70-Sakshi Ghonge SPCC May 2022

Q.1)

→ 1) C] The attribute can take value either from its parent or sibling.

→ 2) C] Flow Graph:

→ 3) C) d → c → b → a

→ 4) B) a-i bc au bne

→ 5) A) Constant Propagation.

→ 6) C) MDTC = 2, MNTC = 6.

→ 7) B) Line 1 - is assembler directive and Line 2 is Declaration.

→ 8) B) S -> E * F | E

→ 9) LC = 102, Intermediate code: (IS,01) (R0,01) (S,1)

→ 10) B) Relocation Bits

Q 2

A) i) Peephole optimization is technique used in compiler construction to optimize code generation for loops by analyzing and optimizing the code at the instruction level. It is called "peephole" because it looks at small window or peephole of instruction at a time, typically a few adjacent instructions, to identify and replace inefficient or redundant sequence of instructions with more efficient ones.

The main idea behind peephole optimization to detect and eliminate the redundant or inefficient instructions that do not contribute to the correct competation of the program. This can include removing unnecessary loads and stores, eliminating redundant computations and replacing expensive operations with more efficient ones. peephole optimization is usually performed as a post-processing step after the code has been generated by the compiler, and it is often combined with other optimization techniques to achieve better overall performance.

ii)	Application Software	System Software
1)	It is designed and built for specific task	1) Gives path to software applications to run
2)	It's specific software	It's general purpose software
3)	Programming is not complex	Programming is complex than application software
4)	Doesn't take hardware into consideration	Interacts with hardware directly
5)	Installed by users as per their need	Installed by manufacturer
6)	Works in the frontend	Works in the backend
7)	Can be said to be resources sharing Client Server	It is packaged program
8)	Example :- Word DOC, spreadsheet excel, Database, etc.	Example :- operating system, communication language.

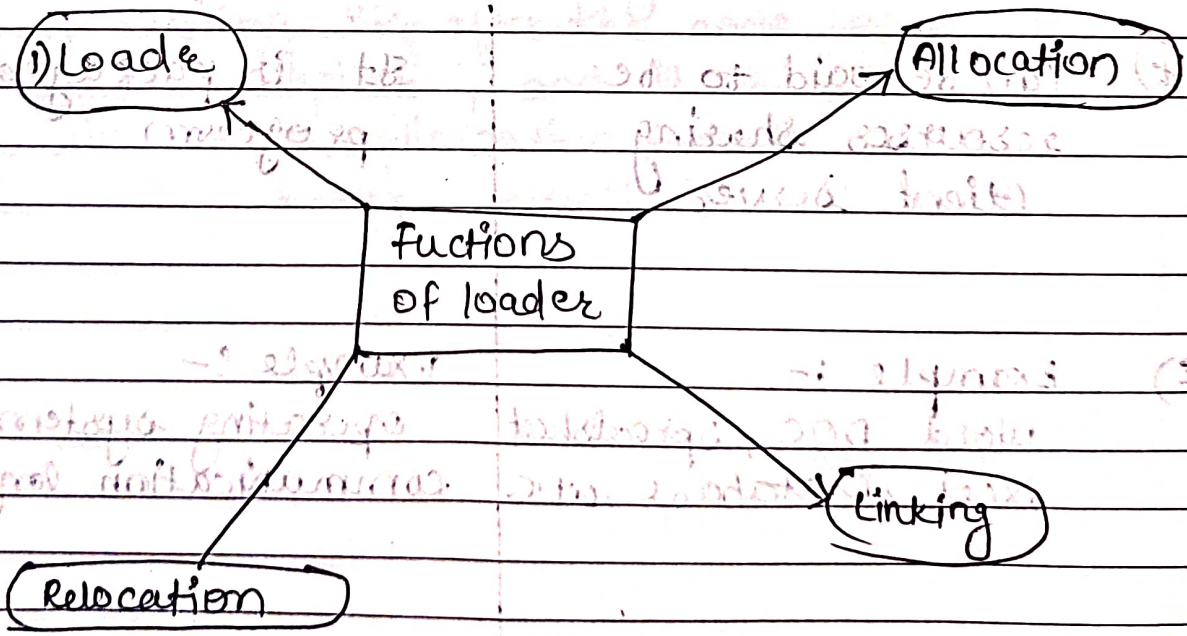
ii) • A loader is a system program which takes the object code of a program as input and prepares it for execution.

• Programmers usually define the program to be loaded at some predefined location in the memory.

• But this loading address given by the programmer is not be co-ordinated with the loader.

• The loader does the job of co-ordinating with the OS to get initial loading address for the .EXE file and load it into memory.

• Loader performs some functions they are as as follows.



B) i)

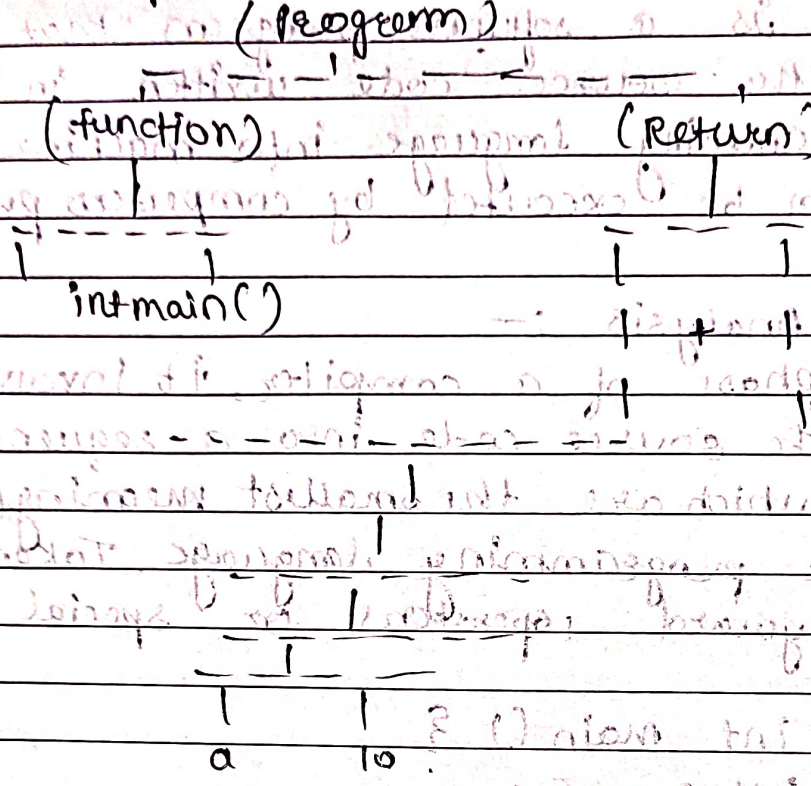
→ A compiler is a software program that translates the source code written in high level programming language into machine code that can be executed by computers processor.

1) Lexical Analysis :-
The first phase of a compiler, it involves breaking the source code into a sequence of tokens which are the smallest meaningful units in a programming language. Tokens can be keyword, operators or special symbols.
Example :-

```
int main () {
int a = 5;
int b = 10;
int sum = a+b;
return sum;
}
```

2) Syntax Analysis :-
It involves analyzing the sequence of tokens generated in the lexical analysis phase and building an abstract syntax tree (AST) which represents tree (ASS) syntactic structure of the source code. The AST is a hierarchical representation that captures the relationships between different language constructs.

Example



3) Semantic analysis :-
It involves checking the correctness of the source code in terms of the language's semantics. It verifies that the code is ~~undeclared~~ to variables, for semantic analysis phase would detect this error and report it

4) Intermediate code generation :-
Once the syntax and semantics analysis are complete the compiler may generate an intermediate step before generating the final machine code. The IR is a lower-level represents that is closer to the target machine architecture but still abstract enough to allow for further applications.

ii) The translation of the source code into the object code for the target machine, a compiler can produce a middle-level language code, which is reffered to as intermediate code or intermediate code representation

Types of intermediate code representation

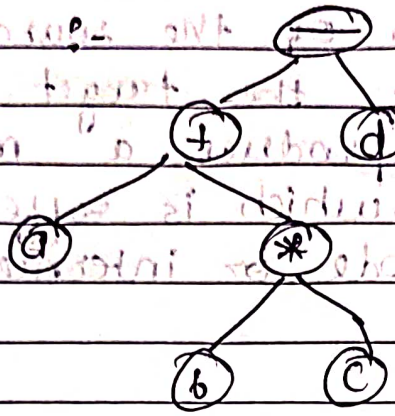
1) Postfix Notation 2) Syntax tree 3) three address code

i) Quadruples ii) Triples iii) Indirect triples.

1) Postfix Notation :-
In postfix notation the operator comes after an operand ie the operator follows an operand.
Example :-
a) Postfix Notation for the expression (a+b) * (c+d) is ⟶ ab+cd+*

2) Syntax tree :-
A tree in which each leaf node describes an operand and each interior node an operator. The syntax tree is shortened form of the parse tree.

Example :-



3) Three address code :-

The three address code is a sequence of a statement of the form of A - = B Op C, where ABC are either programmer defined names, constants or compiler generated temporary names the Op represents for an operator that can be the op ~~repr~~ fixed or floating point arithmatic operator or boolean valued data or logical operator.

In three address code there are 3 subtypes of it they are as follows :-

D* Quadruples representation :-

Records with the field for the operations and operands can be define three address statements. If it possible to use record structure with the field, first hold the operator, op next two hold operators 1 and 2 respectively and the last one holds the result.

ii) Triples Representation :- The contents of operand 1, operand 2 and result field are generally pointers for the names described in this field. Therefore it is important to introduce temporary names into the symbol table as they are generated. This can be prevented by using the positions of statements defines temporary values.

iii) Indirect Triples Representation :-
The indirect triple representation uses an extra array to list the pointer to the triple representation in the desired sequence. The triple representation for the statement X = (a+b)*c is as follows

Statement	Statement	location	operator1	operator2	operator3
(0)	(1)	(1)	+	a	B
(1)	(2)	(2)	-	c	
(2)	(3)	(3)	*	(0)	(1)
(3)	(4)	(4)	/	(2)	d.
(4)	(5)	5	:=	(3)	

Q.3.

A) → In Pass - 1, the assembler scans through the assembly program to generate a symbol table and resolve the address of labels.

Symbol table :-

Symbol	Address
START	501
A	—
B	—
C	—

Pass 2 :-

In Pass 2, the assembler generates the machine code instructions and fills in the missing addresses in the symbol table.

Machine code :-

Location	Instruction	Operand.
501	START	—
502	A	DS 1
503	B	DS 1
504	C	DS 1
505	READ	A
506	READ	B
507	A	—
508	MOVER	AREG, A
509	ADD	AREG, B
510	MOVEM	AREG, C
511	PRINT	C
512	END	—

Symbol table :-

Symbol	Address
START	501
A	507
B	508
C	509

Contents of Data table :-

Location	Data
502	—
503	—
504	—

The assembler assumes that the instruction set has predefined opcode and operand formats, and the addresses for the data symbols A, B and C will be filled in the runtime or by the loader as per the specific system architecture

→ B) There are different code optimization techniques they are as follows.

1. Compiler optimizations :- This refers to the optimizations include constant folding loop unrolling, function inlining, dead code elimination, and register allocation, among others.

2) Loop optimizations :- loops are common constructs in programming, and optimizing them can yield significant performance improvements loop optimizations techniques include loop unrolling loop fusion, loop interchange and loop vectorization

3) Inline expansion :- This techniques involved replacing function calls and return. This can improve performance by reducing the function call overhead and enabling further optimizations by the compiler.

4) Data flow Analysis :- It is a technique used to analyze how data flows through a program and optimize it based on this information.

5) Memory Optimization :- Memory access patterns can significantly impact performance. techniques such as caching, prefetching, and memory alignment can optimize memory usage and reduce memory related bottlenecks.

c)

→ first(A) = { a, c } follow(A) = { b, d, g }
first(B) = { d } follow(B) = { a, c, $ }
first(C) = { g } follow(C) = { a, c, $ }
first(D) = { f, g } follow(D) = { a, c, $ }.

As we can see, there are no common elements in the first and follow sets of any, NOT which means that there is no first/follow set instruction, therefore given grammer is LL(1)

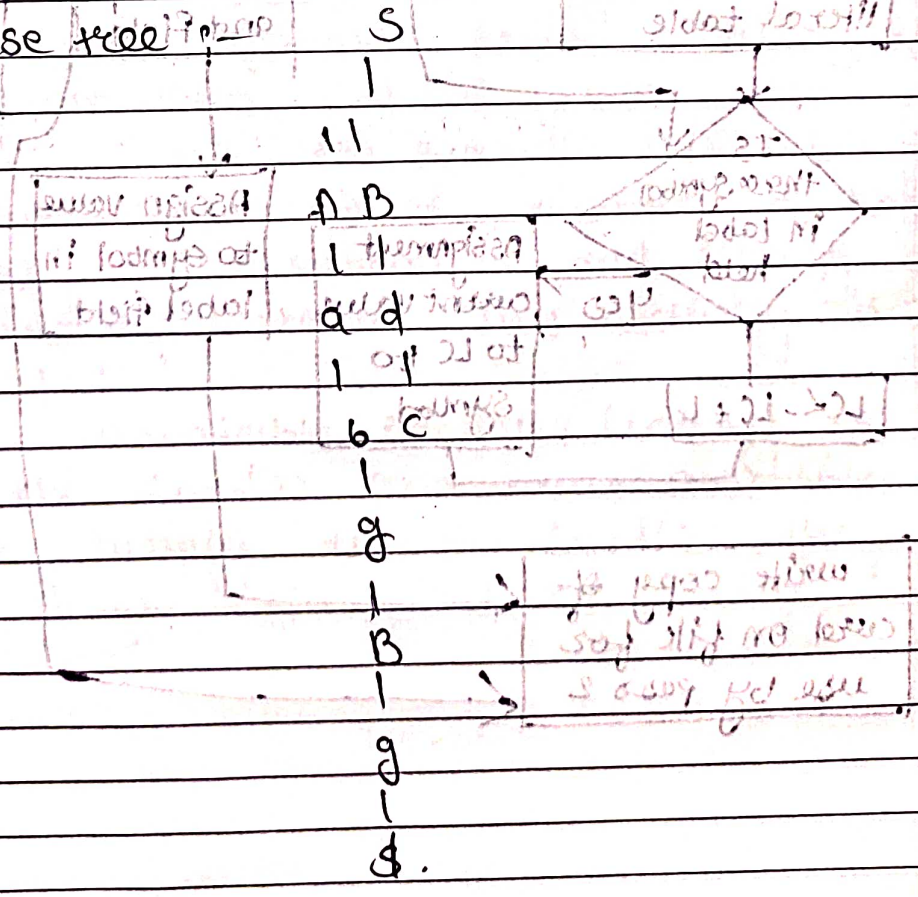
Lets generate a parse tree for the input "gfabg" using the given grammer:

Input: "gfabg"

Derivation: S -> AB -> Ab -> gC -> gB -> gfD -> gfabC -> gfabc -> gfabg.

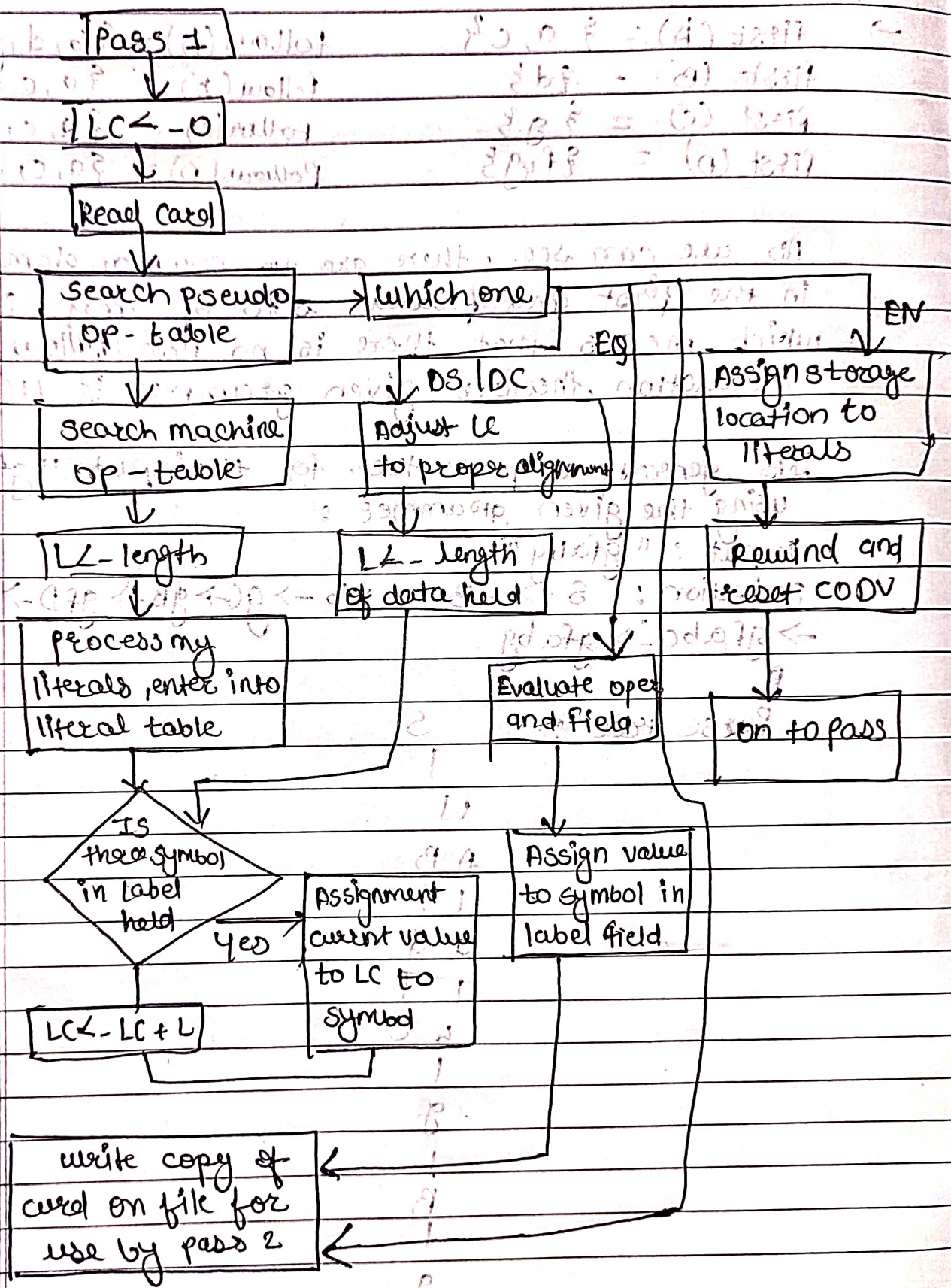
P.

Parse tree:



Q. 4

A. i)


Pass 1
LC <- 0
Read Card
Search pseudo Op - table
Which one
EQ
DS, DC
Adjust LC to proper alignment
LC <- length of data held
END
Assign storage location to literals
Rewind and reset CODV
on to pass
Search machine Op - table
LC <- length
Process any literals, enter into literal table
Evaluate oper and field
Is there symbol in label field
yes
Assignment current value to LC to symbol
Assign value to symbol in label field
LC <- LC + L
write copy of card on file for use by pass 2

Q. ii)

Relocation :-

- It modifies the object program by changing ib the certain instructions so that it can be loaded at different address from location orginally
- There are some program address dependent locations in the program, such address constants must be adjusted according to allocated spaces, such activity is done by loader is called relocation
- In absolute loader relocation is done by the assembler is aware of the starting address of the program.

Linking :-

- It links two or more object codes and provides the information needed to allow references between them.
- It resolves the symbolic references (code/data between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called Linking.
- In relocatable loader, linking is done by the loader and hence assembler must supply to the loader, the locations at which the loading is to be done

(iii) Pattern :- A set of strings is the input for which the same token is produced as output. This set of the steings is described by a rule called pattern associated with the token.

Lexeme :- A lexeme is a sequence of characters in the source program that is matched by the pattern for token

Tokens :- Tokens is a sequence of characters that can be treated as a single logical entity. Typical tokens are, identifiers, keyward, operators, special symbol, constants.

	Pattern	Lexeme	Token
1)	Const	const	const
2)	if	if	if
3)	< or <= or = or <>	<, <=, =, <>	relation
	or > or >= or letter	>, >=	id
	followed by letters & digit		
4)	Any numeric constants	pi	
5)	any character blw	3.14	num
	" and " except		
6)	pattern	"core"	literal

Q. 4.

i) It is used for identifying the macro name and performing expansion.

Features of macro processor :-

1. Recognized the macro defination
2. Save macro defination.
3. Recognized the macro call.
4. perform macro expansion.

Databases required for pass 2. :-

In pass 2 we perform recognize macro call and perform macro expansion

1. Copy file :-
It is a file it contains the output given from pass 1.

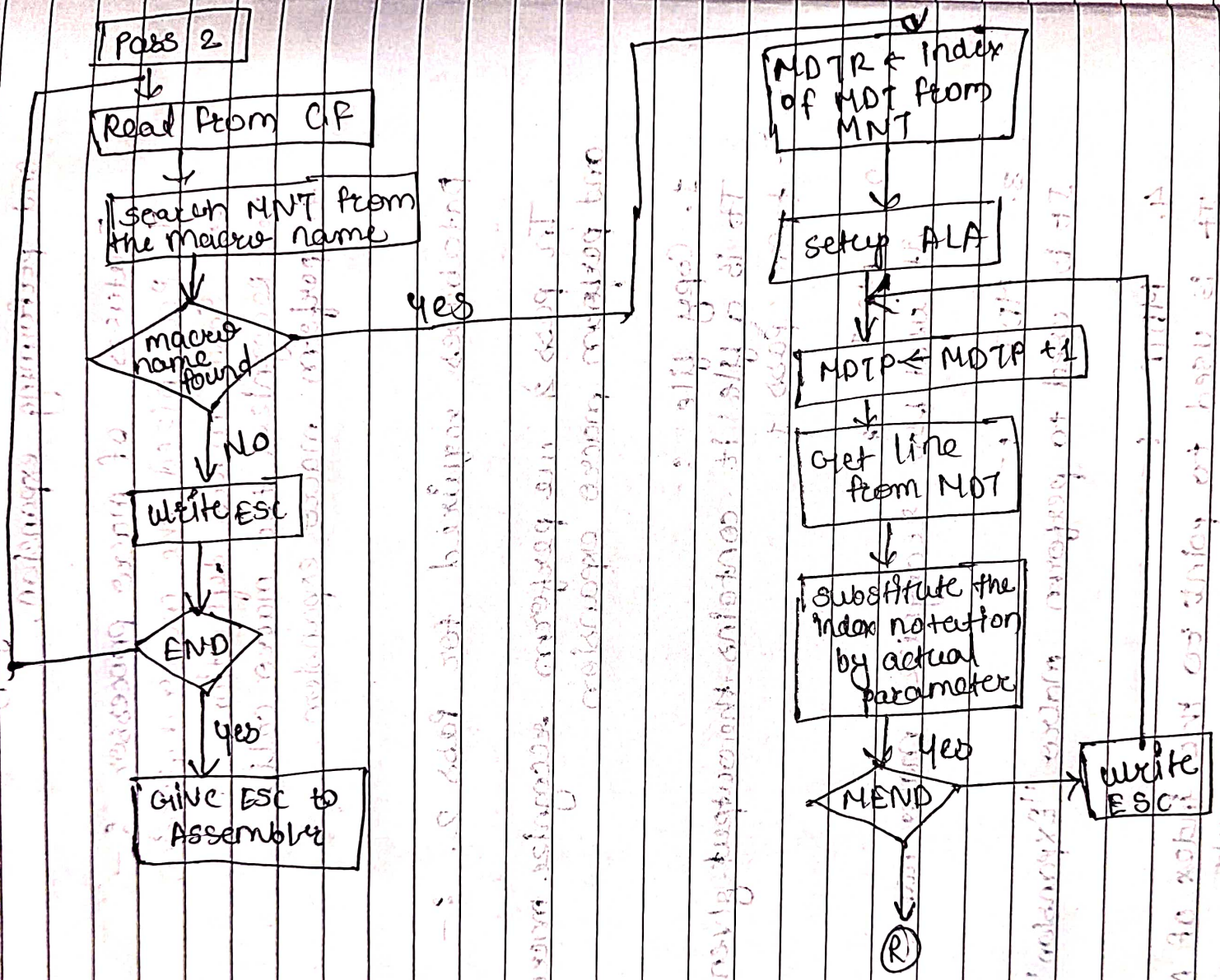
2. MNT :-
It used for recognize macro name

3. MDT :-
It is used to perform macro "Expansion"

4. MDTP :-
It is used to point to the index of MDT. The starting index is given by MNT.

5. ALA :- It is used to replace the index notation by it's actual value.

6. ESC :- It is used to contain the expanded macro call which is given to the assembler for process


Pass 2
Read from CF
Search MNT from the macro name
macro name found
yes
NO
write ESC
END
yes
GIVE ESC to Assembler
MDTR ← index of MDT from MNT
setup ALA
MDTP ← MDTP + 1
Get line from MDT
substitute the index notation by actual parameter
yes
MEND
write ESC
R

ii) A direct linking loader is the most popular loading scheme used. It allows the programmers to use the multiple procedure and data segments giving user a complete choice in referencing data or instruction enclosed in other parts of segments. It provides flexible intersegment referencing accessing ability. Assembler provides following information to loader with respect to each procedure or data segment.

- Segment length
- Records of all the symbol that may be referenced by other symbols and their relative location
- Records of all the symbols, explanation about the address constants, their location in the segment and information about their revised values
- Information about the source codes, machine code translation and the relative address assigned.

design of direct linking loader :-

direct linking loader uses four types of records object file, they are as ESD, TXT, RLD, END.

1) External Symbol directory (ESD)
It combines information about all the symbols that are mentioned in the program but that may be referenced elsewhere.

2) Text Records (TXT)
It contains the information about the actual object code, translated version of the source program.

3) Relocation and Linkage directory (RLD)
RLD cards are used to store those location and address on which the program content is dependent.

4) END Record :-
It specifies the end end of the object file and starting address for execution if the assembled routine is in the main program.