



WINTER- 17 EXAMINATION

Subject Name: System Programming

Model Answer

Subject Code:

17517

Important Instructions to examiners:

- 1) The answers should be examined by key words and not as word-to-word as given in the model answer scheme.
- 2) The model answer and the answer written by candidate may vary but the examiner may try to assess the understanding level of the candidate.
- 3) The language errors such as grammatical, spelling errors should not be given more Importance (Not applicable for subject English and Communication Skills).
- 4) While assessing figures, examiner may give credit for principal components indicated in the figure. The figures drawn by candidate and model answer may vary. The examiner may give credit for any equivalent figure drawn.
- 5) Credits may be given step wise for numerical problems. In some cases, the assumed constant values may vary and there may be some difference in the candidate's answers and model answer.
- 6) In case of some questions credit may be given by judgement on part of examiner of relevant answer based on candidate's understanding.
- 7) For programming language papers, credit may be given to any other program based on equivalent concept.

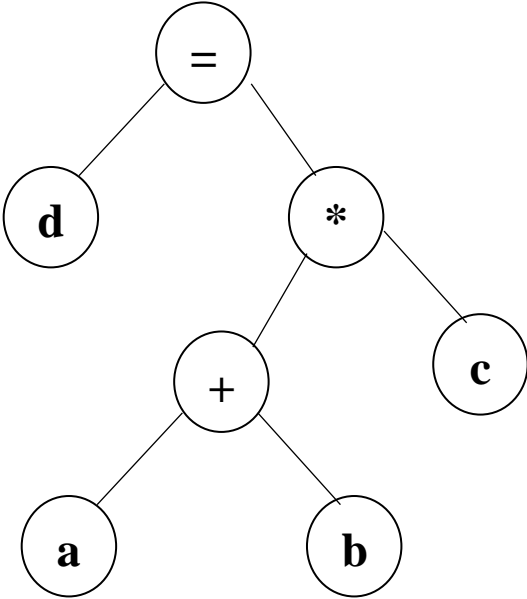
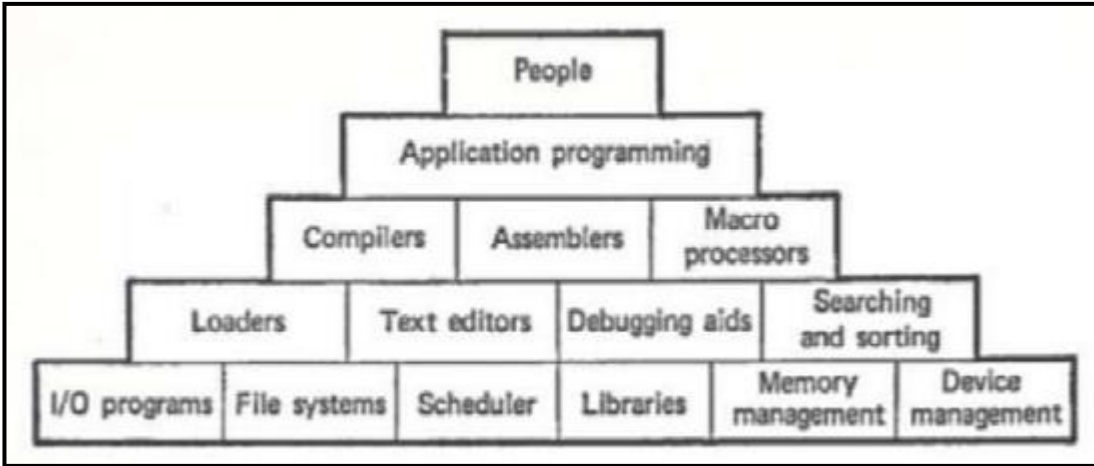
Q. No.	Sub Q. N.	Answer	Marking Scheme
1.	a)	Attempt any three :	Marks 12
	1)	1. Define the following terms : i) Overlays ii) Subroutine linkages.	4M
	Ans:	i) Overlays: 1. Many modern computers use virtual memories that make it possible to run programs larger than physical memory either one program or several programs can be executed even if total size is greater than entire memory available. 2. When a computer does not use virtual memory, running a larger program becomes a problem. One solution is overlays(or chaining). 3. Overlays are based on the facts that many programs can be broken into logical parts such that only one part is needed in memory at any time. 4. The program is divided by the programmer, in to main part (overlay root), that resides in memory during the entire execution and several overlays, (links or segments) that can be called, one at a time, by the root, loaded and executed. 5. The subroutines of a program are needed at different times ii) Subroutine linkage: It is a mechanism for calling another subroutine in an assembly language. The scenario for subroutine linkage.	(Overlays :2marks, Subroutine linkages: 2marks)

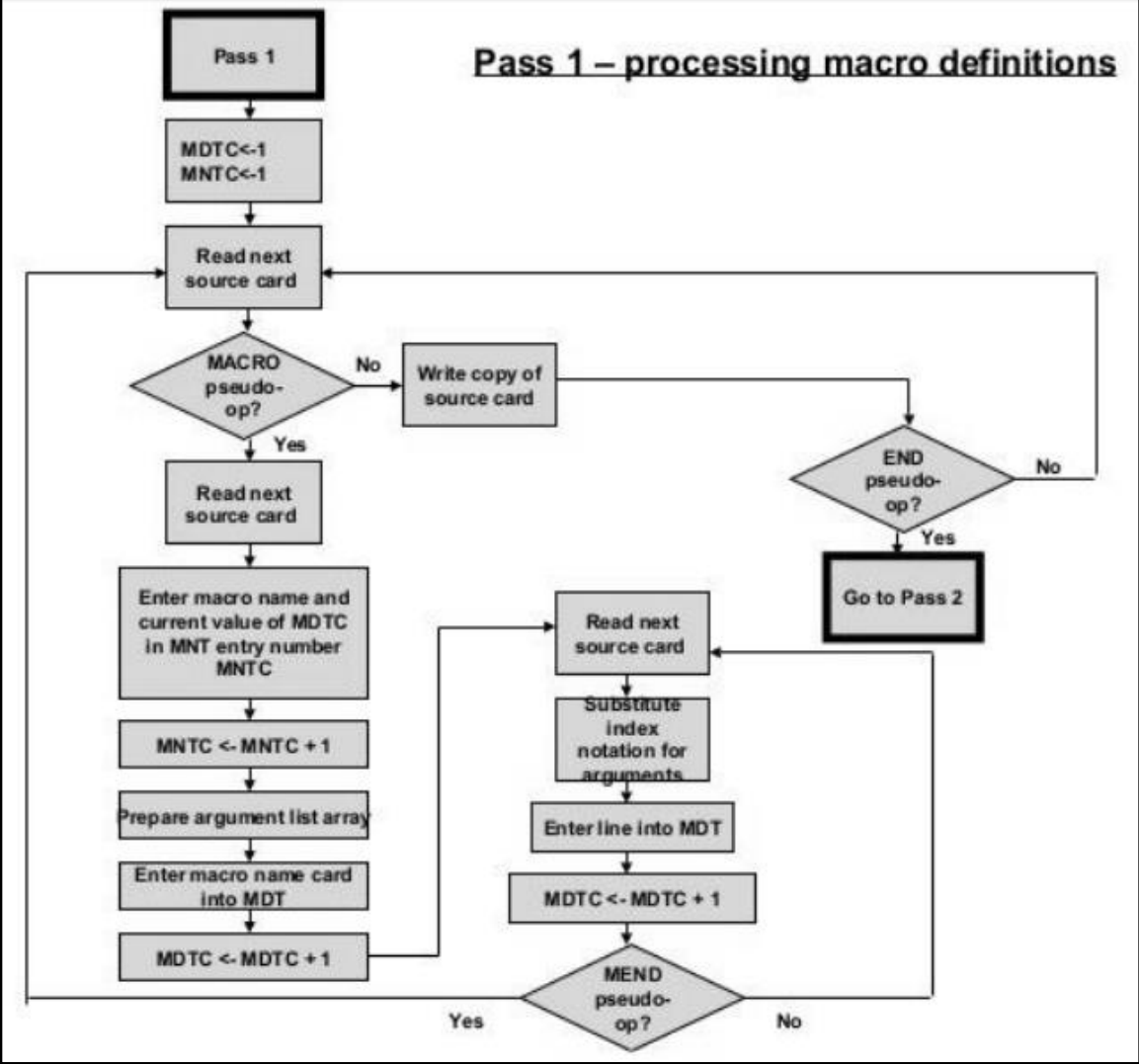


	<ol style="list-style-type: none">1. A main program wishes to transfer to subprogram B.2. The programmer in program A, could write a transfer instruction. Eg (BAL, 14, B) to subprogram B.3. But assembler does not know the value of this transfer reference and will declare it is an error.4. To handle this situation a special mechanism is needed.5. To handle it mechanism is typically implemented with a relocation or direct linking loader. <p>Subroutine linkage uses following special pseudo ops: ENTRY EXTRN</p> <p>It is used to direct or to suggest loader that subroutine followed by ENTRY are defined in this program but they are used in another program. For example: the following sequence of instruction may be a simple calling sequence to another program.</p> <p>MAIN START EXTRN SUBROUT L 15=A(SUBROUT).....CALL SUBROUT BALR 14,15 END</p> <p>The above sequence of instructions first declares SUBROUT as an external variable, that is a variable referenced but not defined in this program. The load(L) instruction loads the address of that variable in to register 15.</p>	
2)	What is operating system? Enlist the features of operating system as a system software.	4M
Ans:	An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers	(Defination: 2 marks, any four features: 2 marks)



	Features of operating system: <ol style="list-style-type: none">1. Assemblers2. Computers, such as FORTRAN, COBOL, and PL/I3. Subroutine libraries, such as SINE, COSINE, SQUARE ROOT.4. Linkage editors and program loaders that bind subroutines together and prepare programs for execution5. Utility routines, such as SRT/MERGE AND TAPE COPY6. Application packages, such as circuit analysis or simulation7. Debugging facilities, such as program tracing and “core dumps”8. Data management and file processing9. Management of system hardware.	
3)	State and explain Binary search method with example.	4M
Ans:	<p>Binary Search Algorithm: A more systematic way of searching an ordered table. This technique uses following steps for searching a keywords from the table.</p> <ol style="list-style-type: none">1. Find the middle entry $(N/2 \text{ or } (N+1)/2)$2. Start at the middle of the table and compare the middle entry with the keyword to be searched.3. The keyword may be equal to, greater than or smaller than the item checked.4. The next action taken for each of these outcomes is as follows <p>If equal, the symbol is found If greater, use the top half of the given table as a new table search If smaller, use the bottom half of the table.</p> <p>Example:</p> <p>The given nos are: 1,3,7,11,15 To search number 11 Indexing the numbers from list [0] up to list[5] Pass 1 Low=0 High = 5 Mid= $(0+5)/2 = 2$ So list[2] = 3 is less than 11 Pass 2 Low= $(\text{Mid}+1)/2$ i.e $(2+1)/2 = 1$ High = 5 Mid= $(1+5)/2 = 6/2 = 3$ So list [3] = 11 and the number if found.</p>	(Description: 2marks, Example: 2 marks)

4)	Draw the output of syntax analysis phase for the string 'd = a + b * c' in the form of syntax tree.	4M
Ans:	 <pre> graph TD A((=)) --- B((d)) A --- C((*)) C --- D((+)) C --- E((c)) D --- F((a)) D --- G((b)) </pre>	(Correct Tree:4 marks)
b)	Attempt any one :	Marks 6
1)	Describe the foundation of system programming.	6M
Ans:	<p>Foundation of System Programming:</p>  <p>System programs e.g. Compilers, loaders, macro processor, operating systems were developed to make computer better adapted to the needs of their users. Compiler is system program that accept people life languages and translate them into machine language. Loaders are system programs that prepare machine language programs for execution.</p>	(Diagram: 3 marks, Description: 3 marks)

		Macro processors allow programmers to use abbreviations. Operating system and file system allows flexible to bring and retrieval of information. The productivity of each computer is heavily dependent upon the effectiveness, efficiency and sophistication of the system programs.	
2)		Draw flowchart of pass-I of two pass macroprocessor.	6M
Ans:		<p style="text-align: center;">Pass 1 – processing macro definitions</p> 	(Correct Flowchart: 6 marks)
2.		Attempt any two:	Marks 16
1)		Explain Address calculation sort with suitable example.	8M
Ans:		<p>This can be one of the fastest types of sorts if enough storage space is available. The sorting is done by transforming the key into an address in the table that “represents” the key.</p> <p>For example if the key were four characters long, one method of calculating the appropriate table address would be to divide the key by the table length in items, multiply by the length is a power of 2, then the division reduces to a shift. This sort would take only N^* (time to calculate address) if it were known that no two keys would be assigned the same address.</p>	(Description: 4 marks, Example: 4 marks)

However, in general. This is not the case and several keys will be reduced to the same address.

Therefore, before putting an item at the calculated address it is necessary to first check whether that location is already occupied. If so, the item is compared with the one that is already there, and a linear search in the correct direction is performed to find the correct place for the new item.

If we are lucky, there will be an empty space in which to put the item in order. Otherwise, it will be necessary to move some previous entries to make room.

Data number	=	1	2	3	4	5	6	7	8	9	10	11	12
Data	=	19	13	05	27	01	26	31	16	02	09	11	21
Calculated address	=	6	4	1	9	0	8	10	5	9	3	3	7
Table	=												
0		---	---	---	---	01	01	01	01	*01	01	01	01
1		---	---	---	---	---	---	---	---	02	02	02	02
2		---	---	---	---	---	---	---	---	05	*05	05	05
3		---	---	---	---	---	---	---	---	09	*09	09	09
4		---	13	13	13	13	13	13	13	13	13	11	11
5		---	---	---	---	---	---	---	16	16	16	13	13
6		19	19	19	19	19	19	19	19	19	19	16	16
7		---	---	---	---	---	---	---	---	---	---	19	*19
8		---	---	---	---	26	26	26	26	26	26	26	21
9		---	---	---	27	27	27	27	27	27	27	27	26
10		---	---	---	---	---	31	31	31	31	31	31	27
11		---	---	---	---	---	---	---	---	---	---	---	31

Figure: Example of address calculation sort

The table is of size 12; since it is known that maximum key is less than 36, the address transformation is to divide the key by 3 and take the integer part. A "*" indicates a conflict between keys, and the arrow indicates when a move is necessary and in which direction. The associated addresses calculated are given in the second row.

2) Explain format of databases in Assembler.

8M

- Ans:** Pass 1 data bases
- Input source program
 - A LC to keep track of each instruction location
 - A MOT (Machine Operation Table)

(Any 8 Databases : 1 mark each)

←----- 6-bytes per entry ----->				
Mnemonic Op-code (4-bytes) (characters)	Binary Op-code (1-byte) (hexadecimal)	Instruction Length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALRb"	5E	10	0001	
"ARbb"	1E	01	000	
...	1A	01	000	
"MVCb"	D2	11	100	
...	

b~ represent the character "blank"

Codes:

Instructions length

01 = 1 half-words = 2bytes

10 = 2 half-words = 4bytes

11 = 3 half-words = 6bytes

Instruction format

000=RR

001=RX

010 = RS

011 = SI

100 = SS

Machine – Op Table (MOT) for pass1 and pass 2

- A POT (Pseudo operation Table)

Pseudo-op (5-bytes) (characters)	Address of routine to process pseudo-op (3 Bytes = 24 bit Address)
"DROPb"	P1DROP
"ENDbb"	P1END
"EQUbb"	P1EQU
"START"	P1START
"USING"	P1USING

These are presumably labels of
routines in pass 1; the Table will
actually contain the physical addresses

Pseudo – Op Table (POT) for pass1 and pass 2

- A ST (Symbol Table)

← 14-bytes per entry →

Symbol (8 Bytes) (Character)	Value (4 Bytes) (Hexadecimal)	Length (1 Byte) (Hexadecimal)	Relocation (1 Byte) (Character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

Symbol Table:

- A LT (Literal Table)

Variable tables

Symbol Table

Symbol	Value	Length	Relocation
PRGAM	0	1	R
AC	2	1	A
INDEX	3	1	A
TOTAL	4	1	A
DATABASE	13	1	A
SETUP	6	1	R
LOOP	12	4	R
SAVE	64	4	R
DATAAREA	8064	1	R
DATA1	8064	4	R

Literal Table

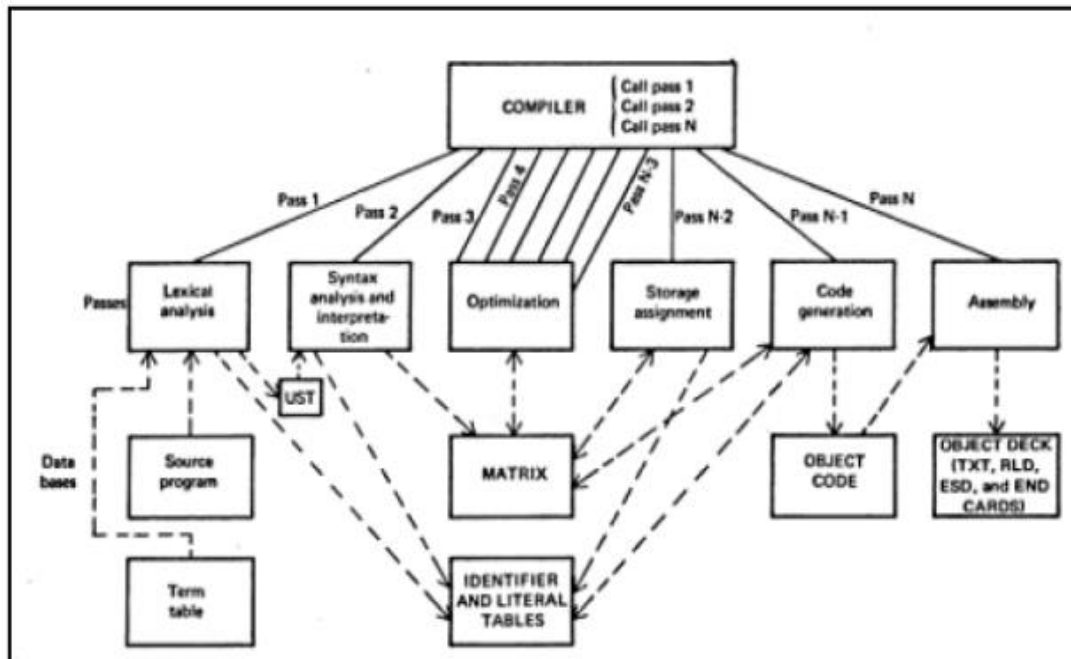
A(DATA1)	48	4	R
F'5'	52	4	R
F'4'	56	4	R
F,8000	60	4	R

- A copy of the input to be used by pass 2

Pass 2 databases

- Copy of source program input to pass 1
- LC: Same as Pass I
- MOT: Same as Pass I
- POT: Same as Pass I
- ST: Same as Pass I

	<ul style="list-style-type: none"> BT (Base table) <div style="border: 1px solid black; padding: 10px; margin: 10px 0;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 5%;"></th> <th style="width: 40%;">Availability Indicator (1-byte) (character)</th> <th style="width: 55%;">Designated relative – address contents of base register (3-bytes = 24-bit address) (hexadecimal)</th> </tr> </thead> <tbody> <tr><td>1</td><td>"N"</td><td>-</td></tr> <tr><td>2</td><td>"N"</td><td>-</td></tr> <tr><td>⋮</td><td></td><td></td></tr> <tr><td>14</td><td>"N"</td><td>-</td></tr> <tr><td>15</td><td>"Y"</td><td>00 00 00</td></tr> </tbody> </table> <p style="text-align: right; margin-top: -20px;">↑ 15 entries ↓</p> <p>Code= Availability Y – register specified in USING pseudo-op N – register never specified in USING pseudo-op or subsequently made unavailable by the DROP pseudo-op</p> </div> <ul style="list-style-type: none"> Output in machine code to be needed by the loader 		Availability Indicator (1-byte) (character)	Designated relative – address contents of base register (3-bytes = 24-bit address) (hexadecimal)	1	"N"	-	2	"N"	-	⋮			14	"N"	-	15	"Y"	00 00 00	
	Availability Indicator (1-byte) (character)	Designated relative – address contents of base register (3-bytes = 24-bit address) (hexadecimal)																		
1	"N"	-																		
2	"N"	-																		
⋮																				
14	"N"	-																		
15	"Y"	00 00 00																		
3)	Explain with flowchart overview of passes of compiler.	8M																		
Ans:	<p>Description: Passes is a logical execution of the compilation process.</p> <ol style="list-style-type: none"> Pass 1 this pass is corresponds to lexical analysis phase. This pass scans the source code and creates an identifier, literal and uniform symbol table. Pass 2 corresponds to syntactic and interpretation phase. It scans the uniform symbol table, produces the matrix and place information about identifier into the identifier table. Passes 3 to N-3 corresponds to the optimization phase. Each separate type of optimization may require several passes over the matrix. Pass N-2 corresponds to the storage assignment phase. This is a pass over the identifier and literal tables rather than program itself. Pass N-1 corresponds to the code generation phase. It scans the matrix and creates the first version of the object deck. Pass N corresponds to the assembly phase. It resolves the symbolic addresses and creates information for the loader. 	<p>(Description: 4 marks, Flowchart / Diagram: 4 marks)</p>																		



3.	Attempt any four:	Marks 16
1)	List and explain four component of system software.	4M
Ans:	<p>Components of system software are:</p> <ol style="list-style-type: none"> 1. Assembler 2. Macros 3. Loader 4. Linker 5. Compiler. <p>1. Assembler :It is a language translator that takes as input assembly language program (ALP) and generates its machine language equivalent along with information required by the loader. ALP Assembler→ Machine language equivalent + Information required by the loader</p> <p>2. Macros: The assembly language programmer often finds that certain set of instructions get repeated often in the code. Instead of repeating the set of instructions the programmer can take the advantage of macro facility where macro is defined to be as “Single line abbreviation for a group of instructions”.</p> <p>The template for designing a macro is as follows</p> <pre> MACRO Start of definition Macro Name ----- ----- MEND End of def. </pre>	(Any four component : 1 mark each)



		<p>3.Loader: Loader is a system program which is responsible for preparing the object programs for execution and start the execution. Functions of loader a. Allocation b. Linking c. Relocation d. Loading Allocation: Allocate the space in the memory where the object programs can be loaded for execution. Linking: Resolving external symbol reference Relocation: Adjust the address sensitive instructions to the allocated space. Loading: Placing the object program in the memory into the allocated space.</p> <p>4. Linker: A linker which is also called binder or link editor, is a program that combines object modules together to form a program that can be executed. Modules are parts of a program.</p> <p>5. Compiler: Compiler is a language translator that takes as input the source program(Higher level program) and generates the target program (Assembly language program or machine language program)</p> <p>Source Program → Compiler → Target program</p>										
	2)	Compare shell sort and Radix Exchange sort on the basis of space and time complexity.	4M									
	Ans:	<table><tr><th>Sort</th><th>Time Complexity</th><th>Space Complexity</th></tr><tr><td>Shell Sort</td><td>O (n²) to O(n* logn)</td><td>O(n)</td></tr><tr><td>Radix Exchange Sort</td><td>O(nd)</td><td>O(k)</td></tr></table>	Sort	Time Complexity	Space Complexity	Shell Sort	O (n ²) to O(n* logn)	O(n)	Radix Exchange Sort	O(nd)	O(k)	(Correct time complexity :2marks, Correct space complexity :2marks)
Sort	Time Complexity	Space Complexity										
Shell Sort	O (n ²) to O(n* logn)	O(n)										
Radix Exchange Sort	O(nd)	O(k)										
	3)	Explain in detail machine dependent optimization.	4M									
	Ans:	Two types of optimization is performed by compiler, machine dependent and machine independent.	(Machine dependent optimization :4 marks)									

Machine dependent optimization is so intimately related to instruction that the generated. It was incorporated into the code generation phase. Whereas Machine independent optimization was done in a separate optimization phase.

Machine- independent optimization:

o When a subexpression occurs in a same statement more than once, we can delete all duplicate matrix entries and modify all references to the deleted entry so that they refer to the remaining copy of that subexpression as shown in following figure.

operator	Operand1	Operand2	Matrix entries	operator	Operand1	Operand2	Matrix entries
1 -	START	FINISH	M1	1 -	START	FINISH	M1
2 *	RATE	M1	M2	2 *	RATE	M1	M2
3 *	2	RATE	M3	3 *	2	RATE	M3
4 -	START	FINISH	M4	4 -	START	FINISH	M4
5 -	M4	100	M5	5 -	M4	100	M5
6 *	M3	M5	M6	6 *	M3	M5	M6
7 +	M2	M6	M7	7 +	M2	M6	M7
8 =	COST	M7		8 =	COST	M7	

Matrix with common subexpressions Matrix with common subexpressions

Machine dependent optimization:

- If we optimize register usage in the matrix, it becomes machine – dependent optimization.
- Following figure depicts the matrix that we previously optimized by eliminating a common subexpression (M4).
- Next to each matrix entry is a code generated using the operators.
- The third column is even better code in that it uses less storage and is faster due to a more appropriate mix of instructions.
- This example of machine-dependent optimization has reduced both the memory space needed for the program and the execution time of the object program by a factor of 2.
- Machine dependent optimization is typically done while generating code.



Optimized Matrix				First try		Improved code				
				L	1, START	L	1, START			
1	*	START	FINISH	S	1, FINISH	S	1, FINISH	M1	→	R1
				ST	1, M1					
				L	1, RATE	L	3, RATE			
2	*	RATE	M1	M	0, M1	MR	2, 1	M2	→	R3
				ST	1, M2					
				L	1, =F'2'	L	5, = F'2'			
3	*	2	RATE	M	0, RATE	M	4, RATE	M3	→	R5
				ST	1, M3					
4										
				L	1, M1					
5	*	M1	100	S	1, =F'100'	S	1, =F'100'	M5	→	R1
				ST	1, M5					
				L	1, M3	LR	7, 5			
6	*	M3	M5	M	0, M5	MR	6, 1	M6	→	R7
				ST	1, M6					
				L	1, M2					
7	+	M2	M6	A	1, M6	AR	3, 7	M7	→	R3
					1, M7					
				L	1, M7	ST	3, COST			
8	=	M7	COST	ST	1, COST					

4) Explain four function performed by loader.

4M

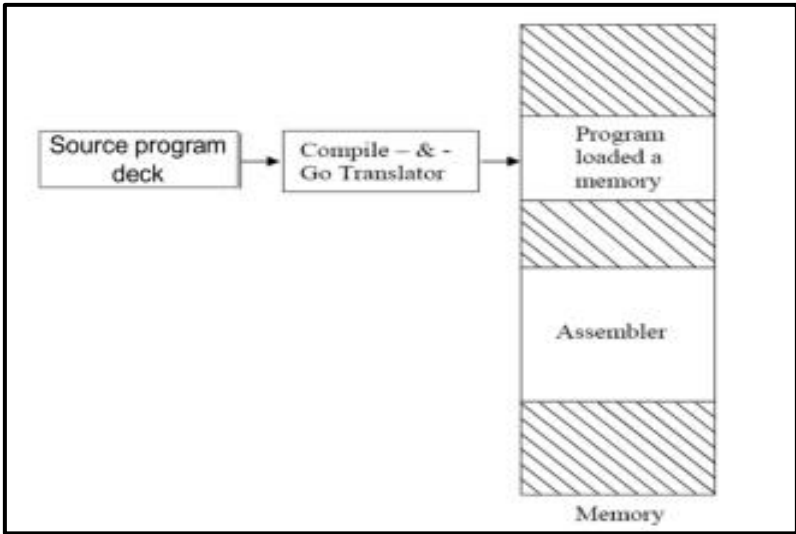
Ans: Loader Function : The loader performs the following functions :

- 1) Allocation - The loader determines and allocates the required memory space for the program to execute properly.
- 2) Linking -- The loader analyses and resolve the symbolic references made in the object modules.
- 3) Relocation - The loader maps and relocates the address references to correspond to the newly allocated memory space during execution.
- 4) Loading - The loader actually loads the machine code corresponding to the object modules into the allocated memory space and makes the program ready to execute.

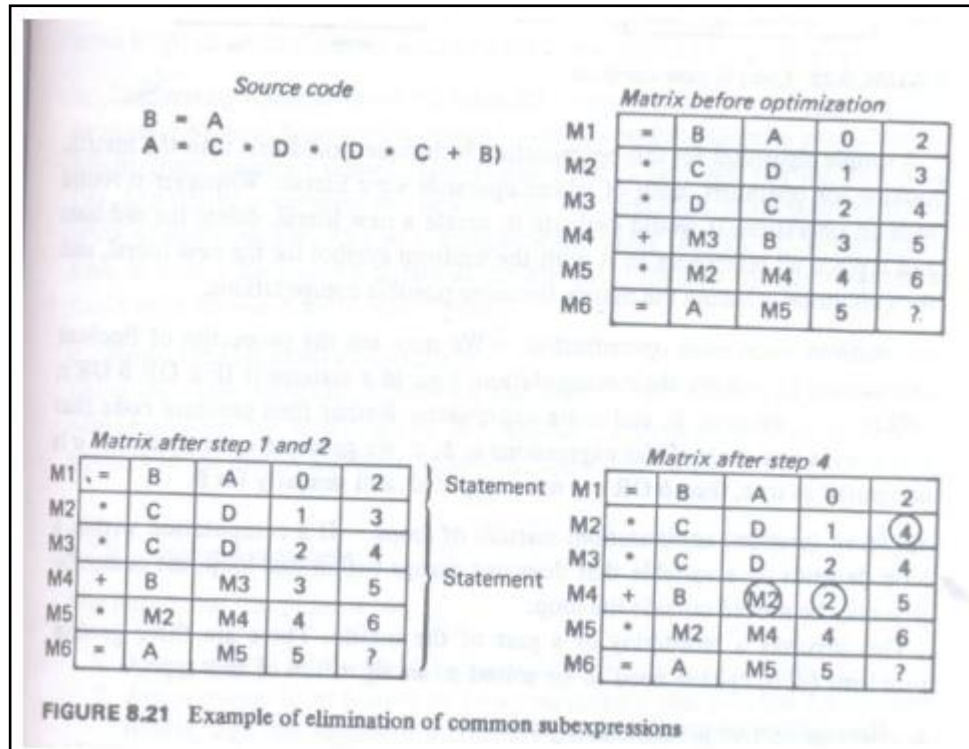
(Four functions :1 mark each)



5)	Draw the parse tree for the string 'acddf' using top down parsing approach.	4M
Ans:	<p>String is "acddf" Assume: $S \rightarrow xyz \mid aBC$ $B \rightarrow c \mid cd$ $C \rightarrow eg \mid df$ Steps Assertion 1 : acddf matches S Assertion 2: acddf matches xyz: Assertion is false. Try another. Assertion 2 : acddf matches aBCi.ecddf matches BC: Assertion 3 : cddf matches cCi.eddf matches C: Assertion 4 : ddf matches eg: False. Assertion 4 : ddf matches df: False. Assertion 3 is false. Try another. Assertion 3 : cddf matches cdCi.edf matches C: Assertion 4 : df matches eg: False. Assertion 4 : df matches df: Assertion 4 is true. Assertion 3 is true. Assertion 2 is true. Assertion 1 is true.</p> <div><p>Step 1 Step 2 Step 3 Step 4 Step 5 Final parse tree</p></div>	(Correct parse tree: 4marks)

4.	a)	Attempt any three :	Marks 12
	1)	Explain “compile and Go” loader scheme.	4M
	Ans:	<p>“Compile and go” loader: The instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. The typical example is WATFOR-77, it's a FORTRAN compiler which uses such “load and go” scheme. This loading scheme is also called as <u>“assemble and go”</u>.</p> <p>Advantages:-</p> <ol style="list-style-type: none"> 1. It is very easy to design and implementation. 2. Relocation can be perform by translator itself. 3. No object files are required. 4. It is suitable for experimental program language like Basic. <p>Disadvantages:-</p> <ol style="list-style-type: none"> 1. For execute program it is necessary to compile a program every time. 2. It is very difficult to handle the multiple modules (Linking problem) <div style="text-align: center;">  <p>Figure: Compile & Go Loader</p> </div>	<p>(Explanati on:3marks , Diagram: 1mark)</p>
	2)	Explain any four optimization technique uses by compiler.	4M
	Ans:	<p>The possible algorithm for four optimization techniques are as follows:-</p> <ol style="list-style-type: none"> 1) Elimination of common sub expression 2) Compile time compute. 3) Boolean expression optimization. 4) Move invariant computations outside of loops. <p>1) Elimination of common sub expression: -The elimination of duplicate matrix entries can result in a more can use and efficient object program. The common subexpression must be identical and must be in the same statement.</p>	<p>(Each technique: 1 mark)</p>

- i. The elimination algorithm is as follows:-
 - ii. Place the matrix in a form so that common subexpression can be recognized.
 - iii. Recognize two subexpressions as being equivalent.
 - iv. Eliminate one of them.
 - v. After the rest of the matrix to reflect the elimination of this entry.
- For example:-



2. Compile time compute:- Doing computation involving constants at compile time save both space and execution time for the object program.

The algorithm for this optimization is as follows:-

- i. Scan the matrix.
- ii. Look for operators, both of whose operands were literals.
- iii. When it found such an operation it would evaluate it, create new literal, delete old line
- iv. Replace all references to it with the uniform symbol for the new literal.
- v. Continue scanning the matrix for more possible computation.

For e.g.-

For e.g.- $A = 2 * 276 / 92 * B$

The compile time computation would be

Matrix Before optimization

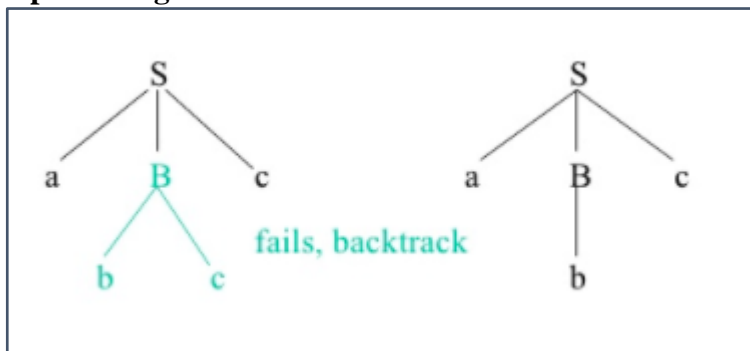
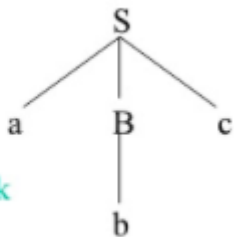
M ₁	*	2	276
M ₂	/	M ₁	92
M ₃	*	M ₂	B
M ₄	=	A	M ₃

Matrix After optimization

M ₁			
M ₂			
M ₃	*	6	B
M ₄	=	A	M ₃

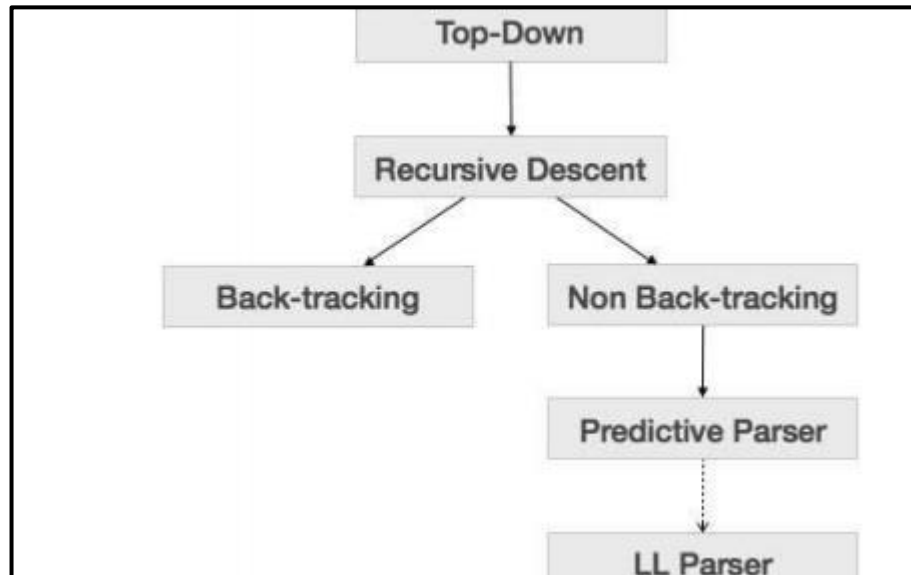


	<p>3) Boolean expression optimization:- We may use the properties of boolean expression to shorten their computation. e.g. In a statement If a OR b OR c, Then when a, b & c are expression rather than generate code that will always test each expression a, b, c. We generate code so that if a computed as true, then b OR c is not computed, and similarly for b.</p> <p>3) Move invariant computation outside of loops:- If computation within a loop depends on a variable that does not change within that loop, then computation may be moved outside the loop. This requires a reordering of a part of the matrix. There are 3 general problems that need to be solved in an algorithm.</p> <ol style="list-style-type: none">1. Recognition of invariant computation.2. Discovering where to move the invariant computation.3. Moving the invariant computation. <p>Original Code is:- For y=0 to height-1 For x=0 to width-1 i=y*width+x Process i Next x Next y here y*width is loop invariant not change in inner loop Modified code is:- For y=0 to height-1 temp= y*width For x=0 to width-1 i=temp+x Process i Next x Next y</p>	
3)	Discuss memory allocation scheme used in compiler.	4M
Ans:	<p>Storage Assignment: The purpose of this phase is as follows:-</p> <ol style="list-style-type: none">1. Assign storage to all variables referenced in the source program2. Assign storage to all temporary locations that are necessary for intermediate result, e.g. the result of matrix lines. These storage references were reserved by the interpretation phase and does not appear in the source code.3. Assign storage to literals.4. Ensure that the storage is allocated and appropriate locations are initialized (literals and any variables with the initial attribute). <p>Storage allocation phase does the following four steps:</p> <ol style="list-style-type: none">1. Update the location counter with any necessary boundary alignment.2. Assign the current value of the location counter to the address field of the variable.3. Calculate the length of the storage needed by the variable.4. Update the location counter by adding this length to it.	(Explanation: 4 marks)

4)	Explain predictive parsing with example.	4M	
Ans:	<p>Predictive Parser: Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking. To accomplish its tasks; the predictive parser uses a look ahead pointer, which points to the next input symbols. To make the parser backtracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar. Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol \$to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.</p> <p>Exapmle:-</p> <p>Grammer:</p> <p style="margin-left: 100px;">$S \rightarrow aBc$ $B \rightarrow bc b$</p> <p>Input String : abc</p> <div><div></div><div><p>Fails, backtrack</p></div><div></div><div><p>correct</p></div></div>	(Description n:2marks, Example:2 marks)	
b)	Attempt any one :	Marks 6	
1)	Explain four basic task of macroprocessor.	6M	
	<p>The 4 basic task of Macro processor is as follows:-</p> <ol style="list-style-type: none">1) Recognize the macro definitions.2) Save the Macro definition.3) Recognize the Macro calls.4) Perform Macro Expansion. <p>1) Recognize the Macro definitions:- A microprocessor must recognize macro definitions identified by the MACRO and MEND pseudo-ops. When MACROS and MENDS are nested, the macro processor must recognize the nesting and correctly match the last or outer MEND with the first MACRO.</p>	(List:2 marks, Each task descriptio n:1 mark)	



		<div><div>MDT</div><div>Macro Definition Table</div><table><tr><td>& LAB</td><td>INCR</td><td>& ARG1,&ARG2,&ARG3</td></tr><tr><td>#0</td><td>A</td><td>1, #1</td></tr><tr><td></td><td>A</td><td>2,#2</td></tr><tr><td></td><td>A</td><td>3,#3</td></tr><tr><td></td><td>MEND</td><td></td></tr></table></div> <div>2) Save the Macro definition:- The pre-processor must save the macro instructions definitions that can be later required for expanding macro calls.</div> <div><div>MNT</div><div>Macro Name Table</div><table><tr><td>1</td><td>"INCRbbbb"</td><td>15</td></tr><tr><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>.</td></tr></table></div> <div>3) Recognize the Macro calls:- The processor must recognize macro call that appear as operation mnemonics. The macro calls appear as operation mnemonics in a program.</div> <div><div>ALA</div><table><tr><td>Index</td><td>Argument</td></tr><tr><td>0</td><td>"bbbbbbbb" (all Blank)</td></tr><tr><td>1</td><td>"Data3bbb"</td></tr><tr><td>2</td><td>"Data2bbb"</td></tr><tr><td>3</td><td>"Data1bbb"</td></tr></table></div> <div>4) Perform Macro Expansion:- The processor must substitute for macro definition arguments the corresponding arguments from a macro call, the resulting symbolic text is then substituted for the macro call.</div>	& LAB	INCR	& ARG1,&ARG2,&ARG3	#0	A	1, #1		A	2,#2		A	3,#3		MEND		1	"INCRbbbb"	15	Index	Argument	0	"bbbbbbbb" (all Blank)	1	"Data3bbb"	2	"Data2bbb"	3	"Data1bbb"	
& LAB	INCR	& ARG1,&ARG2,&ARG3																																									
#0	A	1, #1																																									
	A	2,#2																																									
	A	3,#3																																									
	MEND																																										
1	"INCRbbbb"	15																																									
.	.	.																																									
.	.	.																																									
.	.	.																																									
.	.	.																																									
Index	Argument																																										
0	"bbbbbbbb" (all Blank)																																										
1	"Data3bbb"																																										
2	"Data2bbb"																																										
3	"Data1bbb"																																										
2)	<div>Explain following parsing technique in detail.</div> <div>i) Top-down parsing</div> <div>ii) Bottom up parsing.</div>	6M																																									
Ans:	<div>Top-down Parser: When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.</div> <div>The types of top-down parsing are depicted Below:</div>	(Top-down parsing: 3marks, Bottom up parsing: 3marks)																																									



Recursive Descent Parsing: Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Backtracking: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production. Top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes.

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).

The following example of CFG:

$S \rightarrow rXd|rZd$

$X \rightarrow oa|ea$

$Z \rightarrow ai$

For an input string: read, a top-down parser, will behave like this: It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S ($S \rightarrow rXd$) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left ($X \rightarrow oa$). It does not match with the next input symbol. So the

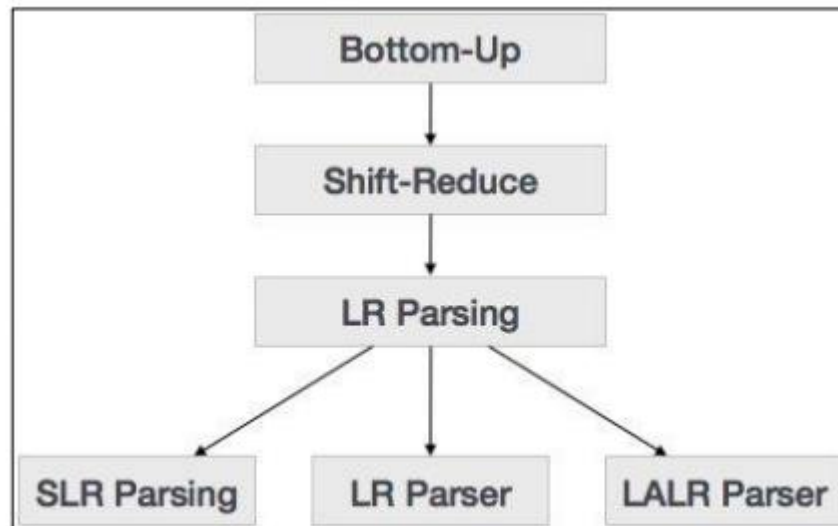
top-down parser backtracks to obtain the next production rule of X , ($X \rightarrow ea$). Now the parser matches all the input letters in an ordered manner. The string is accepted.

Predictive Parser: Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking. To accomplish its tasks; the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser backtracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as $LL(k)$ grammar. Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol $\$$ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

LL Parser: An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven. LL parser is denoted as $LL(k)$. The first L in $LL(k)$ is parsing the input from left to right, the second L in $LL(k)$ stands for left-most derivation and k itself represents the number of look ahead. Generally $k = 1$, so $LL(k)$ may also be written as $LL(1)$.

Bottom-up Parser:

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step. **Shift step:** The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

Reduce step : When the parser finds a complete grammar rule RHS and replaces it to LHS, it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.



		<p>LR Parser</p> <p>The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.</p> <p>There are three widely used algorithms available for constructing an LR parser:</p> <p>SLR1 – Simple LR Parser:</p> <ul style="list-style-type: none">o Works on smallest class of grammaro Few number of states, hence very small tableo Simple and fast construction <p>LR1 – LR Parser:</p> <ul style="list-style-type: none">o Works on complete set of LR1 Grammaro Generates large table and large number of stateso Slow construction <p>LALR1 – Look-Ahead LR Parser:</p> <ul style="list-style-type: none">o Works on intermediate size of grammaro Number of states are same as in SLR1	
5.		Attempt any two :	Marks 16
	1)	Explain design of direct linking loader.	8M
	Ans:	<p>{**Note: Flow charts are optional**}</p> <p>Direct Linking Loader generates relocatable loader and is most popular load scheme presently used.</p> <p>Direct Linking loader It allow programmer to used multiple procedure and multiple data segments. This is possible due to assembler provides following information....</p> <ul style="list-style-type: none">A.The length of segment.B.List of symbol, define in one segment and call in other segment.C.List of symbol, not define in segment but call in segment.D.Information about address located in segment.E.Machine code of program and relative address assigned. <p>Step 1. Specification of problems Step 2. Specification of data structure Step 3. Format of data structure Step 4: Algorithm</p> <p>1. Specification of problems : There will be 4 different card format in a object desk.</p> <ol style="list-style-type: none">1. ESD: This card contain the information necessary for the building symbol table.2. TXT: This card contain instruction and data card called “text ” of programe.3. RLD: Relocation and linking directory cards4. END: End card	(Each step: 2marks)

2. Specification of data structure :

Pass-1

1. Input object deck
2. IPLA-Initial Program Load Address supplied by program's or OS.
3. PLA-Program load address counter, to keep track of each segment
4. GEST-Global External Symbol Table, to store each external symbol & its corresponding core add.
5. A copy of program to be used by Pass-2.
6. LOAD MAP-Printed listing, that specifies each external symbol & it's assigned value.

Pass-2

1. Copy of object program inputted by Pass-1.
2. IPLA-Initial Program Load Address supplied by program's or OS.
3. PLA-Program load address counter, to keep track of each segment
4. GEST-Global External Symbol Table, to store each external symbol & its corresponding core add.
5. LESA-Local External Symbol Array, which is used establish correspondences between
Absolute address value. ESD Id and

3. Format of Data bases :

Format of databases is to specify the format and content of following data bases.

1. ESD
2. TXT
3. RLD
4. END
5. GEST
6. LESA

4. Algorithm :

Following are the steps of an Algorithm for direct linking loader

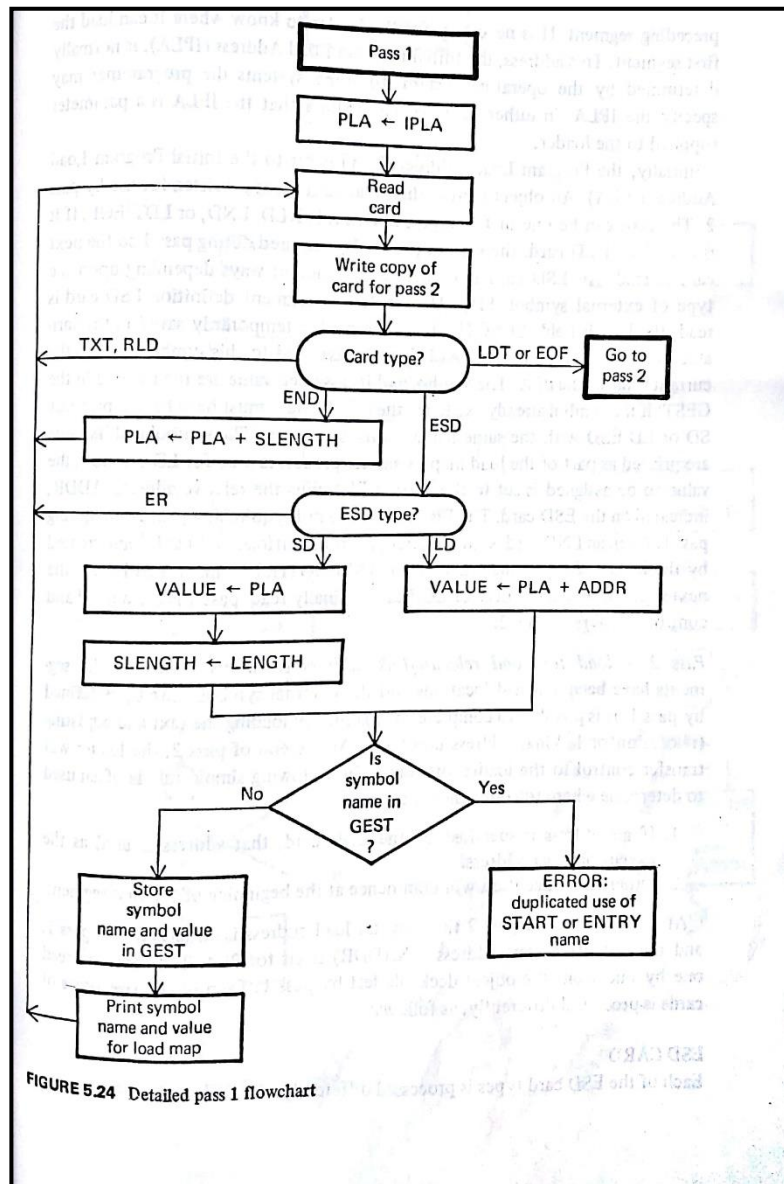
Pass1 :

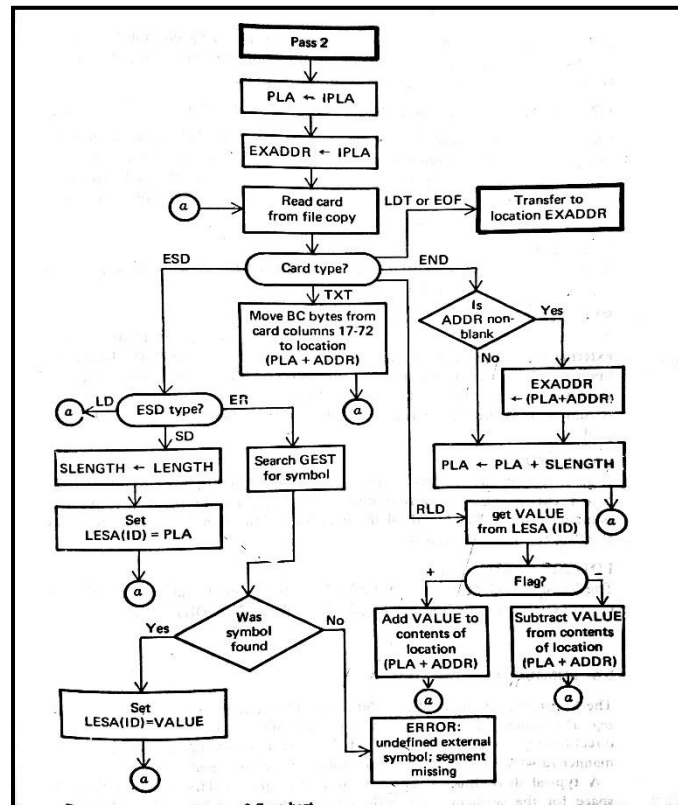
- a. Allocate Segments
 - i. Initial Program Load Address (IPLA)
 - ii. Assign each segment the next table location after the preceding segment.
- b. Define Symbols
 - i. SD
 - ii. LD
 - iii. ER

Pass2 :

- a. ESD record types is processed differently.
- b. TXT is copied from the record to the relocated core location (PLA + ADDR).
- c. RLD value to be used for relocation and linking is extracted from the GEST.

- d. The relocated address of the address constant is the sum of the PLA and the ADDR field specified on the RLD record.
- e. END execution start address is relocated by the PLA
- f. The loader transfers control to the loaded program at the address specified by current contents of the execution

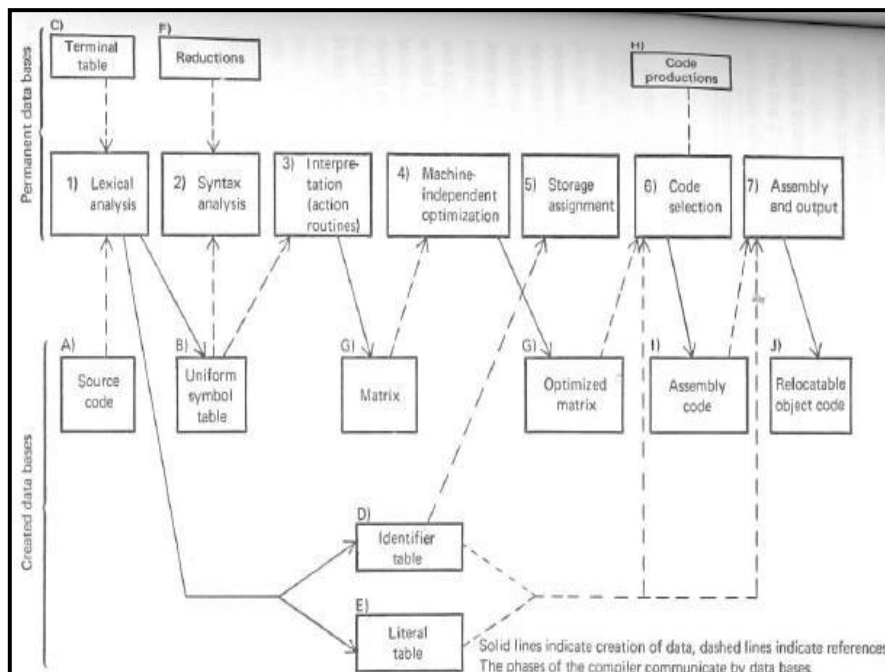




2) Draw and explain phases of compiler in detail.

8M

Ans:



(Diagram :3marks,
Discription:5marks)



1) Lexical Phase:-

- Its main task is to read the source program and if the elements of the program are correct it generates as output a sequence of tokens that the parser uses for syntax analysis.
- The reading or parsing of source program is called as scanning of the source program.
- It recognizes keywords, operators and identifiers, integers, floating point numbers, character strings and other similar items that form the source program.
- The lexical analyzer collects information about tokens in to their associated attributes.

2) Syntax Phase:-

- In this phase the compiler must recognize the phases (syntactic construction); each phrase is a semantic entry and is a string of tokens that has meaning, and 2nd Interpret the meaning of the constructions.
- Syntactic analysis also notes syntax errors and assure some sort of recovery. Once the syntax of statement is correct, the second step is to interpret the meaning (semantic). There are many ways of recognizing the basic constructs and interpreting the meaning.
- Syntax analysis uses a rule (reductions) which specifies the syntax form of source language.
- This reduction defines the basic syntax construction and appropriate compiler routine (action routine) to be executed when a construction is recognized.
- The action routine interprets the meaning and generates either code or intermediate form of construction.

3) Interpretation Phase:-

- This phase is typically a routine that are called when a construct is recognized. The purpose of these routines is to on intermediate form of source program and adds information to identifier table.

4) Code optimization Phase:-

- Two types of optimization is performed by compiler, machine dependent and machine independent. Machine dependent optimization is so intimately related to instruction that the generated. It was incorporated into the code generation phase. Where Machine independent optimization is was done in a separate optimization phase.

5) Storage Assignment:-

The purpose of this phase is as follows: -

- Assign storage to all variables referenced in the source program.
- Assign storage to all temporary locations that are necessary for intermediate results.
- Assign storage to literals.
- Ensure that storage is allocated and appropriate locations are initialized.

6) Code generation:-

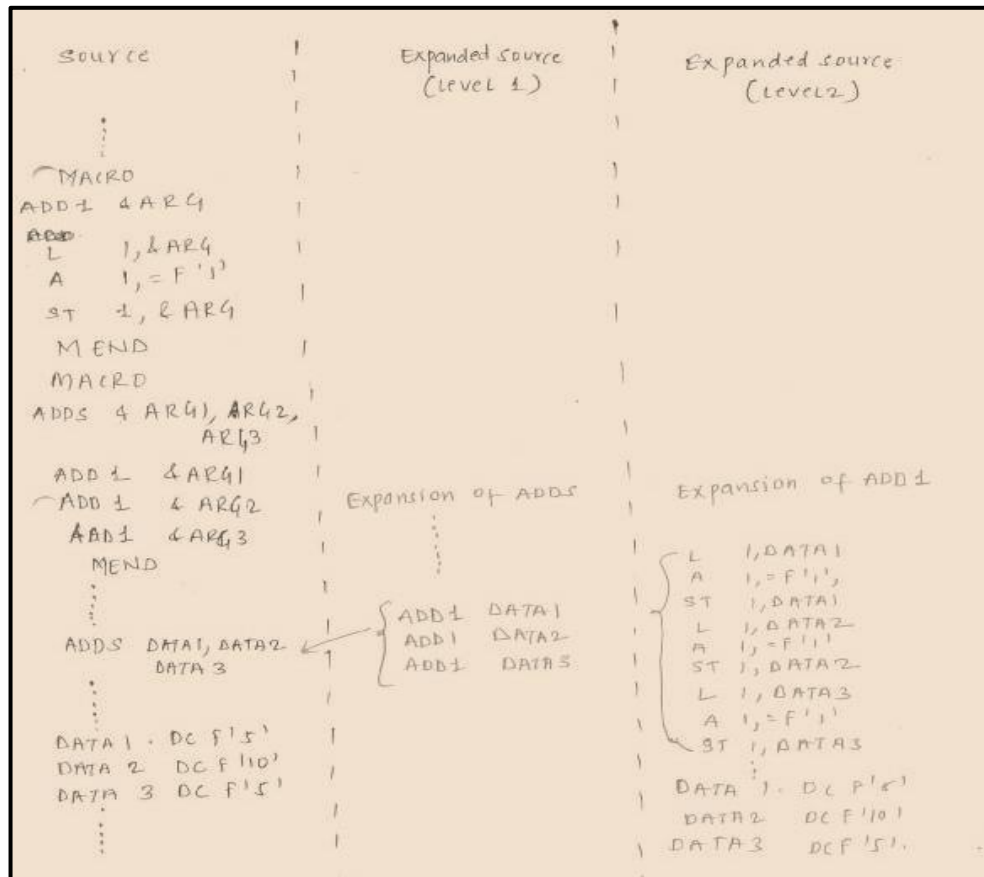
- This phase produce a program which can be in Assembly or



		<p>machine language.</p> <ul style="list-style-type: none">• This phase has matrix as input.• It uses the code production in the matrix to produce code.• <p>It also references the identifier table in order to generate address & code conversion.</p> <p>7) Assembly phase:-</p> <ul style="list-style-type: none">• The compiler has to generate the machine language code for computer to understand.• The task to be done is as follows:-<ul style="list-style-type: none">○ Generating code Resolving all references.○ Defining all labels.○ Resolving literals and symbolic table.	
	3)	Explain general design procedure of Assembler.	8M
	Ans:	<p>Following are six steps for the General Design procedure of an Assembler.</p> <ol style="list-style-type: none">1. Specify the problem2. Specify data structures3. Define format of data structures4. Specify algorithm5. Look for modularity6. Repeat 1 to 5 on modules <p>1. Specify the problem. This includes translating assembly language program into machine language program using two passes of assembler. Purpose of two passes of assembler are to determine length of instruction, keep track of location counter, remember values of symbol, process some pseudo ops, lookup values of symbols, generate instructions and data, etc.</p> <p>2. Specify data structures. This includes establishing required databases such as Location counter(LC), machine operation table (MOT), pseudo operation table (POT), symbol table(ST), Literal Table(LT), Base Table (BT), etc.</p> <p>3. Define format of data structures. This includes specifying the format and content of each of the databases – a task that must be undertaken before describing the specific algorithm underlying the assembler design.</p> <p>4. Specify algorithm. Specify algorithms to define symbols and generate code.</p>	<p>(List:2 marks, Explanation: 6marks for all steps)</p>



		<p>5. Look for modularity. This includes review design, looking for functions that can be isolated. Such functions fall into two categories: 1) multi-use 2) unique.</p> <p>6. Repeat 1 to 5 on modules.</p>	
6.		Attempt any four :	Marks 16
	1)	Explain implementation of Macro call within Macro.	4M
	Ans:	<p>Macro calls are “abbreviations” of instruction sequences, it seems reasonable that such “abbreviations” should be available within other macro definitions. For example,</p> <pre>MACRO ADD1 &ARG L 1, &ARG A 1, =F"1" ST 1, &ARG MEND MACRO ADDS &ARG1, &ARG2, &ARG3 ADD1 &ARG1 ADD1 &ARG2 ADD1 &ARG3 MEND</pre> <p>Macro Within the definition of the macro “ADDS” are three separate calls to a previously defined macro “ADD1”. The use of the macro “ADD1” has shortened the length of the definition of „ADDS” and thus had made it more easily understood. Such use of macros result in macro expansions on multiple “levels”.</p>	<p>(Explanati on: 3 marks , Example: 1 mark)</p>



2) Sort the following element in descending order using Bucker sort.
45, 21, 12, 36, 97

4M

Ans:

Original table	First Distribution	Merge	Second Distribution	Final Merge
	9)		9) 97	
45	8)	97	8)	97
	7) 97		7)	
21	6) 36	36	6)	45
	5) 45		5)	
12	4)	45	4) 45	36
	3)		3) 36	
36	2) 12	12	2) 21	21
	1) 31		1) 12	

97 0)

21 0)

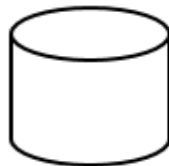
12

Separate Based on last Digit

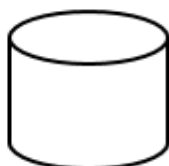
Separate Based on First Digit

OR

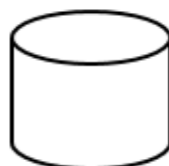
Step1: Initialize all buckets



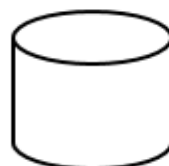
99 - 90



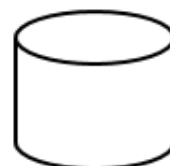
89 - 80



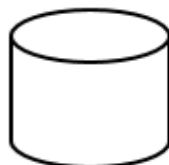
79 - 70



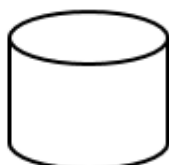
69 - 60



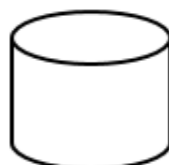
59 - 50



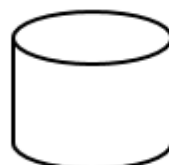
49 - 40



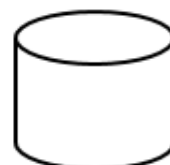
39 - 30



29 - 20

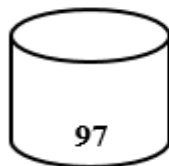


19 - 10



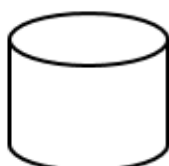
9 - 0

Step2: Putting all elements in appropriate bucket

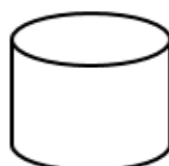


97

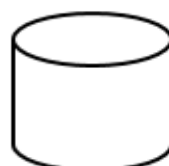
99 - 90



89 - 80



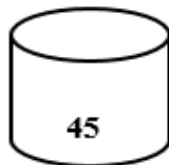
79 - 70



69 - 60

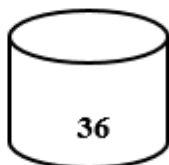


59 - 50



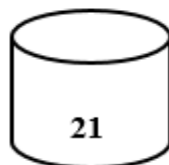
45

49 - 40



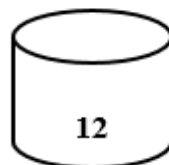
36

39 - 30



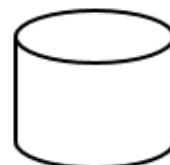
21

29 - 20

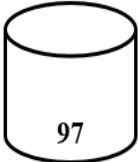
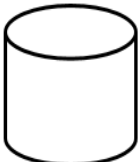
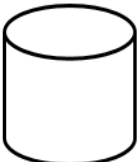
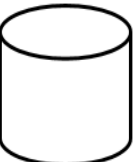

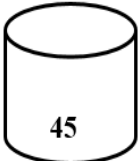
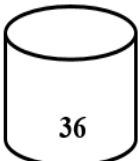
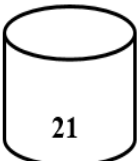
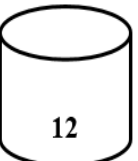



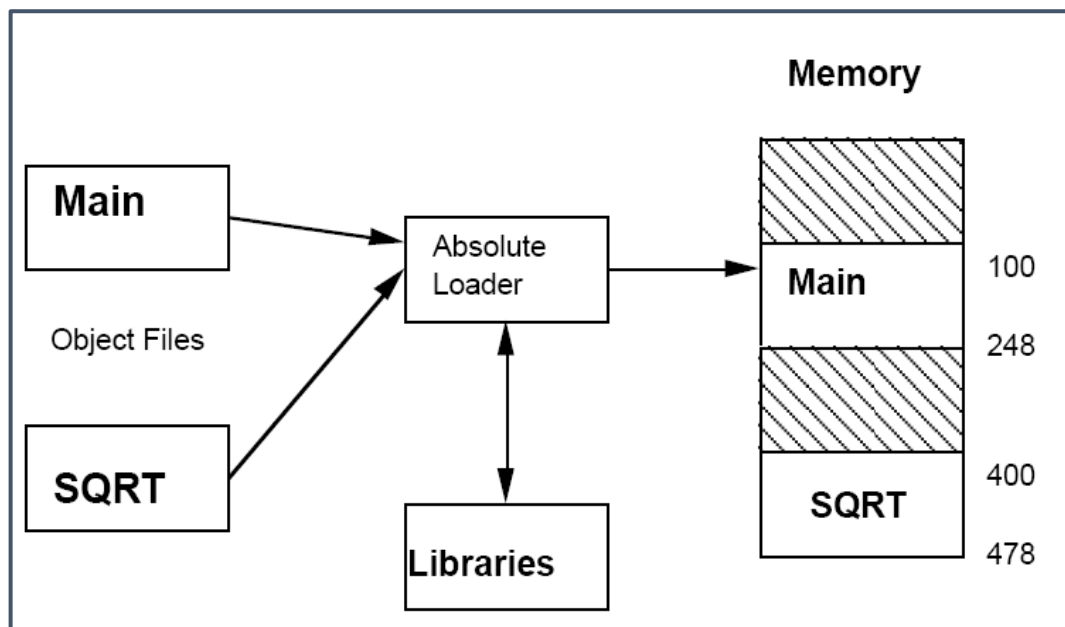
12

19 - 10



9 - 0

		<div style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 80%;"> <p>Step3: Sorting individual bucket</p> <div style="display: flex; justify-content: space-around; align-items: flex-end; margin-bottom: 20px;"> <div style="text-align: center;">  97 99 - 90 </div> <div style="text-align: center;">  89 - 80 </div> <div style="text-align: center;">  79 - 70 </div> <div style="text-align: center;">  69 - 60 </div> <div style="text-align: center;">  59 - 50 </div> </div> <div style="display: flex; justify-content: space-around; align-items: flex-end;"> <div style="text-align: center;">  45 49 - 40 </div> <div style="text-align: center;">  36 39 - 30 </div> <div style="text-align: center;">  21 29 - 20 </div> <div style="text-align: center;">  12 19 - 10 </div> <div style="text-align: center;">  9 - 0 </div> </div> <p>Step 4: After retrieving all elements from individual bucket we get following elements in descending order</p> <p style="text-align: center;">97, 45, 36, 21, 12</p> </div>	
	3)	Explain in detail absolute loader.	4M
	Ans:	<p>An absolute loader is the simplest type of loader scheme that fits the general model of loaders. The assembler produces the output in the same way as in the "compile and go loader". The assembler outputs the machine language translation of the source program.</p> <p>Disadvantage:</p> <ol style="list-style-type: none"> 1. The programmer has to specify the address to the assembler that where the program is to be loaded. 2. It is very difficult to relocate in case of multiple subroutine. 3. Programmer has to remember the address of each subroutine and use that absolute address explicitly in other subroutines to perform subroutine linkage <p>In short:</p> <ol style="list-style-type: none"> 1. Allocation and linking is by programmer 2. Relocation is by assembler 3. Loading is done by loader. 	<p>(Diagram : 1 mark, Explanation: 3 marks)</p>



The figure illustrates the operation of an absolute loader.

4) List and give syntax of database table use in lexical analysis phase of compiler.

4M

Ans:

Lexical Phase:-

- 1) **Source program:** original form of program; appears to the compiler as a string of character
- 2) **Terminal table:** a permanent data base that has an entry for each terminal symbol.
Each entry consists of the terminal symbol, an indication of its classification, and its precedence.

Symbol	Indicator	Precedence
--------	-----------	------------

- 3) **Literal table:** created by lexical analysis to describe all literals used in the source program. There is one entry for each literal, consisting of a value, a number of attributes, an address denoting the location of the literal at execution time, and other information.

Literal	Base	Scale	Precision	Other information	Address
---------	------	-------	-----------	-------------------	---------

**(List:
1mark,
Syntax:
3marks)**



	<div>4) Identifier Table: created by lexical analysis to describe all identifiers used in the source program. There is one entry for each identifier. Lexical analysis creates the entry and places the name of identifier into that entry. The pointer points to the name in the table of names. Later phases will fill in the data attributes and address of each identifier</div> <div><table><tr><td>Name</td><td>Data attributes</td><td>address</td></tr></table></div> <div>5) Uniform Symbol table: created by lexical analysis to represent the program as a string of tokens rather than of individual characters. Each uniform symbol contains the identification of the table of which a token is a member</div> <div><table><tr><td>Table</td><td>Index</td></tr></table></div>	Name	Data attributes	address	Table	Index																								
Name	Data attributes	address																												
Table	Index																													
5)	Explain the format of databases of loader.	4M																												
Ans:	<div>Following are some formats of databases of Loaders...</div> <div><ul style="list-style-type: none">• External Symbol Dictionary (ESD) record: Entries and Externals• (TXT) records control the actual object code translated version of the source program.• The Relocation and Linkage Directory (RLD) records relocation information• The END record specifies the starting address for execution• IPLA-Initial Program Load Address supplied by program’s or OS.• PLA-Program load address counter, to keep track of each segment• GEST-Global External Symbol Table, to store each external symbol & its corresponding core add.• LESA-Local External Symbol Array, which is used establish correspondences between</div> <div>ESD records</div> <div><table><tr><td>symbol</td><td>type</td><td>Rel-location</td><td>Length</td></tr><tr><td>JOHN</td><td>SD</td><td>0</td><td>64</td></tr><tr><td>RESULT</td><td>LD</td><td>52</td><td></td></tr><tr><td>SUM</td><td>ER</td><td>—</td><td>—</td></tr></table></div> <div>RLD record</div> <div><table><tr><td>Symbol</td><td>Flag</td><td>Length</td><td>Relative location</td></tr><tr><td>JOHN</td><td>+</td><td>4</td><td>48</td></tr><tr><td>SUM</td><td>+</td><td>4</td><td>56</td></tr></table></div> <div>TXT record</div>	symbol	type	Rel-location	Length	JOHN	SD	0	64	RESULT	LD	52		SUM	ER	—	—	Symbol	Flag	Length	Relative location	JOHN	+	4	48	SUM	+	4	56	<div>(Listing of databases :2 marks, Any two formats: 2 marks)</div>
symbol	type	Rel-location	Length																											
JOHN	SD	0	64																											
RESULT	LD	52																												
SUM	ER	—	—																											
Symbol	Flag	Length	Relative location																											
JOHN	+	4	48																											
SUM	+	4	56																											