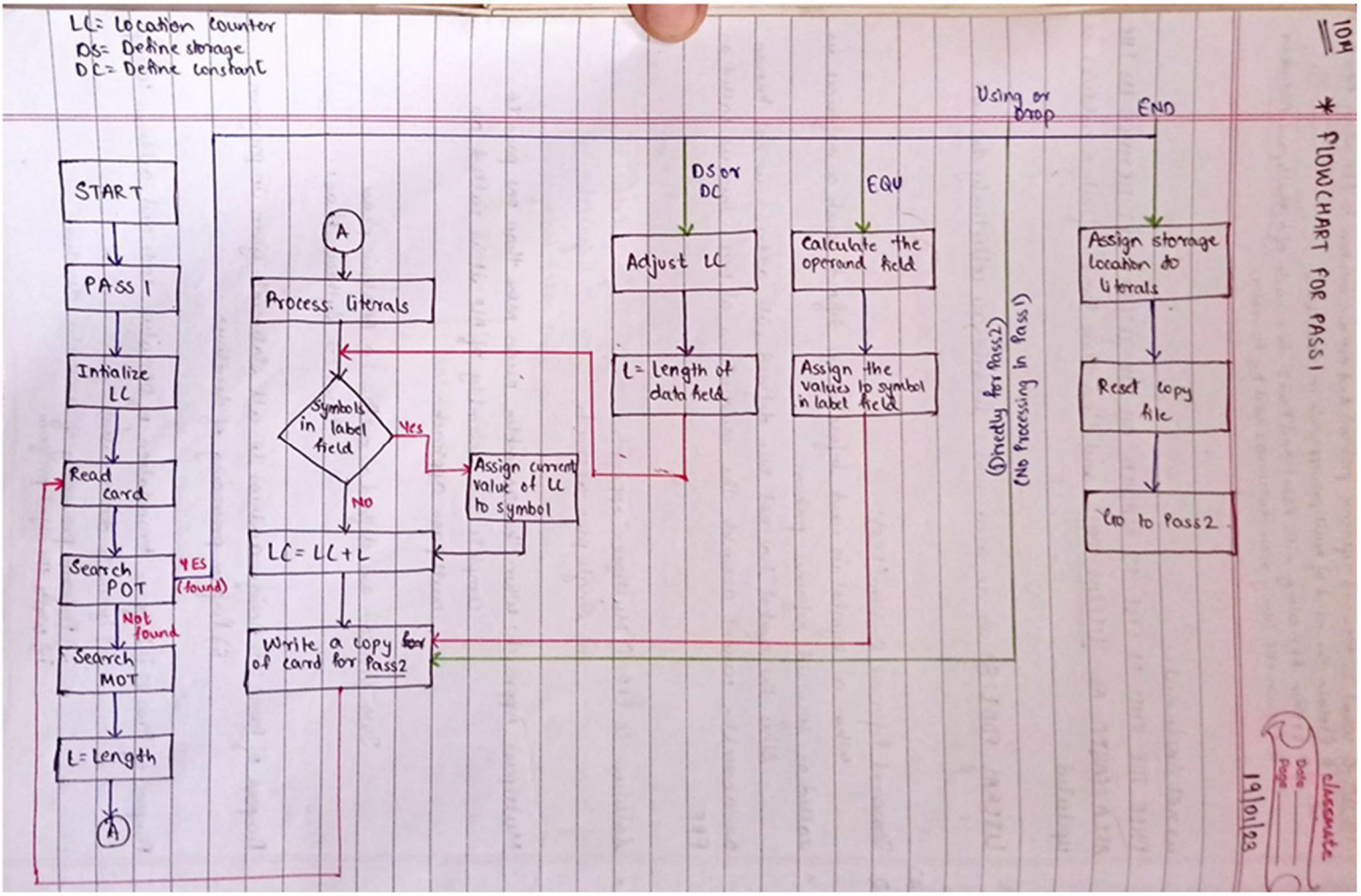
During Pass 1:

- When a symbol (label) is used before it's defined, its reference (location where it is needed) is stored in the Forward Reference Table.
- The symbol is added to the symbol table with an undefined address.

During Pass 2:

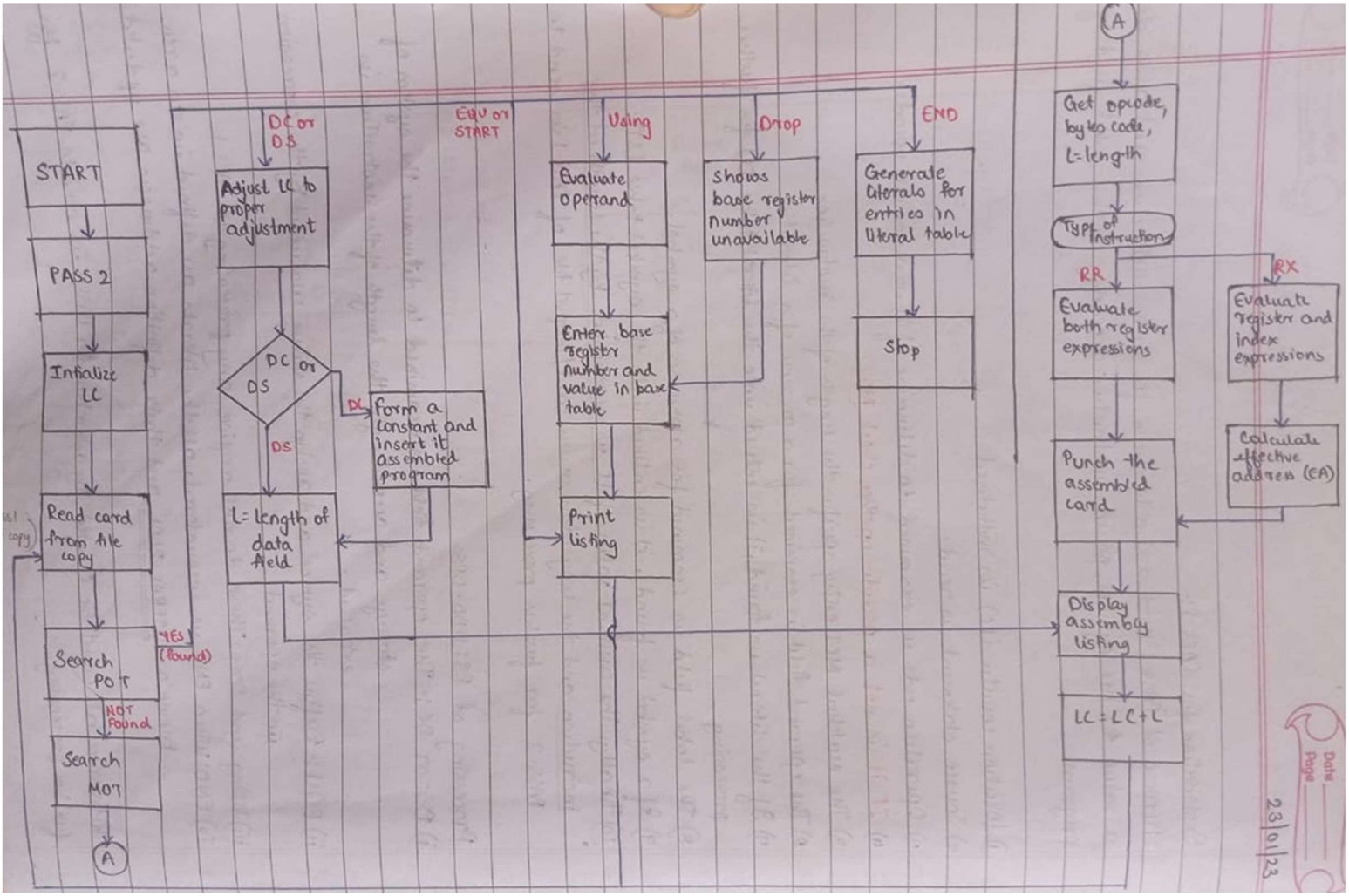
- When the symbol is finally defined, the assembler updates the symbol table.
- The assembler then revisits the locations stored in the FRT and patches them with the correct address.

4. Explain with flowchart design of two pass assembler.



Pass 1:

1) Location counter (LC) is initialized.
2) Source statement is read.
3) Operation code is examined to determine if it is a pseudo-opcode.
4) If it is not a pseudo-op then check MOT.
5) The matched MOT entry specifies the length of the instruction.
6) The operand field is examined for a presence of a literal.
7) If the literal is found, it is entered into the literal table (LT) for further processing.
8) The label field is examined for a presence of a symbol.
9) If a symbol is found, it is entered into the symbol table (ST).
10) Finally, the current value of LC is incremented by the length of the instruction and the copy of the source card / Output file of PASS-1 is send to PASS-2 for further processing.



Pass 2:

1. LC is initialized.
2. Statement is read from the copy file created by PASS-1
3. Operation code is examined to determine if it is a pseudo-op.
4. If it is not a pseudo-op, MOT is examined to find a match from the source statement.
5. There are two types of instruction format:
    a. RR: In RR format each of the register specific fields are evaluated.
    b. RX: Both register and index fields are evaluated.
6. After the instruction is evaluated, it is put into a necessary format for the processing by the loader.
7. A listing containing the copy of the source card is used to assign storage locations in hexadecimal format.

5. Draw and explain the flowchart of Pass 1 of two pass assembler with suitable example.

(Refer previous answer for flowchart and theory of Pass 1)

Goal of Pass 1:

- Assign addresses to all statements.
- Build the Symbol Table.
- Track Literals.

Example assembly program:

```
START 100
MOVER     AREG, =’1’
LOOP      ADD AREG, NUM
          SUB AREG, =’2’
          STOP
NUM       DC 5
END
```

Step-by-Step Pass 1 Table:

Line	Label	Opcode	Operand	LC
1		START	100	100
2		MOVER	AREG, ='1'	100
3	LOOP	ADD	AREG, NUM	101
4		SUB	AREG, ='2'	102
5		STOP		103
6	NUM	DC	5	104
7		END		—

Symbol Table:

Symbol	Address
LOOP	101
NUM	104

Literal Table:

Literal	Address
='1'	105
='2'	106

6. State and explain the types of assembly language statements with examples.

1. Imperative Statements:

Imperative statements in assembly language are instructions that explicitly specify the operations to be performed by the computer's CPU. These statements provide step-by-step instructions for the processor to execute.

```
MOV AX, 5     ; Move the value 5 into register AX
ADD AX, 3     ; Add 3 to the value in register AX
```

Purpose: Directly map to machine instructions, telling the processor what actions to perform

2. Declarative statements:

Declarative statements in assembly language are used to declare data or define structures without specifying the sequence of operations to be performed. These statements are often used to reserve memory space for variables, constants, or data structures.

```
DATA SEGMENT
X DB 5          ; Declare a byte variable X with value 5
Y DW 1000    ; Declare a word (2 bytes) variable Y with value 1000
DATA ENDS
```

Purpose: Define data, labels, or constants, providing information for the assembler rather than generating executable code.

a) DC (Define Constant):

Used to create and initialize constants (numbers or text).

```
NUM  DC  10      ; Store number 10
TEXT DC  'Hi'    ; Store text "Hi"
```

b) DS (Define Storage):

Used to reserve memory without giving it a value.
Useful for variables or buffers that will be filled during the program.

```
X   DS  1       ; Reserve 1 byte for variable X
BUF  DS  5      ; Reserve 5 bytes for a small buffer
```

3. Assembler directives:

Assembler directives are special commands used by the assembler (the program responsible for translating assembly code into machine code) to control the assembly process. These directives provide instructions to the assembler regarding how to process and organize the code.

```
.MODEL SMALL            ; Specify the memory model for the program
```

a) START:

The START directive is used to specify the starting address of the program.

START 1000 ; The program starts at address 1000

b) END:

The END directive marks the end of the assembly language program.

... END ; Marks the end of the program

c) EQU (Equate):

The EQU directive is used to assign a constant value to a symbol or label.

LENGTH EQU 100 ; Define a constant symbol LENGTH with a value of 100

d) ORG (Origin)

Sets the starting memory address.

ORG 2000 ; Start placing code/data at address 2000

e) LTORG (Literal Origin)

Tells the assembler to store any **literals** (constants) here.

DC X'0A' ; Define a hex value

LTORG ; Store the value here

7. **Consider the following assembly program:**

```
START   501
        A DS 1
        B DS 1
        C DS 1
READ A
READ B
MOVER       AREG, A
ADD         AREG, B
MOVEM       AREG, C
PRINT C
END
```

-Generate Pass-1 and Pass-2 and also show the content of database table involved in it.

(ChatGPT 'ed)

Pass 1: Tasks

- Build **Symbol Table (SYMTAB)**
- Generate **intermediate code**
- Maintain **location counter (LC)**

Location Counter (LC) Tracking:

LC	Statement	Action
501	START 501	Set LC = 501
501	A DS 1	Add A to SYMTAB
502	B DS 1	Add B to SYMTAB
503	C DS 1	Add C to SYMTAB
504	READ A	Intermediate code
505	READ B	Intermediate code
506	MOVER AREG, A	Intermediate code
507	ADD AREG, B	Intermediate code
508	MOVEM AREG, C	Intermediate code
509	PRINT C	Intermediate code
510	END	Stop processing

Symbol Table (SYMTAB):

Symbol	Address
A	501
B	502
C	503

Intermediate Code (Pass 1 Output)

(Format: LC) (Opcode Class, Opcode No) Operand)

LC	Intermediate Code
501	(AD,01) 501
501	(DL,01) 1
502	(DL,01) 1
503	(DL,01) 1
504	(IS,09) (S,01)
505	(IS,09) (S,02)
506	(IS,04) 1 (S,01)
507	(IS,01) 1 (S,02)
508	(IS,05) 1 (S,03)
509	(IS,10) (S,03)
510	(AD,02)

(Assumed opcodes: MOVER – 04, MOVEM – 05, ADD – 01, READ – 09, PRINT – 10, START – 01, END – 02)

Pass 2: Tasks

Generate **final machine code** using symbol table and intermediate code

Machine Code (Pass 2 Output)

Address	Machine Code (Example Format)
504	09 00 501 ← READ A
505	09 00 502 ← READ B
506	04 01 501 ← MOVER AREG, A
507	01 01 502 ← ADD AREG, B
508	05 01 503 ← MOVEM AREG, C
509	10 00 503 ← PRINT C

Summary of Tables Used

Symbol Table:

- Maps labels (variables) to memory locations.

Literal Table:

- Not used here (no literals like ='5' used)

Pool Table:

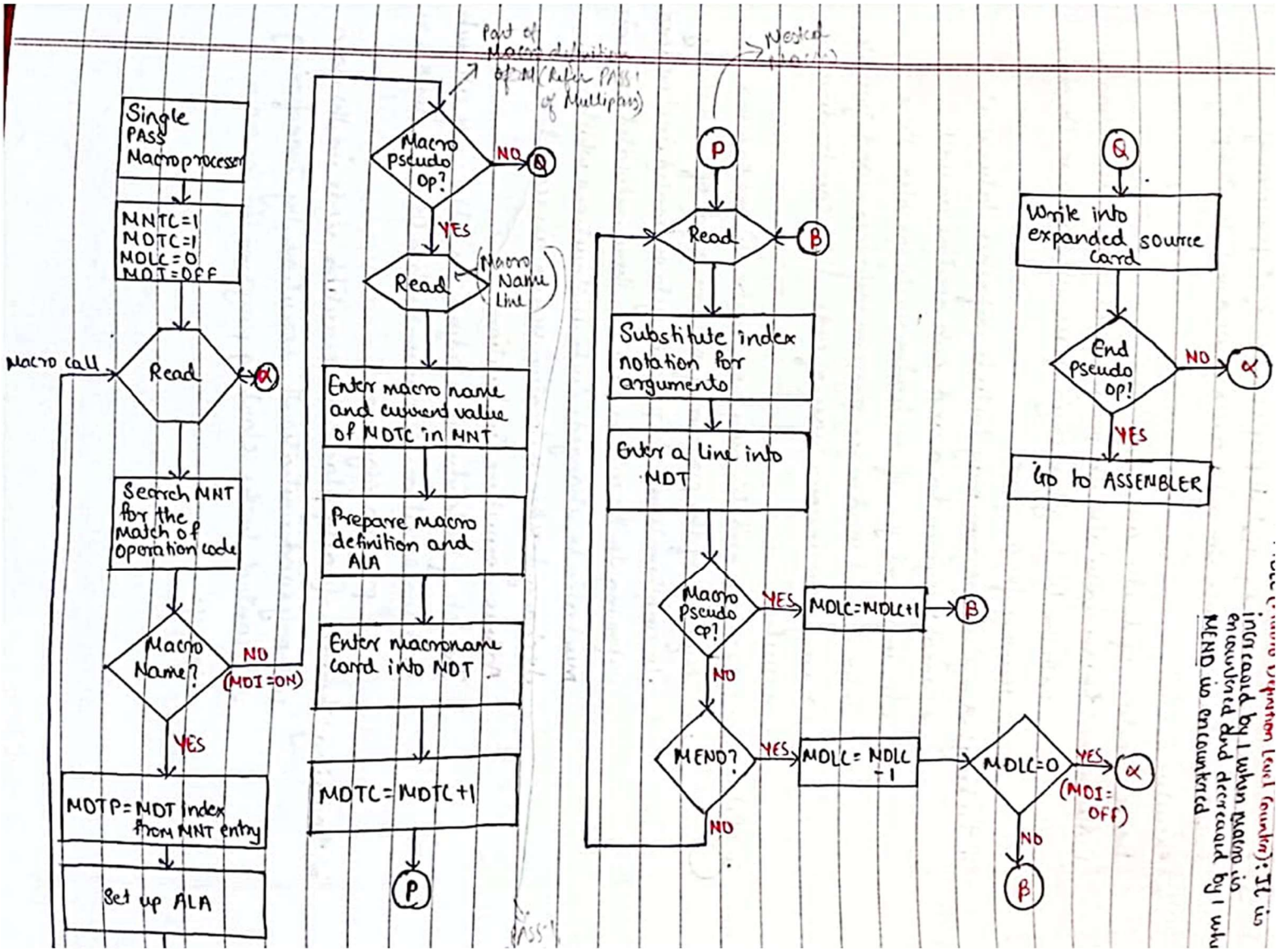
- Not needed (no literal pools here)

Intermediate Code:

- Stored between Pass 1 and Pass 2 to help generate final code

3. Macros and Macro Processor

8. Explain the working of a Single pass macro processor with neat flowchart.



Components of a Single Pass Macro processor:

1. Macro Name Table (MNT)

- Stores the names of all macros defined in the source code.
- Each entry includes:
  - Macro name
  - Starting index/line in the Macro Definition Table (MDT)

2. Macro Definition Table (MDT)

- Stores the actual body of each macro.
- Each line of the macro definition is saved here, with placeholders for parameters..

3. Argument List Array (ALA)

- Temporary structure used during macro definition and expansion.
- Stores formal arguments (during definition) and actual arguments (during call).

In a single-pass macro processor, these four variables play a crucial roles in managing macro definitions and expansions during processing:

MNTC (Macro Name Table Counter): Tracks the next free entry in the Macro Name Table (MNT).

MDTC (Macro Definition Table Counter): Points to the next free entry in the Macro Definition Table (MDT).

MDLC (Macro Definition Level Counter): Keeps count of nesting level during macro definitions.

MDI (Macro Definition Indicator): A flag (ON/OFF) to show whether a macro is currently being defined.

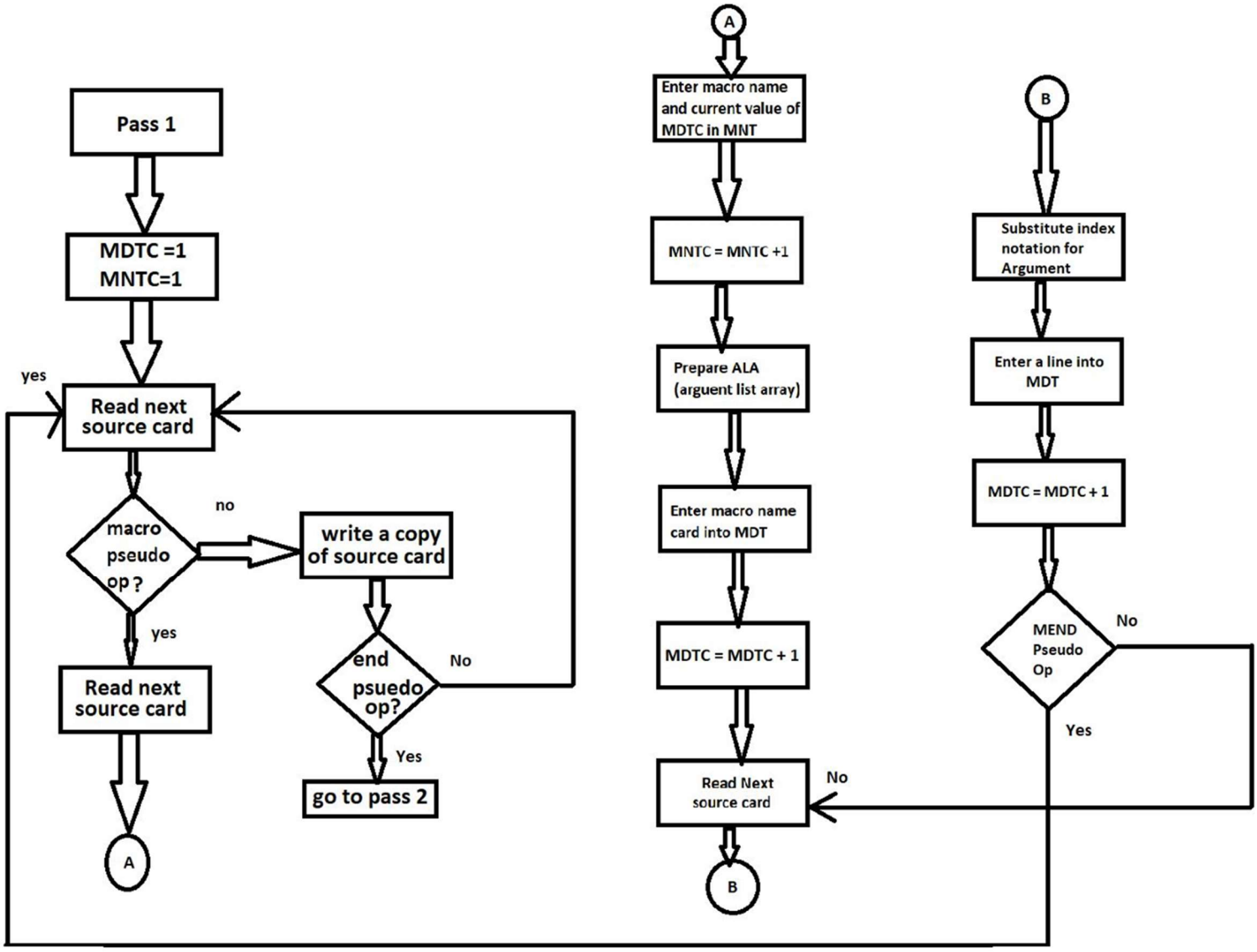
Working of a Single Pass Macro processor:

- The macro processor starts by initializing tables like:
    - Macro Name Table (MNT)
    - Macro Definition Table (MDT)
    - Argument List Array (ALA)
- It reads each line of source code:
    - If it finds a macro definition (MACRO):
        - The macro name is added to the MNT.
        - The macro body is stored in the MDT.
        - Formal parameters are replaced with indexes in the ALA for future substitution.
        - When it reaches MEND, the macro definition is complete.
- If it finds a macro call:
    - The MNT is used to locate the macro in the MDT.
    - Actual arguments are stored in the ALA.
    - The macro body is expanded using the ALA and written to output.
- If it's a normal (non-macro) line, it is directly copied to the output.
- The process continues until an END pseudo-op is found.
- The final expanded source code is then passed to the assembler for further compilation.

9. Explain and Draw a neat flowchart of two pass macro processor.

Pass 1 (Definition of MACROS)

- Scans input line by line, initializing **MDTC = 1** and **MNTC = 1**.
- Stores macro definitions (excluding *MACRO* line) in **MDT** and adds the name to **MNT** with a pointer.
- Ends when *END* pseudo-op is encountered, transferring control to **Pass 2**.



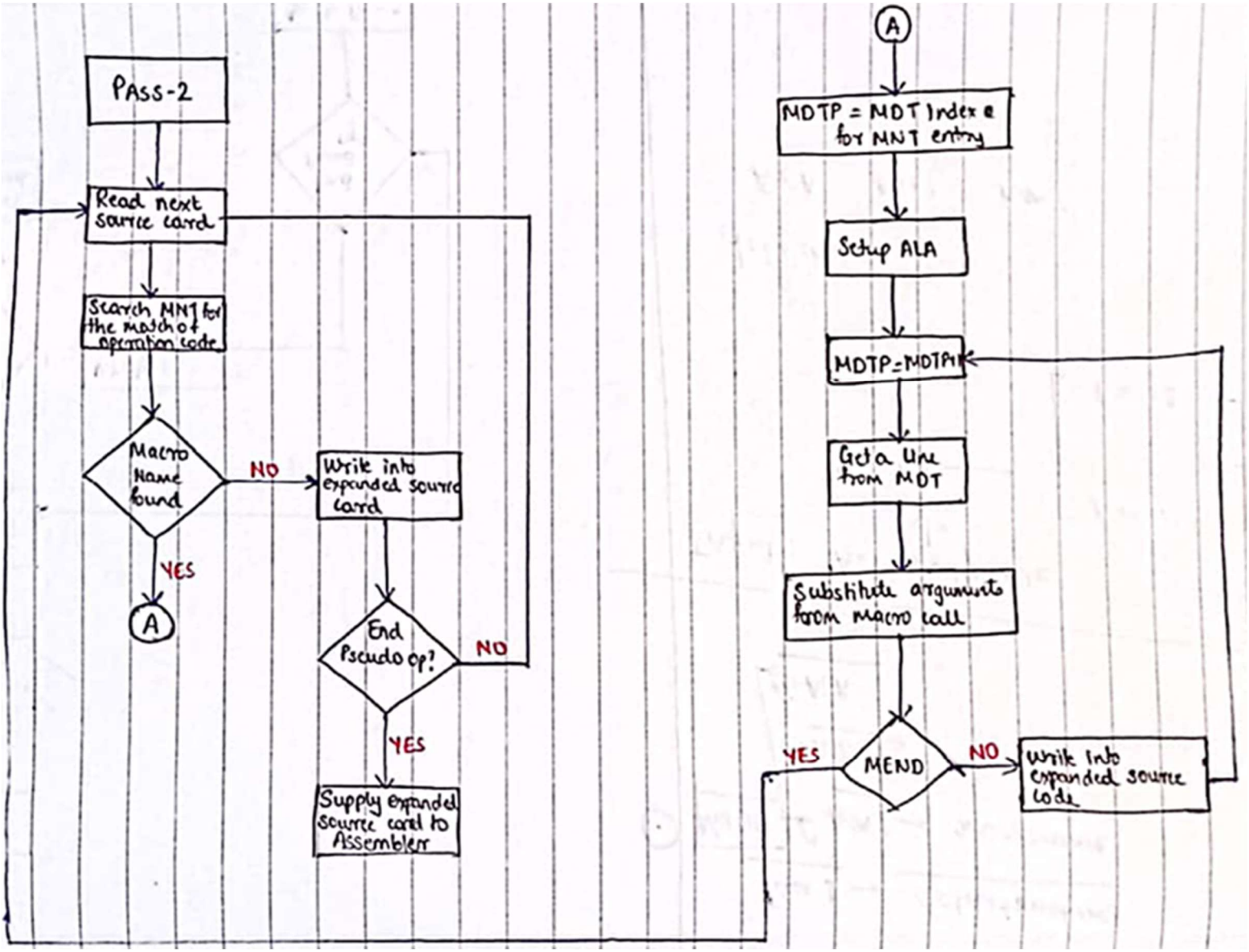
Pass 2 (Replacing MACRO calls by its definition)

Reads input lines, checking if the opcode matches an **MNT** entry.

Retrieves **MDTP** from **MNT** and sets up **ALA** for argument substitution.

Expands macros by replacing dummy arguments with actual values from the call.

Stops expansion at *MEND* and continues scanning until *END*, passing the expanded source to the assembler.



10. Explain advanced macro facilities with suitable examples.

Advance macro facilities are aimed at supporting semantic expansion. Key features include:

1. Facilities for alteration of flow of control during expansion.
2. Expansion time variables
3. Attributes of parameters.

1. Alteration of Flow of Control during Expansion

Macro processors can control which macro statements are expanded using:

a) Sequencing Symbols (SS):

Defined using a label in the macro body (e.g., .LOOP).

Used to control macro flow using AIF, AGO, or ANOP.

b) AIF (Assembler IF):

Used for conditional expansion.

AIF (&ARG EQ 0) SKIP

MOVER AREG, ONE

.SKIP ANOP

If &ARG is 0, control jumps to label .SKIP, skipping the instruction.

c) AGO (Assembler GO):

Unconditional jump during expansion.

Syntax:

AGO <sequencing symbol>

Example:

AGO END

MOVER AREG, B

.END ANOP

Always jumps to .END, skipping MOVER.

d) ANOP (Assembler No Operation):

Defines a label (sequencing symbol) with no action.

Syntax:

<sequencing symbol> ANOP

2. Expansion Time Variables (EVs)

EVs are variables used only during macro expansion.

Local EVs (LCL): Exist only during a single macro call.

Global EVs (GBL): Accessible across multiple macro calls in a program.

Example:

```
MACRO
 LCL &X
 &X SET 1
 DB &X
MEND
```

When the macro is invoked, &X is a local variable that exists only during the macro call, is assigned the value 1 with &X SET 1, and its value is then used in DB &X to output DB 1.

3. Attributes of Parameters

Attributes of parameters allow the macro processor to check the type, value, or length of arguments passed to a macro. These are useful in conditional expansion based on parameter properties.

Commonly used attributes:

Attribute	Description	Example
`LENGTH`	Returns the number of characters	`&LEN = LENGTH(&PARAM)`
`TYPE`	Returns type (constant, register, etc.)	`TYPE(&ARG)`
`VALUE`	Returns numeric value	`&VAL = VALUE(&ARG)`

11. Write a short note on: Parameterized macros.

A parameterized macro is a type of macro that accepts arguments and inserts them into its body during macro expansion. It cannot directly modify the instruction being replaced; instead, it uses dummy parameters in the macro definition, which are substituted by actual arguments during macro invocation.

This allows flexibility by customizing each macro call while keeping the macro definition reusable. These macros are powerful tools for inline code expansion without writing repetitive code.

Example:

```
MACRO
LOADVAL &REG, &VAL
   MOV &REG, &VAL
MEND
```

Macro Call: LOADVAL A, 5

D

Expansion: MOV A, 5

This makes code cleaner and more efficient by reusing logic with different values or registers.

12. With reference to MACRO, explain the following tables with suitable example:
I) ALA ii) MDT iii) MNT

1. Argument List Array (ALA)

ALA is used in both pass 1 and 2.

– Dummy arguments are substituted with the positional indicators in ALA during pass 1. Also the dummy arguments on the memory are represented by #. Where # is used by the macros. For example consider the Fig. 3.5.1. The store macro definition would be

```
LOOP1   LAB   &A1, &A2, &A3
& STRG          D    3, #3
 # 0             D       2, #2
                 D       1, #1
                MEND
                 ⋮
```

Fig. 3.5.1 : MACRO DEFINITION TABLE FOR PASS 1

– The index markers # are replaced with macro call arguments in PASS2.

LOOP1 LAB DATA1,DATA2, DATA3

Generated ALA is

Argument list Array	
Index	Arguments (2 bytes)
0	"XYZ bbb"
1	"RECORD1 bbb"
2	"RECORD2 bbb"
3	"RECORD 3 bbb"

b – blank character

Fig. 3.5.2 : ALA during PASS 2

2. Macro Definition Table (MDT)

The MDT consists of 80 bytes string per entry. MDT stores each line of every macro definitions except the word MACRO. The MEND indicates the end of the MACRO.

Macro Definition Table (80 bytes per entry)		
Index		Card
:		:
10	& STRG	LAB &A3, &A1, &A2
11	# 0	A 3, #3
12		A 1, #1
13		A 2, #2
14		MEND
⋮		⋮

Fig. 3.5.3 : Macro Definition Table

Macro Name Table (MNT)

MDT table indicates the beginning of the macro definition consists of a character string and a pointer to the entry in the MDT. MNT is almost similar to MOT and POT

	8 bytes	4 bytes
Index	Name	MDT Index
⋮	⋮	⋮
4	"LAB bbb"	10
⋮	⋮	⋮

Fig. 3.5.4 : Macro Name Table

13. Explain conditional macro with suitable example.

A conditional macro is a type of macro that uses conditions (like IF, ELSE, ENDIF) to control which parts of the macro body get expanded based on the values of arguments passed during the macro call.

This helps make the macro more flexible and reusable, especially when the output code depends on some condition.

Basic Syntax:

```
IF condition
  <true block>
ELSE
  <false block>
ENDIF
```

Example:

```
MACRO
PRINTMSG &FLAG
IF &FLAG EQ 1
   MVI A, 'Y'          ; Load Yes
ELSE
   MVI A, 'N'          ; Load No
ENDIF
MEND
```

The macro PRINTMSG takes one argument: &FLAG.

If the value is 1, it loads 'Y' in register A.

If the value is 0, it loads 'N'.

The macro changes its output based on the condition.

14. Construct the necessary data structure after compiling the following code by Pass-1 of two-pass macro processor:

```
1.  MACRO
2.  COMPUTE   &x, &a, &p
3.  MOVER     &a, &x
4.  MULT      &a, ='4'
5.  MOVEM     &a, &p
6.  MEND
7.  MACRO     &g, &k, &r
8.  MOVER     &r, &k
9.  SUB       &r, ='4'
10. MEND
```

To construct the necessary data structures after Pass-1 of a two-pass macro processor, we identify:

1. **Macro Name Table (MNT)**
2. **Macro Definition Table (MDT)**
3. **Argument List Array (ALA)**

Macro Name Table (MNT)

Index	Macro Name	MDT Index
1	COMPUTE	1
2	G	6

Macro Definition Table (MDT)

Index	MDT Entry
1	COMPUTE &x,&a,&p
2	MOVER &a,&x
3	MULT &a,='4'
4	MOVEM &a,&p
5	MEND
6	G &g,&k,&r
7	MOVER &r,&k
8	SUB &r,='4'
9	MEND

ALA for COMPUTE

Index	Formal Parameter
1	&x
2	&a
3	&p

ALA for G

Index	Formal Parameter
1	&g
2	&k
3	&r

15. Write a short note on: Macro facilities.

A macro facility refers to the set of features or functions that enable the processing and expansion of macros. A list of common macro facilities within macro processors:

1. **Macro Definition**
    - Allows the user to define a macro, typically with a name and a sequence of instructions or statements (e.g., MACRO in assembly language).
2. **Macro Expansion**
    - The macro processor replaces each macro call in the code with the defined macro content, performing text substitution.
3. **Parameters**
    - Macros can accept parameters, allowing for more flexible and reusable macro definitions (e.g., MACRO X, Y).
4. **Conditional Assembly**
    - Allows macros to be conditionally expanded or not, based on certain conditions (e.g., using IF, ELSE, ENDIF).
5. **Recursive Macros**
    - Some macro processors allow macros to call other macros or even themselves, which enables more complex expansions.

16. Explain Macro and Macro expansion with example.

Macro: A macro is a sequence of instructions grouped under a single name that can be reused multiple times in an assembly program. It is a way to define code templates that get expanded wherever they are called, reducing redundancy and improving code readability.

Macro Expansion: When a macro is called in the program, the assembler replaces the macro call with the actual sequence of instructions defined in the macro. This process is known as macro expansion.

Example:

```
ADDNUM MACRO A, B
   MOV AX, A              ;
   ADD AX, B
ENDM


ADDNUM 5, 10           ; Expands to:
                                ; MOV AX, 5
                                ; ADD AX, 10
```

This creates a macro named ADDNUM that takes two values (A and B) and generates two instructions.

Macro Call: ADDNUM 5, 10

When this line is used, the assembler replaces it with:

```
MOV AX, 5
ADD AX, 10
```

This is called macro expansion.

17. Explain different features of macros with suitable example.

1. **Code Reusability**
   - Macros help avoid the repetition of code by defining a set of instructions or statements that can be reused multiple times in a program. Once a macro is defined, it can be invoked as many times as needed, saving time and reducing errors.
   - **Example**: MACRO MOVE 2, 3 can be called multiple times to move to the same coordinates.
2. **Parameterization**
   - Macros can accept parameters, which makes them flexible. Instead of writing the same sequence of instructions multiple times, you can pass different values each time the macro is invoked, making the macro more dynamic and adaptable.
   - **Example**: MACRO ADD a, b expands to a + b when called as ADD 5, 3.
3. **No Memory Overhead**
   - Unlike function calls, macros do not create new memory locations for arguments or return addresses. They are expanded directly into the program code by the macro processor at the preprocessing stage, which reduces the memory overhead typically associated with function calls.
   - **Example**: MACRO PRINT 5 directly inserts the print instruction without using memory for a function call.
4. **Faster Execution**
   - Since macros are expanded inline by the macro processor, they eliminate the need for function calls or context switches, which leads to faster execution. The code for the macro is directly inserted into the program at the point of use.
   - **Example**: MACRO INC X expands to X = X + 1, directly updating the value.
5. **Conditional Expansion**
   - Macros can include conditional logic, allowing for different code expansions depending on certain conditions or flags. This makes macros adaptable to different contexts or environments, such as debugging or platform-specific code.
   - **Example**: MACRO DEBUG_LOG only expands if DEBUG is set, otherwise does nothing.
6. **Nested Macros**
   - Macros can invoke other macros, allowing for more complex, modular, and structured macro definitions. This capability lets a simple macro rely on the functionality of other macros, making code easier to maintain and extend.
   - **Example**: MACRO CUBE X can call SQUARE X to compute X * X * X.

4. Loaders and Linkers

18. What are the functions of a Loader. #

A loader is a system program responsible for loading executable programs into memory for execution. It performs the following key functions:

1. **Allocation:** Assigns the necessary memory space for the program in the main memory.
2. **Linking:** Combines two or more object programs or modules and supplies necessary information.
3. **Relocation:** It modifies the object program so that it can be loaded at an address different from its original location
4. **Loading:** It loads the object program into the main memory for execution.

19. Explain design of Direct Linking Loader with suitable example. Discuss the databases used.

Questions asked breakdown:
Explain Direct Linking Loader in detail. X2
Explain design of Direct Linking Loader. X2
Discuss the databases used in Direct Linking Loader.
Explain Direct Linking Loader with suitable example.

Direct Linking Loader:

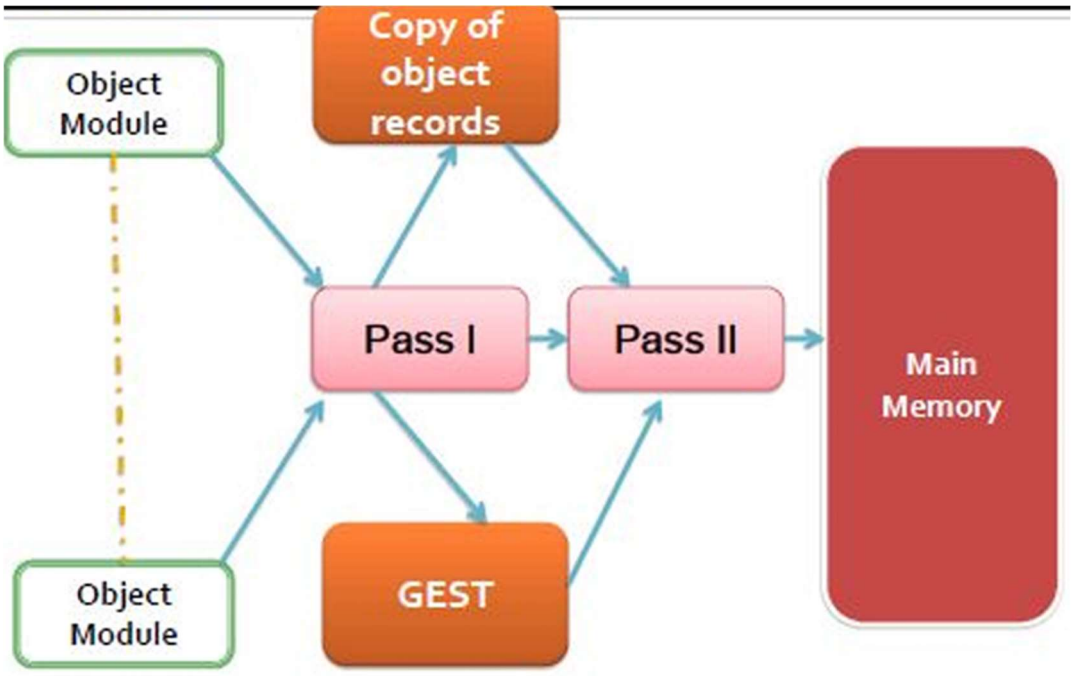
It is a general relocatable loader and is perhaps the most popular loading scheme presently used.
It is a relocatable loader.
It allows multiple procedure segments and multiple data segments.
The assembler must give the loader the following information:

1. The length of the segment.
2. A list of symbols defined in the current segment that may be referenced by other segments – public declaration.
3. A list of all symbols not defined in the segment but referenced in the segment – external variables.
4. Information about address constants.
5. The machine code translation of source program and the relative addresses assigned.

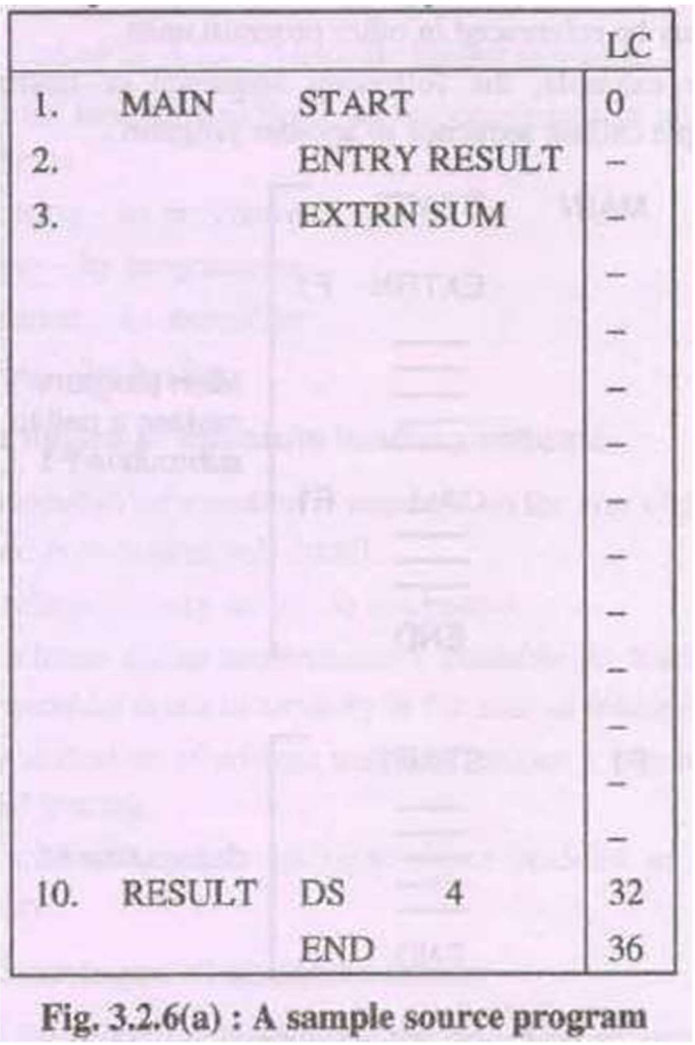
The object module produced by the assembler is divided into 4 sections:

1. External Symbol Directory (ESD)
2. Actual Assembled Program (TXT)
3. Relocation Directory (RLD)
4. End of the Object Module (END)

Two pass direct linking loader scheme:



Example:



			LC
1.	MAIN	START	0
2.		ENTRY RESULT	–
3.		EXTRN SUM	–
			–
			–
			–
			–
			–
			–
			–
			–
			–
			–
			–
10.	RESULT	DS 4	32
		END	36

Fig. 3.2.6(a) : A sample source program

1. External Symbol Directory (ESD):
Lists all external symbols (like functions or variables) that are defined in or required by the program.

Line No.	Symbol	Type	Rel. Location	Length
1	MAIN	SD	0	36
2	RESULT	LD	32	—
3	SUM	ER	—	—

2. Actual Assembled Program (TXT):
Contains the machine code instructions generated by the assembler. It also includes data and their corresponding load addresses.

Source Object Record No.	Relative Location	Object Code
1	0	`1F 0A`
2	32	`AA FF`

3. Relocation Directory (RLD):
Identifies addresses within the code that need to be modified during relocation.
Used when the program is loaded at a different memory location than originally assumed.

Source Object Record No.	ESD_ID	Length (in Bytes)	Flag (+/-)	Relative Address
1	3 (SUM)	2	+	10

4. End of the Object Module (END):
Marks the end of the object module. It may also contain the starting address for execution.

Advantages of Direct Linking Loader:

1. **Modular Programming Support –** Allows linking of multiple object modules developed independently.
2. **Efficient Memory Use** – Loads only necessary modules; helps save memory.

Disadvantages of Direct Linking Loader:

1. **Complex Implementation –** Requires detailed tables (ESD, RLD, TXT) and multiple passes.
2. **Longer Load Time** – Linking is done during loading, which can delay program start.

Databases Required for Direct Linking Loader: ***(If asked)***

(I) PASS 1

1. Object files used for input.
2. Initial Program Load Address (IPLA) gives the address to load the first segment given by operating system or programmer.
3. Program Load Address (PLA) keeps the track of each location for the segments assigned to it.
4. To store external symbol and its corresponding, assigned core address GEST is used.
5. Output of PASS1 is used as input for PASS2.
6. Load map is used to specify external symbols and their assigned output values.

(II) PASS 2

1. Output of PASS1 is used as input for PASS2.
2. The Initial Program Load Address Parameter (IPLA) from PASS1.
3. Program Load Address Counter prepared by PASS1.
4. GEST prepared by PASS1.
5. The correspondence between the external symbols core address and name mentioned on ESD and RLD cards is given by Local External Symbol array.

20. Explain absolute loader. State its advantages and disadvantages.

An absolute loader loads a binary program into memory for execution. The binary file includes:

- A header record with the load origin, program length, and execution start address.
- A series of binary image records, each containing part of the program's code, its load address, and byte count.

The loader reads the header, records the load origin and length, and then repeatedly reads image records, placing code into memory at specified addresses. Finally, it transfers control to the execution start address.

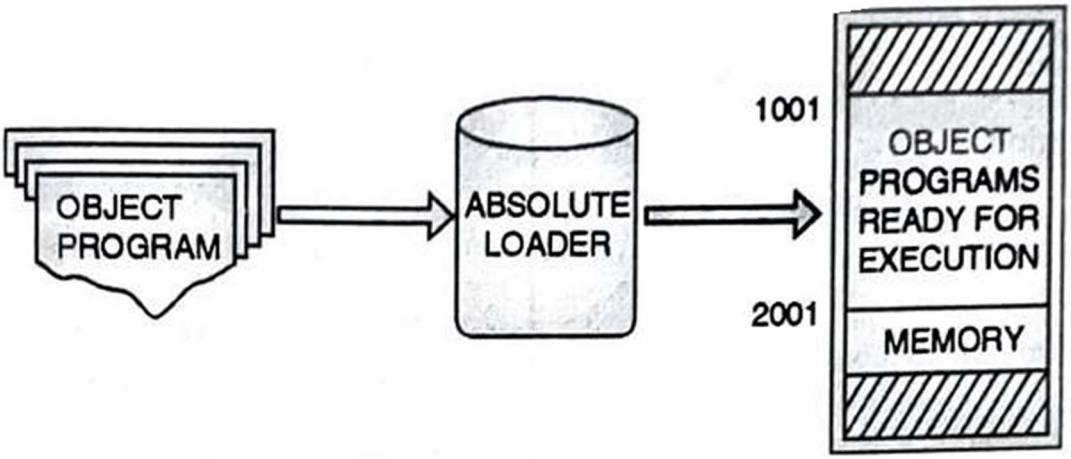


Fig. 4.5.3 : Absolute Loader

Advantages of Absolute Loading

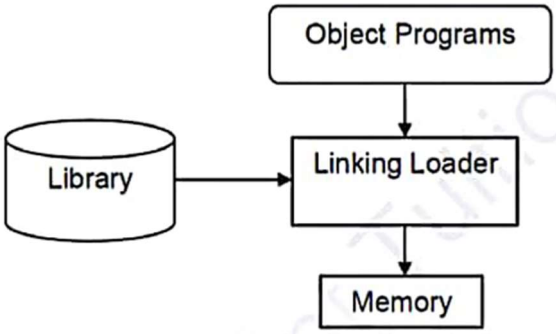
1. **Simplicity & Speed**: Easy to implement and fast in execution.
2. **Memory Efficient**: Loader is smaller than the assembler, saving memory.
3. **Multi-language Support**: Supports programs written in different languages by converting to a common object format.

Disadvantages of Absolute Loading

1. **Manual Addressing Required**: Programmer must specify memory addresses to the assembler.
2. **Risk of Overlap**: Programmer must ensure no overlapping of addresses.
3. **Reassembly Needed on Modification**: Any changes require complete reassembly of the program.

21. Explain Dynamic Linking Loader in detail.

A Dynamic Linking Loader is a component of an operating system that handles the loading of dynamically linked libraries (DLLs) or shared objects (SOs) into memory at runtime, rather than at compile time.



1. **Purpose:**
   - It allows programs to use shared libraries at runtime instead of linking them directly during the compilation process.
   - It ensures that external code or resources are loaded into memory only when needed, saving memory and disk space.
2. **Operation:**
   - When a program starts, the loader checks for the dynamic dependencies (libraries) defined in the program's binary (e.g., ELF or PE files).
   - It then loads these libraries into memory. If a library is already loaded, it is reused (shared among processes).
   - The loader resolves symbols (function names, variables) from the libraries and links them with the program.
3. **Process Flow:**
   - **Load Libraries:** The loader identifies the shared libraries required by the program and loads them into memory.
   - **Relocation:** If the library code isn't loaded at the exact address expected, the loader adjusts the program's address references to point to the correct memory locations.
   - **Symbol Resolution:** The loader resolves any symbols (functions, variables) that the program or libraries use.
   - **Linking:** The loader binds the resolved symbols to the actual memory locations in the loaded libraries.
   - **Execution:** Once the libraries are linked, the program starts executing with access to the dynamically linked functions.
4. **Advantages:**
   - **Memory Efficiency:** Multiple programs can share a single copy of a library.
   - **Modularity:** Programs can be updated without recompiling them, as long as the interface to the library remains the same.

22. What is relocation and linking concept in Loaders.

Relocation:

Relocation is the process of modifying address-dependent instructions in a program so it can be loaded at a different memory location than originally specified. Some instructions contain absolute addresses; the loader updates these using relocation information to fit the current memory space.

Linking:

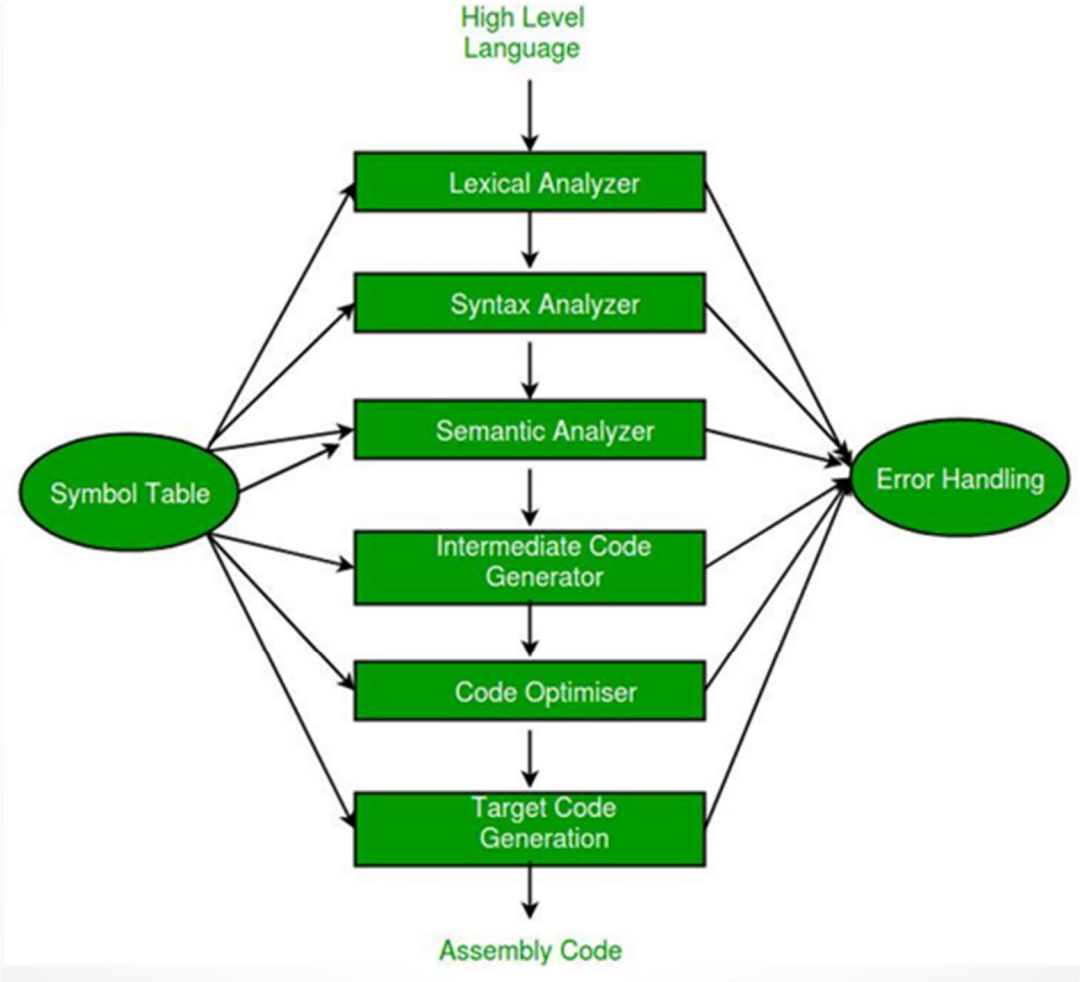
Linking is the process of combining multiple object modules into a single executable. It resolves external references between programs or modules by connecting function calls and variable references to their correct definitions across modules.

Together, relocation and linking ensure that programs are loaded correctly and external dependencies are resolved before execution.

5. Compilers: Analysis Phase

23. Explain the different phases of compiler with suitable example; or a given statement. (Previously asked statements)

i. a = a * b – 5 * 3 / c
ii. P = Q + R – S * 3
iii. a=b*c+10



Analysis Phase:

An intermediate representation is created from the given source code. It consists of the following:

- Lexical Analyzer: The lexical analyzer divides the program into “tokens.
- Syntax Analyzer: The Syntax analyzer recognizes “sentences” in the program using the syntax of the language
- Semantic Analyzer: Semantic analyzer checks the static semantics of each construct.
- Intermediate Code Generator: Intermediate Code Generator generates “abstract” code.

Synthesis Phase:

An equivalent target program is created from the intermediate representation. It has two parts :

- Code Optimizer: Enhances intermediate code by eliminating inefficiencies.
- Code Generator: Translates abstract intermediate code into specific machine instructions.

iii. a=b*c+10

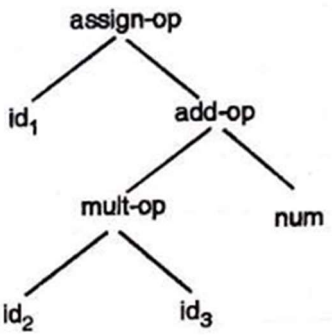
a = b * z + 10

a = b * z + 10

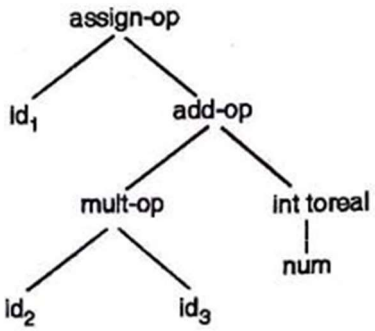
⇓ LEXICAL ANALYSIS

id_1 assign-op id_2 mult-op id_3 add-op num

⇓ SYNTAX ANALYSIS



⇓ SEMANTIC ANALYSIS



⇓ INTEMEDIATE CODE GENERATION

temp1 = id_2 * id_3

temp2 = int to real (num)

temp3 = temp1 + temp 2

id1 = temp3

⇓ CODE OPTMIZATION

t1 = id_2 * id_3

id1 = t1 + r_{num}

⇓ CODE GENERATION

movf	id_3, r_2
mulf	id_2, r_2
movf	r_{num}, r_1
addf	r_2, r_1
movf	r_1, id_1

f – floating point arithmetic

r_1, r_2 two registers to store the values

i.

```
int a, b, c =1 ;
a = a*b - 5*3/c;
```

Soln. :

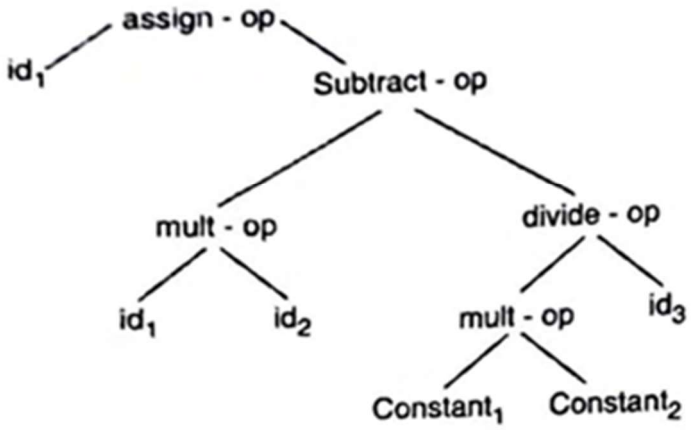
int a,b,c = 1,
a = a*b – 5*3/c,

⇓

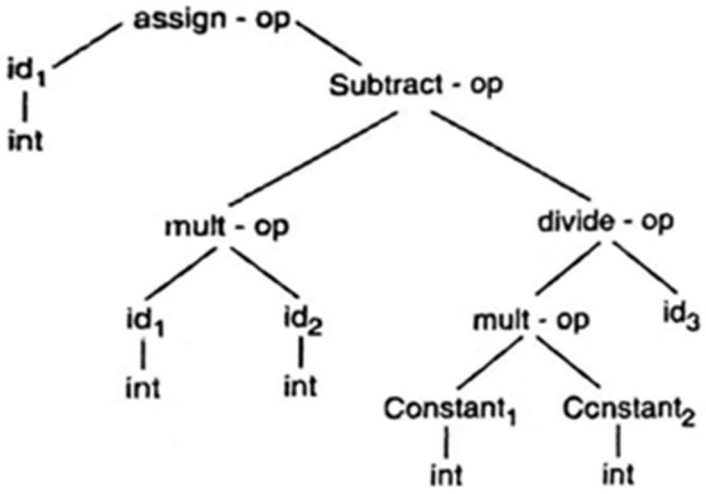
Lexical Analysis

id_1, id_2, id_3, = 1
id_1 assign-op id_1 mult-op id_2 subtract- op $constant_1$
mult - op $constant_2$ divide-op id_3

⇓ Syntax Analysis



⇓ Semantic Analysis



⇓ **In mediate Code Generation**

$$temp_1 = id_1 \times id_2$$
$$temp_2 = int\ (constant_1 \times constant_2)$$
$$temp_3 = temp_2 / id_3$$
$$temp_4 = temp_1 - temp_3$$
$$id_1 = temp_4$$

⇓ **Code Optimization**

$$temp_1 = id_1 \times id_2$$
$$temp_2 = (constant_1 \times constant_2) / id_3$$
$$temp_3 = temp_1 - temp_2$$

⇓ **Code generation**

MOV F	id_1, r_1
MOV F	id_2, r_2
MOV F	id_3, r_3
MOV F	r_1, r_2
MOV F	r_{num1}, r_4
MOV F	r_{num2}, r_5
MUL F	r_4, r_5
Divide F	r_4 /id_3
SUB F	$r_1 - r_4$

24. Construct LL(1) parsing table for the given grammar and state whether the given grammar is LL(1) or not. (Previously asked questions)

S→ AB | gDa
A→ ab | c
B→ dC
C→ gC | g
D→ fD | g

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \mid \varepsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \varepsilon$
$F \rightarrow f \mid \varepsilon$

S→ Ad
A→ aB | BC
B→ b
C→ e | ε

S→ 1AB | ε
A→ 1AC | 0C
B→ 0S
C→ 1

S → iCtSE | a
E → eS | ε
C → b

Example 5.19.16 : Test whether following grammar is LL (1) or not. Construct LL (1) Parse table.

S → AB | g Da
A → ab | c
B → dC
C → gC | g
D → fD | g

(10 Marks)

Solution :

S′ → S$
S → AB I g D a
A → ab I c
B → dC
C → gC I g
D → fD I g

For this we have to find the FIRST and FOLLOW sets. The FIRST and FOLLOW sets give the information about the next characters expected in the grammar.

FIRST (S) = {a, c, g}
FIRST (A) = {a, c}
FIRST (B) = { d }
FIRST (C) = { g }
FIRST (D) = {f, g}
FOLLOW (S) = { $ }
FOLLOW (A) = { d }
FOLLOW (B) = { $ }
FOLLOW (C) = { $ }
FOLLOW (D) = { a }

Table P. 5.19.16 : LL(1) Parse table

	A	b	C	d	f	g
S	S′ → S$		S′ → S$			S′ → S$
A	A → ab		A → C			
B				B → dC		
C	C → gc					C → g
D					D → fD	D → g

The above given grammar is LL (1) as their is no multiple entry in one column.

Example 5.19.16(A) : Construct LL(1) parsing table

$S \rightarrow aBDh$

$B \rightarrow Cc$

$C \rightarrow bC \mid \varepsilon$

$D \rightarrow EF$

$E \rightarrow g \mid \varepsilon$

$F \rightarrow f \mid \varepsilon$

MU - Dec. 17. 10 Marks

Solution :

$S' \rightarrow S\$$

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \varepsilon$

$D \rightarrow EF$

$E \rightarrow g \mid \varepsilon$

$F \rightarrow f \mid \varepsilon$

FIRST (S) = {a} FOLLOW(S) = {$}

FIRST (B) = {c} FOLLOW(B) = {g, f}

FIRST (C) = {b, ε} FOLLOW(C) = {$}

FIRST (D) = {g, f} FOLLOW(D) = {f, h}

FIRST (E) = {g, ε} FOLLOW(E) = {$}

FIRST (F) = {f, ε} FOLLOW(F) = {$}

FIRST (S') = {a}

LL(1) Passing Table

	a	b	c	g	f	h	$	ε
S'	$S' \rightarrow S\$$							
S	$S \rightarrow aBDh$							
B			$B \rightarrow cC$					
C		$c \rightarrow bC$						$C \rightarrow \varepsilon$
D				$D \rightarrow EF$	$D \rightarrow EF$			
E				$E \rightarrow g$				$E \rightarrow \varepsilon$
F					$F \rightarrow f$			$F \rightarrow \varepsilon$

25. Write a short note on: Syntax directed translation. #

Syntax-Directed Translation (SDT) combines grammar with semantic rules to assign meaning to syntactic structures. Each non-terminal in SDT can have attributes, evaluated using rules associated with production rules. These attributes store values such as numbers, strings, or memory locations. SDT enables translating programming constructs according to predefined semantic rules, as seen in compiler design.

SDT (syntax-directed translation) adds 'semantics rules' or actions to CFG products. As a result, we may refer to the grammar that has been augmented as attributed grammar

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

Syntax Directed Definition

A Context-Free Grammar (CFG) with attributes and rules is called a Syntax-Directed Definition (SDD). It associates grammar symbols in an extended CFG with rules that govern grammar production.

Types of Syntax Directed Definitions

S-attributed Translation If the node's attributes are synthesised attributes, the SDD is S-attributed. For evaluation of an S-attributed SDD, the nodes of the parse tree can be traversed in any bottom-up sequence.

L-attributed Translation: If the attributes of nodes are synthesized or inherited, an SDD is L-attributed. The parse tree can now be traversed exclusively from left to right. This is because L stands for left-to-right traversal in 'L-attributed translation.'

Types of Attributes: There are two types of attributes:

1. Synthesized Attributes

A synthesized attribute is derived from the attribute values of a node's children.
For a non-terminal symbol (node) N, synthesized attributes are:

- The attribute value of children.
- The total attribute values of N.

2. Inherited Attributes

An inherited attribute is derived from the attribute values of a node's parent or siblings.
For a non-terminal symbol (node) N, inherited attributes are:

- Parent's attribute values.
- Sibling's attribute values.
- Total attribute values of N.

26. Construct operator precedence parser for the grammar:

E → E+E | E*E | a

Parse the string "a + a * a" using the same parser. #

(id instead of a)

$E \rightarrow E + E \mid E * E \mid id$ MU – Dec. 15, Dec. 16, 10 Marks

Precedence Table

	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
Id	·>	·>	·>	·>
$	<·	<·	<·	<·

Consider the input string "id1 + id2 * id3"

The string with the precedence relations inserted from the above table is,

$ <· id1 ·> + <· id2 ·> * <·id3 ·> $

Having precedence relations allows identifying handles as follows :

- scan the string from left until seeing ·>
- scan backwards the string from right to left until seeing <·
- everything between the two relations <· a ·> forms the handle
- The handles can be found as

If <· Push in stack

If ·> Pop till a <· is found and reduce

$ <· id1 ·> + <·id2 ·> * <·id3·> $

$ <· E + <·id2 ·> * <·id3·> $

$ <· E + <·id2 ·> * <·id3·> $

$ <· E + <·id2 ·> * <·id3·> $

$ <· E + <· E * <·id3·> $

$ <· E + <· E * E ·> $

$ <· E + E· > $

S <· E ·> $

27. What is Left recursion? Check if the following grammar is left recursive. and take necessary action if it exists:

S → SS + | SS*| a #

Left recursion occurs in a grammar when a non-terminal refers to itself as the first symbol on the right-hand side of its production rule.

It is of the form: A → Aα | β

This is left-recursive because A appears first in Aα.

Yes, the given grammar is left-recursive, because:

S → SS+ and S → SS* both start with S on the right-hand side.

Removing Left Recursion:

To remove left recursion, we rewrite the grammar using a new non-terminal.

Original:

S → SS+ | SS* | a

Transformed:

S → aS'

S' → S+S' | S*S' | ε

This removes left recursion and makes the grammar suitable for top-down parsing.

28. Write a short note on: YACC

YACC stands for Yet Another Compiler-Compiler. It is a tool used to generate parsers in compiler design. It takes a context-free grammar as input and produces C code that can parse input according to that grammar.

- It is used with lex (a lexical analyser) to build compilers or interpreters.
- YACC helps in syntax analysis by creating a parser that checks if the tokens from the lexer follow the grammar rules.

Example Use:
Used to build parsers for programming languages like C, where YACC handles grammar rules and syntax checking.

Advantages of YACC:

1. **Automates Parser Creation:** Simplifies the process of writing parsers by generating code from grammar rules.
2. **Handles Complex Grammars:** Supports LALR(1) parsing, suitable for most programming languages.

29. Construct LR(0) parsing table for a given grammar and analyse the contents of stack and input buffer and action taken after each step while parsing the input string .

LR(0) Parser

Step1: Augment the Grammar and number it

Step2: Draw DFA Diagram

Step3: Parsing table

Step 4: Stack Implementⁿ

Step5: Parse Tree

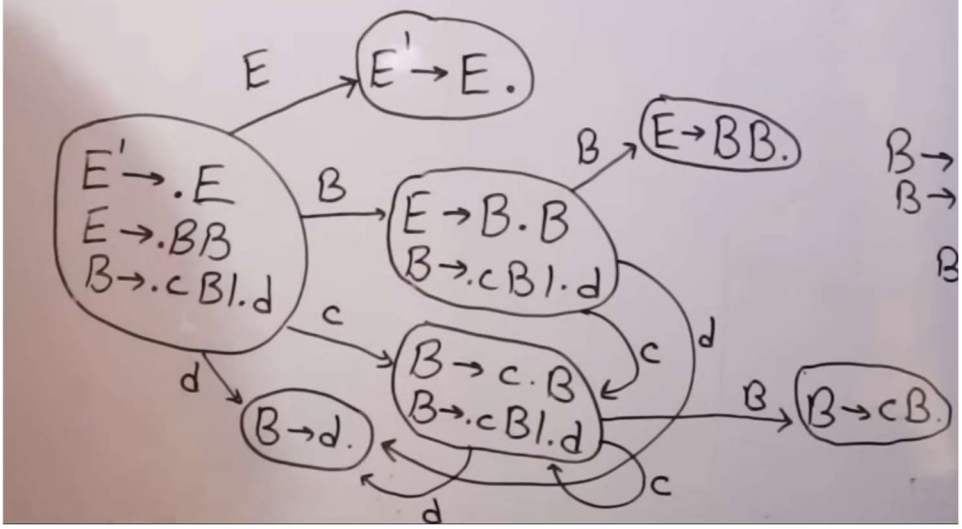
E → BB
B → cBId

String: ccdd$

Step1: Augment the grammar and number it

E' → E ⓪
E → BB ①
B → cBId
② ③

Step2: Draw DFA Diagram



Step3: Parsing Table

State	Action			Goto	
	c	d	$	B	E
I_0	S3	S4		2	1
I_1			A		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

Step:4: Stack Implementation

Stack	Input	Action
$0	ccdd$	Shift c → S3
$0c3	cdd$	Shift c → S3
$0c3c3	dd$	Shift d → s4
$0c3c3d4	d$	reduce B → d
$0c3c3B6	d$	reduce B → cB
$0c3B6	d$	reduce B → cB
$0B2	d$	Shift d → S4
$0B2d4	$	reduce B → d
$0B2B5	$	reduce E → BB
$0E1	$	Accept

30. Construct SLR parser for the following grammar and parse the input "()()" :
S→ (S)S | ε

Solution :

Augmenting grammar

1. $S' \rightarrow S$
2. $S \rightarrow (S)\ S$
3. $S \rightarrow \varepsilon$

Find the item sets

$I0 \rightarrow$ $S' \rightarrow \cdot S$
$S \rightarrow \cdot (S)\ S$
$S \rightarrow \cdot \varepsilon$

$I3 \rightarrow$ $S \rightarrow (S \cdot)\ S$

$I4 \rightarrow$ $S \rightarrow (S) \cdot S$
$S \rightarrow \cdot (S)\ S$
$S \rightarrow \cdot$

$I1 \rightarrow$ $S' \rightarrow S \cdot$

$I2 \rightarrow$ $S \rightarrow (\cdot S)\ S$
$S \rightarrow \cdot (S)\ S$
$S \rightarrow \cdot$

$I5 \rightarrow$ $S \rightarrow (S)\ S \cdot$

And now finally the parsing table is as :

State	Input/Action			Go To
	(	)	$	S
0	S2	R3	R3	G1
1			Accept	
2	S2	R3	R3	G3
3		S4		
4	S2	R3	R3	G5
5		R2	R2	

31. Compare Bottom-Up and Top-Down parser.

Parameter	Top-Down Parsing	Bottom-Up Parsing
Definition	Starts from the highest level of the parse tree and works downward using grammar rules.	Starts from the lowest level of the parse tree and works upward using grammar rules.
Parsing Approach	Attempts to find the leftmost derivations for an input string.	Attempts to reduce the input string to the start symbol of a grammar.
Parse Tree Construction	Parsing starts from the top (start symbol of the parse tree) and moves downward to the leaf nodes.	Parsing starts from the bottom (leaf nodes of the parse tree) and moves upward to the start symbol.
Derivation Order	Uses leftmost derivation.	Uses rightmost derivation.
Decision Making	Selects which production rule to apply to construct the string.	Selects when to apply a production rule to reduce the string to the start symbol.
Complexity	Easier to implement but may not work for all grammars (e.g., left-recursive grammars).	More powerful as it can handle a broader class of grammars.
Backtracking	May require backtracking if the grammar is not LL(1).	Does not require backtracking as it reduces input stepwise.
Lookahead Requirement	Requires lookahead to decide which rule to apply.	Can work without extensive lookahead due to shift-reduce approach.
Error Handling	Difficult error detection, may fail for ambiguous grammars.	Better error recovery as partial derivations can be analyzed.
Example	Recursive Descent Parser, LL(1) Parser	Shift-Reduce Parser, LR(0), LR(1) Parsers

32. Compare Pattern, Lexeme and token with example.

Parameter	Pattern	Lexeme	Token
Definition	A rule or regular expression describing a set of strings	The actual string in the source code matching a pattern	A symbolic name or identifier representing a lexeme
Role	Specifies how lexemes are identified	Is the real input that gets matched	Used by the parser for syntax analysis
Abstraction level	High-level (general form)	Concrete (specific input)	Intermediate (links pattern and meaning)
Processed by	Lexer (used to match input)	Matched by lexer	Generated by lexer for parser
Example	`[0-9]+` (pattern for integer literals)	`123` (a number in source code)	`<NUM, 123>` (Token with type and value)

6. Compilers: Synthesis phase

33. Explain with suitable example code optimization techniques.

1. Machine Independent code optimization techniques:

1. Constant Folding:
Constant folding evaluates expressions with constant operands at compile time and replaces them with a single computed value.

Example:
area := (22.0 / 7.0) * r ** 2

is replaced with

area := 3.14286 * r ** 2

2. Constant Propagation:
Constant propagation replaces a variable with its assigned constant value wherever it appears in an expression.

Example:
pi := 3.14286
area = pi * r ** 2

is replaced with

area = 3.14286 * r ** 2

3. Dead Code Elimination:
Dead code refers to portions of the program that will never be executed. If a computation's result is never used, it is considered dead and can be removed from the code.

Example:

```
i = 0;
if(i == 1) {
   a = b + 5;
}
```

Here, the if statement is dead code because i == 1 will never be true.

2. Machine-Dependent Optimization

1. **Instruction Scheduling:** Reordering instructions to minimize CPU pipeline stalls and improve execution speed.

Example:

Before:

LOAD A

MUL A, B

ADD C, A

This may stall because MUL and ADD both depend on A.

After (Reordered to reduce stall):

```
LOAD A
ADD C, A      ; Done while MUL waits for A to be ready
MUL A, B
```

2. **Register Allocation:** Efficiently assigning frequently used variables to CPU registers instead of memory for faster access.

Example:

Before:

```
int a = x + y;
int b = a * 2;
```

'a' stored in memory, accessed again for 'b'

After (using register):

```
MOV R1, x
ADD R1, y          ; 'a' stored in register R1
MUL R2, R1, 2
```

Avoids storing and reloading from memory.

3. **Peephole Optimization:** Localized transformations in small instruction sequences to eliminate redundant operations, such as replacing multiply by 2 with left shift.

Example:

```
; Before optimization
MUL R1, R1, 2

; After optimization
SHL R1, 1          ; shift left by 1 = multiply by 2
```

Replaces a costly multiply with a faster shift instruction.

34. Explain different issues in code generation phase of a compiler.

The code generation phase translates intermediate code into target machine code. It must ensure that the generated code is correct, efficient, and optimized for the specific architecture. Several issues must be handled carefully to produce high-quality code:

1. Input Data for Code Generator

The code generator takes intermediate representations such as:

- **Three-Address Code** (quadruples, triples)
- **Linear Notations** (prefix, postfix)
- **Graphical Forms** (syntax trees, DAGs)

The generator receives such intermediate representations and it must understand and handle the format accurately.

2. Target Program

Knowledge of the machine architecture and instruction set is essential. It influences whether the code will be absolute, re-locatable, or assembly-level.

3. Instruction Selection

Choosing the right machine instructions based on the IR level and instruction set affects efficiency.
Example: Selecting INC A over ADD A, 1 is faster on some architectures.

4. Register Allocation and Assignment

Efficient use of limited CPU registers improves performance. The code generator decides which variables go into which registers at different stages.
Example: Keeping loop counters in registers avoids memory access.

5. Evaluation Order

The order of evaluating expressions affects temporary variable usage and register pressure.
Example: In a + b * c, evaluating b * c first may save registers.

6. Machine-Specific Constraints

Some architectures have alignment rules or special registers. The code must respect such constraints to run correctly.

7. Preservation of Correctness

The generated code must preserve the exact semantics of the source program under all execution paths.

8. Optimization Trade-offs

The compiler must balance between compilation speed and code quality. Over-optimizing can slow down compilation.

35. Construct three-address code for the following program: (PYQs)

```
while (a<b) do
    if(c<d) then
        x:=y+z
    else
        x:=y-z
```

```
For(i=0;i<10;i++)
{
If (i<5)
a=b+c*3;
else
x=y+z;
}
```

```
i= 1;
x = 0;
while (i <= n)
{
        x = x + 1;
        i = i + 1;
}
```

i)

Example 6.3.3 : Generate three address code for following code.

```
While (a < b) do
        if (c < d)  then
        x = y + 2
else
        x = y – 2
```

MU - May 15, May 19, 10 Marks

Solution :

Three address code for a given expressions

```
        while (a < b) do
        if (c < d) then
        x = y + 2
        else
        x = y - 2
```

```
L1 :    while (a < b) goto L2
        goto last
L2 :    if (c < d) goto L3
        Goto L4:
L3 :    t1 = y
        t2 = 2
        t3 = t1 + t2
        x = t3
L4 :    t1 = y
        t2 = 2
        t3 = t1 – t2
        x = t3
        last
```

ii)

```
for (i = 0; i < 10; i++)
{
 if (i < 5)
  a = b + c * 3;
  else
  x = y + z;
  }
```

Three-Address Code (TAC):

```
i = 0
L1:   if i >= 10 goto L4
      if i < 5 goto L2
       t1 = y + z
       x = t1
       goto L3
L2:   t2 = c * 3
       t3 = b + t2
       a = t3
L3:   i = i + 1
        goto L1
L4:   exit
```

iii) i = 1;

x = 0;

while (i <= n)

{

x = x + 1;

i = i + 1;

}

Three-Address Code:

i = 1

x = 0

L1: if i > n goto L2

t1 = x + 1

x = t1

t2 = i + 1

i = t2

goto L1

L2: exit

36. What are the different ways for Intermediate code representation? Explain with example.

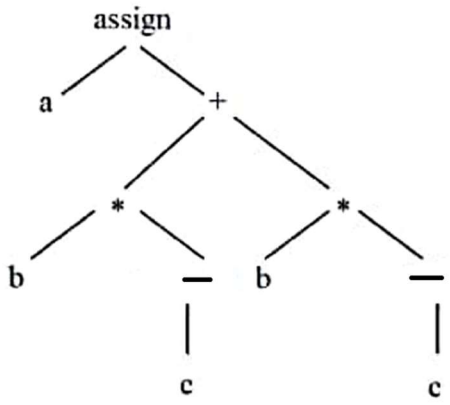
After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.
This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

Three ways of intermediate representation:

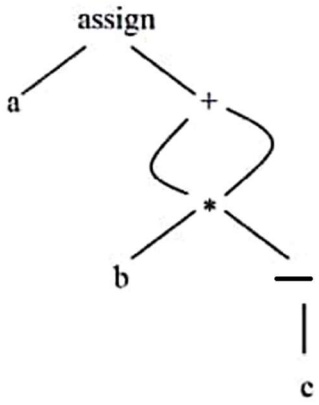
- Graphical Representation (Syntax tree & DAG)
- Postfix notation
- Three address code

Graphical Representations:
Syntax Tree & DAG: A Syntax Tree represents the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified.

A syntax tree and DAG for the statement a = b * - c + b * - c are as follows:



(a) Syntax Tree



(b) DAG

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is:
a b c - * b c - * + assign

Three-Address Code (TAC):
(Also applicable for Write a short note on: Three-address code representation. # [Asked twice])

Three-address code is a sequence of statements of the general form x: = y op z
Where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed-point or floating-point arithmetic operator, or a logical operator on Boolean valued data.
Example:

a = b + c;

d = a - e;

Three-address code:

t1 = b + c

t2 = t1 - e

d = t2

Three-address code is an intermediate representation in compilers where each instruction has at most three operands. It can be implemented using:

1. Quadruples

- Uses four fields: op, arg1, arg2, and result.
- Stores the operation and its two operands, along with the result.

Index	Operator	Arg1	Arg2	Result
1	+	b	c	t1
2	-	t1	e	t2
3	=	t2	—	d

2. Triples

- Uses only three fields: op, arg1, and arg2.
- Avoids naming temporary variables by using statement positions.

Index	Operator	Arg1	Arg2
0	+	b	c
1	-	(0)	e
2	=	(1)	d

Here, (0) refers to result of instruction 0, i.e., t1.

3. Indirect Triples

- Uses an array of pointers to triples instead of direct triples.
- Allows rearranging execution order by modifying the pointer list.

Pointer Table:

Index	Points to Instruction
0	0
1	1
2	2

Uses same Instruction Table of triple.

37. Generate 3 address code for a given C programs and construct flow graph with the help of basic blocks:

Consider the following C-code for bubble sort.

```
for i:=n-1 to 1 do
    for j:=1 to i do
        if(a[j]>a[j+1]) then
            begin
                temp:=a[j]
                a[j]:=a[j+1]
                a[j+1]:=temp
            end
    end for j loop
    end for i-ioop
```

Fig. 6.14.3 : C code for Bubble Sort

i:=n-1	temp:=a[t9]
L5: if (i<1) goto L1	t10:=j+1
j:=1	t11:=t10-1
L4 : if(j>i) goto L2	t12:=4*t11
t1:=j-1;	t13:=a[t12]
t2:=4*t1	t14:=j-1
t3:=a[t2]	t15:=4*t14
t4:=j+1	a[t15]:=t13
t5:=t4-1	t16:=j+1
t6:=4*t5	t17:=t16-1
t7:=a[t6]	t18=4*t17
if(t3<=t7) goto L3	a[t18]:=temp
t8:= j–i	L3: j:=j+1
t9:=4*t8	goto L4
	L2: i=i–1
	goto L5
	L1: stop

Fig. 6.14.4 : Three Address Code for Bubble Sort

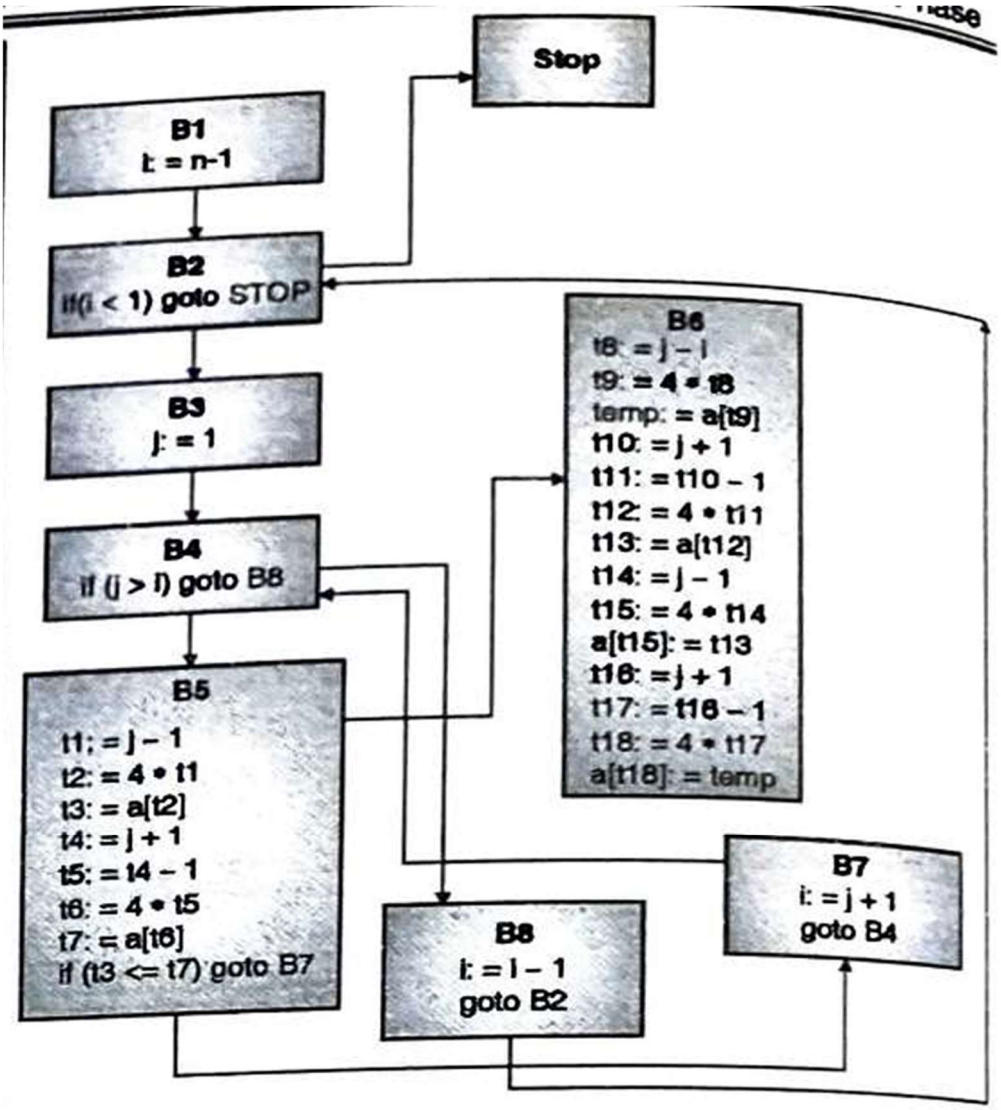


Fig. 6.14.5 : Control Flow Graph for Three Address Code of Bubble Sort

Example 2

Consider the following fragment of code :

0: n = readvalue

1: if n % 2 = = 0 goto 4

2: print n "is odd number." A

3: goto 5

4: print n "is even number."

5: exit B

In this code, we have 4 basic blocks:

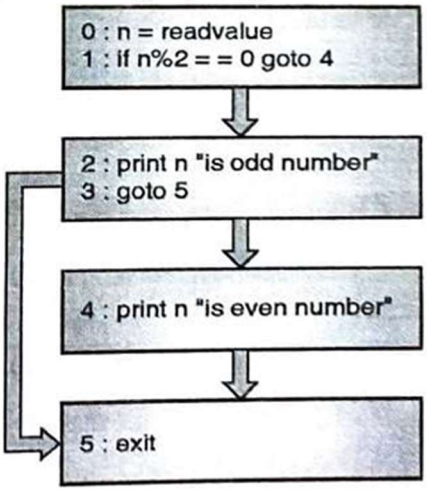
A from 0 to 1, Entry Block

B from 2 to 3,

C at 4

D at 5. Exit Block

Lines 4 and 5 are jump targets.



38. Explain DAG and construct DAG for a given expression.

In compilers, a syntax tree represents expressions with a unique path from the root to each leaf. However, when common sub-expressions exist, a Directed Acyclic Graph (DAG) is more efficient, as it allows sharing of identical subtrees, saving space and avoiding redundant computation.

A DAG is a directed graph with no cycles, where:

- Each node represents an operation or operand.
- Edges point from operators to their operands.
- A sub-expression like (b - c) reused in different parts of the expression is represented only once in the DAG.

Example Expression:
(a / b) + (a / b) * (c + d)

Step 1: Identify Common Sub-expressions

- (a / b) appears twice → it should be computed once in the DAG.
- (c + d) appears once.

Step 2: Build the DAG Step-by-Step

1. Leaf Nodes:

a, b, c, d

2. Build Sub-expressions:

- a / b → **node1**
- c + d → **node2**
- node1 * node2 → **node3**
- node1 + node3 → **Final node (root)**

DAG representation of (a | b) + (a | b) • (c + d)

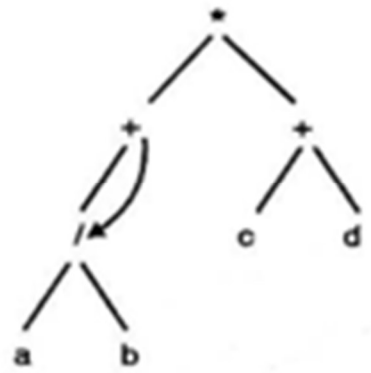


Fig. P. 6.15.1 : DAG Representation

Another example:

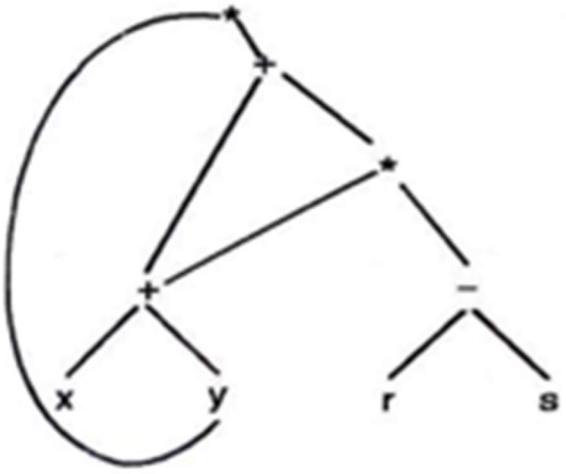


Fig. 6.15.4(b) : DAG for expression
((((x + y) *(r - s)) + (x + y)) * y)

39. Write a short note on Peephole Optimization.

Peephole optimization is a machine-dependent, local optimization technique used in the final stages of compilation. It examines a small set of instructions (a "peephole") in the generated code and looks for patterns that can be replaced with simpler, faster, or shorter equivalents without changing the program's behaviour.

Key Characteristics:

- Works on small windows of code (typically 1 to 5 instructions).
- Focuses on improving performance and reducing code size.
- Applied after intermediate code is generated, just before final machine code.

Common Optimizations Include:

- **Constant folding:** e.g., ADD R1, 0 → (remove)
- **Strength reduction:** e.g., MUL R1, 2 → SHL R1, 1
- **Eliminating unreachable or dead code**

Example:

; Before optimization

MUL R1, R1, 2

; After optimization

SHL R1, 1 ; shift left by 1 = multiply by 2

Replaces a costly multiply with a faster shift instruction.

40. Explain the concept of basic blocks and flow graph with example of the three-address code.

Basic Block: A basic block is a sequence of consecutive instructions in a program that:

- Has only one entry point
- Has only one exit point
- No jumps or labels in the middle of the block.

Example:

a = 5

b = 10

c = a + b

This is a basic block. If there were a jump or label between these, it would break the block.

Flow Graph: A flow graph is a directed graph where:

- Nodes represent basic blocks.
- Edges represent control flow (how control moves from one block to another).

Example with Three-Address Code (TAC):

C program:

```
if (a < b)
   c = a + b;
else
   c = a - b;
d = c * 2;
```

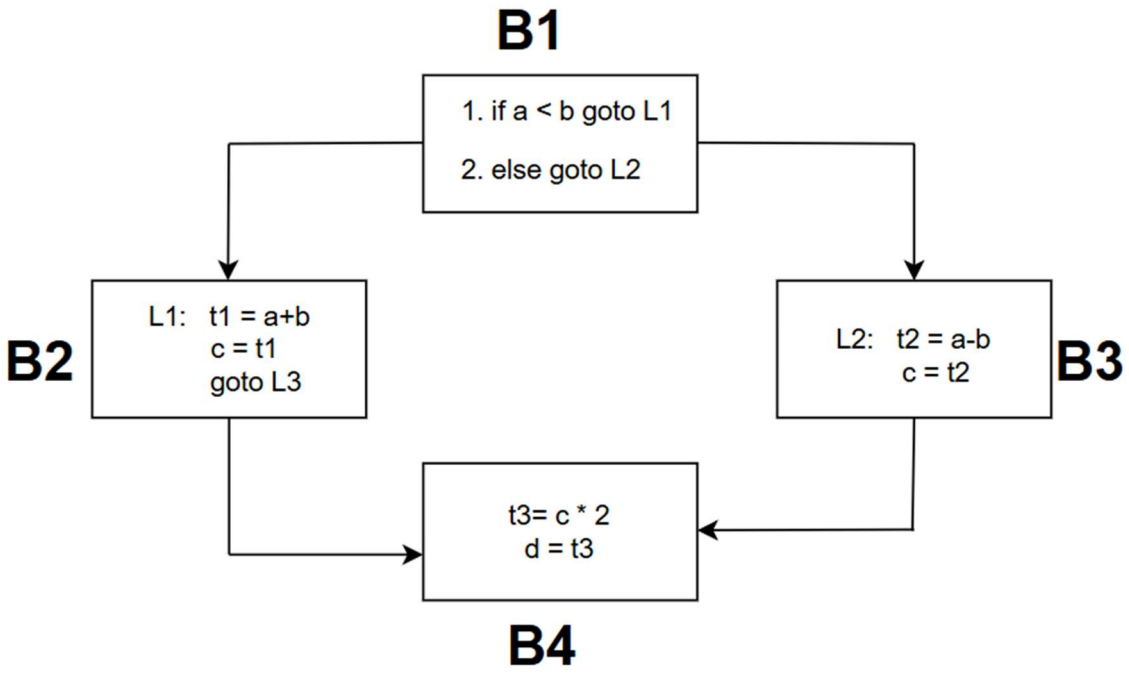
Three-Address Code (TAC):

```
1. if a < b goto L1
2. else goto L2
3. L1:    t1 = a + b
4.        c = t1
5.        goto L3
6. L2:    t2 = a - b
7.        c = t2
8. L3:    t3 = c * 2
9.        d = t3
```

Basic Block Division:

Basic Block	Instructions
B1	1, 2
B2	3, 4, 5
B3	6, 7
B4	8, 9

Flow graph:



~ AJ