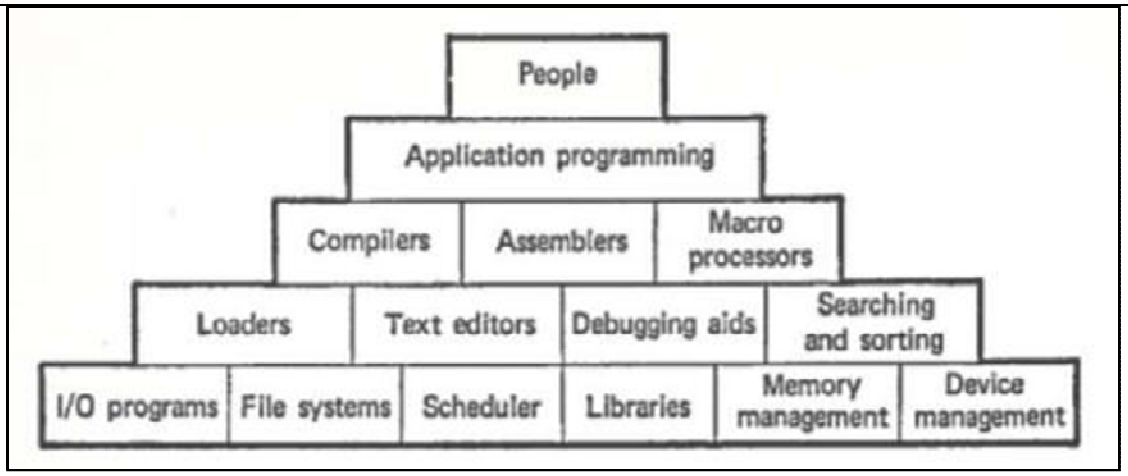
Important Instructions to examiners:

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:**	**(3×4=12)**
	1)	**State and explain the functions of loader.**	**4M**
	Ans:	• **Loader Function:** The loader performs the following functions: • **Allocation**- The loader determines and allocates the required memory space for the program to execute properly. • **Linking-** The loader analyses and resolve the symbolic references made in the object modules. • **Relocation-** The loader maps and relocates the address references to correspond to the newly allocated memory space during execution. • **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.	***(Each function 1 mark, Only List :1 mark)***
	2)	**What are the four components of system software?**	**4M**
	Ans:	**Components of system software are:** 1. Assembler 2. Macros 3. Loader 4. Linker 5. Compiler **1. Assembler:** It is a language translator that takes as input assembly language	***(Any 4 Component: 1 mark Each)***

		program (ALP) and generates its machine language equivalent along with information required by the loader. **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". The template for designing a macro is as follows **3. Loader**: It is responsible for loading program into the memory, prepare them for execution and then execute them. **OR** Loader is a system program which is responsible for preparing the object programs for execution and start the execution. **4. Linker:** A linker which is also called binder or link editor is a program that combines object modules together to form program that can be executed. Modules are parts of a program. **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).	
	3)	**Describe the steps of design for assembler.**	**4M**
	Ans:	**Step 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. **Step 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. **Step 3: Define format of data structures** This includes specifying the format and content of each of the data bases – a task that must be undertaken before describing the specific algorithm underlying the assembler design. **Step 4: Specify algorithm** Specify algorithms to define symbols and generate code **Step 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 **Step 6: Repeat 1 to 5 on modules**	***(All Steps : 4 marks)***
	4)	**What must the compiler do in order to produce the machine language equivalent of WCM?** **(Any relevant answer can also be considered)**	**4M**

	Ans:	1. Recognize certain strings as basic element e.g. recognize COST is a variable, WCM label, PROCEDURE is a keyword, and "=" is an operator. 2. Recognize combinations of elements as Syntactic units and interpret their meaning, e.g. ascertain that the first statement is a procedure name with three arguments, that the next statement defines four variables to be fixed binary numbers of 31 bits, that the third statement is an assignment statement that requires seven computations, and that the last statement is a return statement with one argument. 3. Allocate storage and assign location for all variables in this program 4. Generate the appropriate object code.	***(Each step :1 mark)***
	b)	**Attempt any one:**	**(1×6=6)**
	1)	**Explain the foundation of system programming.**	**6M**
	Ans:	 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. 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.	***(Diagram: 2 marks, Description: 4 marks)***
	2)	**Explain macro instructions with the help of its structure and example**	**6M**
	Ans:	**Description: -** Macro is used to give single line abbreviation to group of lines which are repeatedly used in program. These statements are combined and kept in macro. Whenever such single line abbreviation is encountered macro processor expands replaces this abbreviation with associated group of lines. Macro Processor is a program that lets you	***(Description :2 marks, Structure :2 marks,Example: 2 marks)***

define the code that is reused many times giving it a specific Macro name and reuse the code by just writing the Macro name only.

Structure of Macro:

```
MACRO MACRO_NAME
  …
  MACRO BODY
  …
MEND
```

Example:-

Source

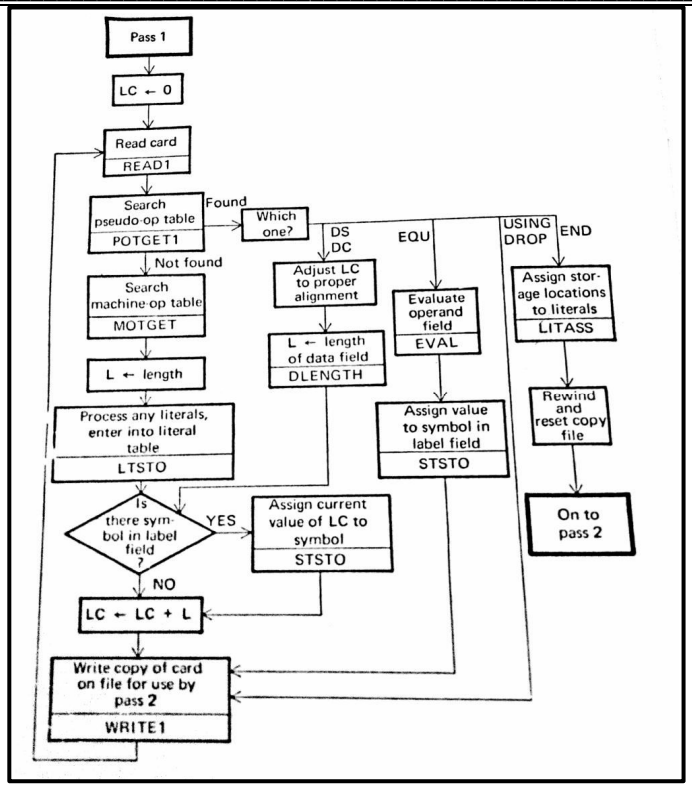
```
MACRO  SPR
       STA     DATA1
       STB     DATA2
       STX     DATA3
       MEND
   .
SPR
   .
SPR
   .
   .
```

Expanded source

```
   .
   .
   .
       STA     DATA1
       STB     DATA2
       STX     DATA3
   .
       STA     DATA1
       STB     DATA2
       STX     DATA3
   .
```

2.		**Attempt any two:**	**(2×8=16)**
	1)	**Draw and explain the flowchart for pass-I of assembler**	**8M**

Ans:



(***Flowchart: 4 marks, Description: 4 marks***)

PASS1:

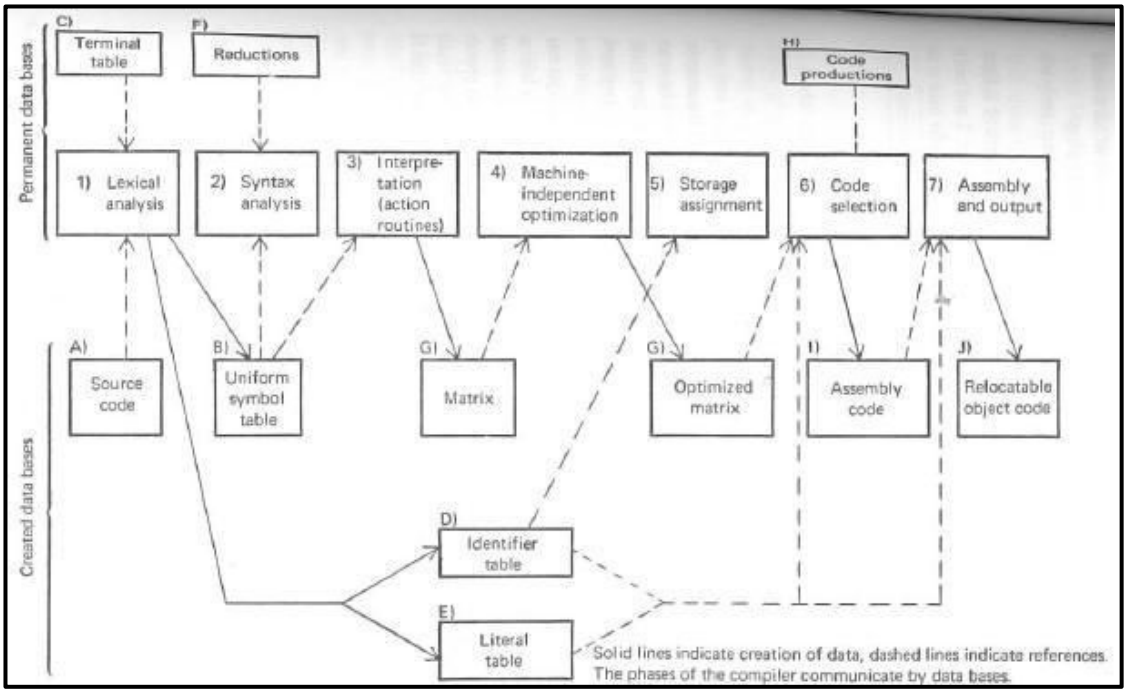
DEFINE SYMBOLS the purpose of the first pass is to assign a location to each instruction and data defining pseudo- instruction, and thus to define values for symbols appearing in the label; fields of the source program. Initially, the Location Counter (LC) is set to the first location in the program (relative address 0) then a source statement is read the operation code field is examine to determine if it is pseudo-op; if it is not, table of machine op-code (MOT) is search to find match of source stamen. Op-code field the match MOT entry specifies the length (2, 4 or 6 Bytes) of the instructions the operand field is scanned for the presence of literal. If a new literal is found, it Is entered into the literal table (LT) for later processing. The label field of source statement is then examine for the presence of the symbol if there is label, symbol is saved in the symbol table (ST) along with the current value of the location counter. Finally, the current value of the Location counter is increment by Length of the instruction. And the copy of as our card is saved for used by Pass 2. The above sequence is then repeated for the next instruction.

The simplest procedure occurs for USING and DROPP as s1 is only concern with pseudo-ops that defines symbols (Labels) or affects the location counter; USING and DROP do neither assembler need only save the USING and DROP card for Pass 2.

In case of EQU pseudo-op during Pass1 We can concern only with defining the symbol in the label field this require evaluating the expression in the operand field (The symbol in the operand field and EQU statement must have been defined previously).

The DS and DC pseudo-op scan affects both the location counter and definition of symbols in Pass1. The operand field must be examine to determine the number of by

		test of storage require due to requirement for certain alignment conditions. It may be necessary to adjust the location counter before defining the symbol. When the END Pseudo–op is encountered Pass 1 is terminated before transferring to control to Pass2. There are various "housekeeping" operation that must be performed this including assigning location the literal that have been collected during Pass1, a procedure that is very similar that for the DC pseudo-op, finally conditions are reinitialized for processing by Pass 2.	
	2)	**What is the need of searching and sorting techniques in system programming? Elaborate your answer in details.** **(Any Relevant Description can also be considered)**	**8M**
	Ans:	This attention to searching and sorting techniques derives from an awareness of critical performance areas or potential bottlenecks. All of the identified assembler functional modules are of comparable programming complexity; whereas the listing format module is used once for every source card, the symbol table search module, for example, contains a loop that may be executed hundreds or thousands of times for every source card. Since the software designer only has a finite amount of time, it is essential to accentuate those particular modules that yield the highest performance results. A simple example should illustrate this point. Consider an assembler running on a medium-speed computer, such as an IBM System/360 Model 40, which has a typical instruction time of 12 microseconds (i.e., it averages 83,000 instructions per second). Assume that we wish to assemble a fairly large assembly source program of 5,000 cards, which contains about 2,000 symbols. On an average, each source card has at least one symbolic reference in the operand field. Let us compute the amount of time spent in searching the symbol table. If a linear search is used and programmed as in Figure 3.12, there will be five instructions executed for each loop of the search. Thus each iteration will take about 5x12.60 microseconds. The number of iterations for each search will be approximately half the symbol table size, 1,000P (I/2) of 2,000. Finally, there will be approximately one search for each of the 5,000 source cards. Thus we can estimate the total time spent searching as: Total search time = number of searches x number of iterations per search x time per iteration = (5x103) x (103) x (60x106) = 300 seconds = 5 minutes On the other hand, if a binary search is used, the average number of iterations per search becomes only log2 (2000) -1 10, and the time per iteration is about 100 microseconds if programmed Total search time = (5x103) x (10) x (100x I 0-6) 5000 x iO3 rr 5 seconds	***(Need : 4 marks, Elaboration: 4 marks)***

	3)	**Draw the structure of compiler and explain it.**	**8M**
	Ans:	In analyzing the compilation of simple program there are seven distinct logical problems as follows and summarized in figure below.  **1. Lexical analysis–** Recognition of basic elements of creation of uniform symbols. **2. Syntax analysis–**Recognition of basic syntactic constructs through reductions. **3. Interpretation–** Definition of exact meaning, creation of matrix and tables by action routines. **4. Machine Independent Optimization–** Creation of more optimal matrix. **5. Storage Assignment–**Modification of identifier and literal tables. If makes entries in the matrix that allow code generation to create code that allocates dynamic storage and that also allow the assembly phases to reserve the proper amounts of STATIC storage. **6. Code Generation–** Use of macro processor to produce more optimal assembly code. **7. Assembly And Output–** Resolving symbolic addresses and generating machine language.	***(Diagram: 4 marks, Description: 4 marks.)***
3.		**Attempt any four:**	**(4×4=16)**
	1)	**Elaborate the evolution of operating system.**	**4M**
	Ans:	**Batch Systems** Several jobs are kept in main memory at the same time, and the CPU is multiplexed among them. **Multiprogramming I/O** routine supplied by the system. Memory management – the system must allocate the memory to several jobs. CPU scheduling – the system must choose amongst several jobs ready to run. Allocation of devices. Multitasking is a logical extension of multiprogramming. Multiple jobs are executed	***(Elaboration on Evolution of any 4 Operating System : 1 mark each)***

		by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running. **Time-Sharing Systems–Interactive Computing** The CPU is multiplexed among several jobs that are kept in memory and on disk (the CPU is allocated to a job only if the job is in memory). A job swapped in and out of memory to the disk. On-line communication between the user and the system is provided; when the operating system finishes the execution of one command, it seeks the next "control statement" from the user's keyboard. On-line system must be available for users to access data and code. **Desktop Systems or Personal computers** – computer system dedicated to a single user. I/O devices – keyboards, mice, display screens, small printers. User convenience and responsiveness. Can adopt technology developed for larger operating system" often individuals have sole use of computer and do not need advanced CPU utilization of protection features. May run several different types of operating systems (Windows, MacOS, UNIX, Linux). **Distributed system or distributed data processing** is the system in which processors, data and other aspects of a data processing system may be dispersed within on the organization. A DDP system involves a partitioning of the computing function and may also involve a distributed of databases, device control and interaction (network) control. **A Real Time system** is used when there are rigid time required for the operation of a processor or the flow of data and thus is often used as a control device in a dedicated application. A Real Time system is considered to function correctly only if it returns the correct result within any time constraint. Hard real-time system Soft real-time system	
	2)	**Apply linear search on following numbers and search the number 15 from it.** **1,3,7,9,11,13,15,19,21**	**4M**
	Ans:	**Algorithm** 1. Start with the first Number in the list. 2. Compare the current Number to the target 3. If the current Number matches the target then we declare search found and stop. 4. If the current number is not equal to the target then set the current number to be the next Number and repeat from 2. Search for number 15 in the list i.e. Target = 15	***(Correct steps : 4 marks)***

1	3	7	9	11	13	15	19	21

1	3	7	9	11	13	15	19	21

No match

Next Number

1	3	7	9	11	13	15	19	21

No match

Next Number

1	3	7	9	11	13	15	19	21

No match
Next Number

1	3	7	9	11	13	15	19	21

No match
Next Number

1	3	7	9	11	13	15	19	21

No match
Next Number

1	3	7	9	11	13	15	19	21

No match
Next Number

1	3	7	9	11	13	15	19	21

Match.
Target = Number. **(Target = 15)**
Search Found
STOP.

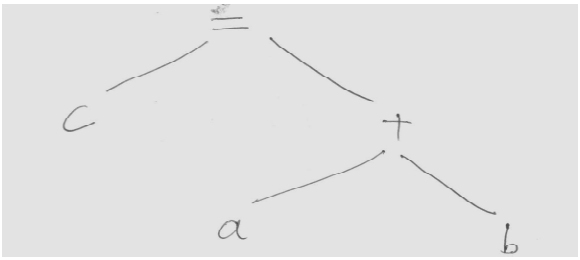
3)	**Explain syntax analysis with the help of example.**	**4M**

Ans:

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

(Explanation of Syntax analysis : 3 marks, Example : 1 mark)

		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. e.g. The syntax phase takes as input tokens generated by lexical phase and if meaning is correct it generates a parse tree. The output of syntax analysis phase for the string 'c=a+b' in the form of syntax tree is as follows 	
	4)	**Explain compile and go loader.**	**4M**
	Ans:	**"Compile and go" loader:** The assembler runs in one part of the memory and place the assembled machine instructions and data as they are assembled directly into their assigned memory locations. When assembly is completed, the assembler causes a transfer to the starting instruction of the program. **Advantages:-** 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. **Disadvantages:-** 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)	***(Description: 2 marks, Diagram :2 marks)***

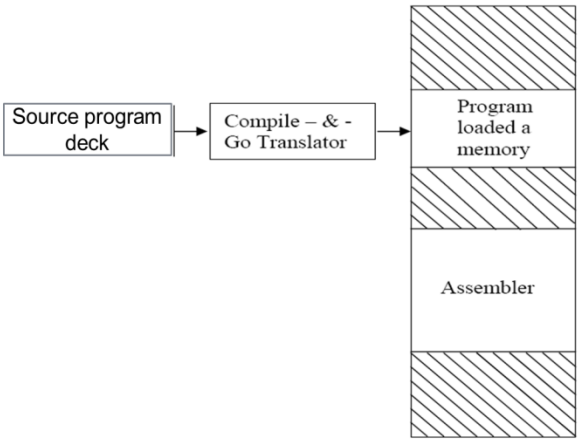


Fig.2.7 Compile - & - Go Loader

	5)	**Explain the meaning of top down and bottom up parser.**	**4M**
	Ans:	**Top-down Parser** The top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. It can be done using recursive decent or LL(1) parsing method. It cannot handle left recursion. It is only applicable to small class of grammar. **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. It starts from a sentence and then apply production rules in reverse manner in order to reach the start symbol. It is a table driven method and can be done using shift reduce, SLR, LR or LALR parsing method. It handled the left recursive grammar. It is applicable to large class of grammar. Consider the grammar S →cAd A → abla and the input string w = cad	***(Top down parser : 2 marks and bottom up parser: 2 marks)***

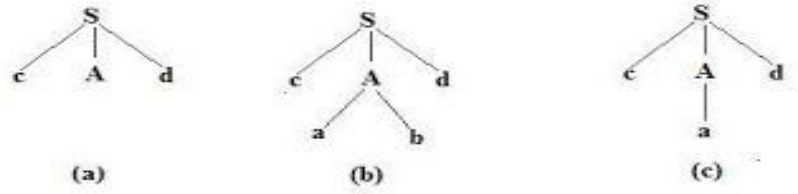


Fig: Top – Down Parsing

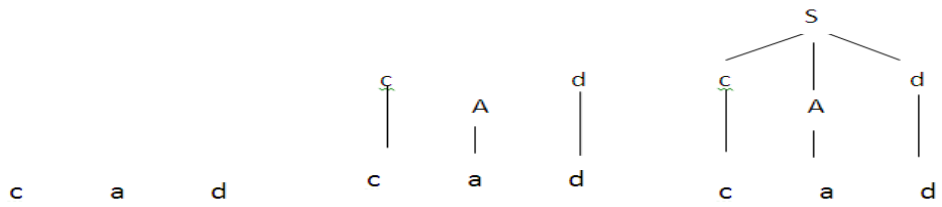


Fig:-Steps in bottom up parse

4.	**a)**	**Attempt any three:**	**(3×4=12)**
	1)	**Explain four cards in the objects desk of assembler i.e. ESD, TXT, RLD and END.**	**4M**
	Ans:	There are four sections of the object deck for a direct linking loader. The ESD card the information necessary to build the external symbol. The external symbols are symbols that can be referred beyond the subroutine level. The normal labels in the source program are used only by the assembler. The ESD card contains the information necessary to build the external symbol. The external symbols are symbols that can be referred beyond the subroutine level. The normal labels in the source program are used only by the assembler. **ESD card format:** **The TXT card** contains the blocks of data and the relative address at which data is to be placed. Once the loader has decided where to load the program, it adds the Program Load Address (PLA) to relative address. The data on the TXT card may be instruction, non-related data or initial values of address constants. **TXT card format**	***(Explanation of each card, diagram optional:1 mark each)***

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters ESD
5-14	Blank
15-16	ESD identifier (ID) for program name (SD) external symbol(ER) or blank for entry (LD)
17-24	Name, padded with blanks
25	ESD type code (TYPE)
26-28	Relative address or blank
29	Blank
30-32	Length of program otherwise blank
33-72	Blank
73-80	Card sequence number

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters TXT
5	Blank
6-8	Relative address of first data byte
9-10	Blanks
11-12	Byte Count (BC) = number of bytes of information in cc. 17-72
13-16	Blank
17-72	From 1 to 56 data bytes
73-80	Card sequence number

The RLD cards contain the following information 1. The location and length of each address constant that needs to be changed for relocation or linking. 2. The external symbol by which the address constant should be modified. 3. The operation to be performed. RLD card format

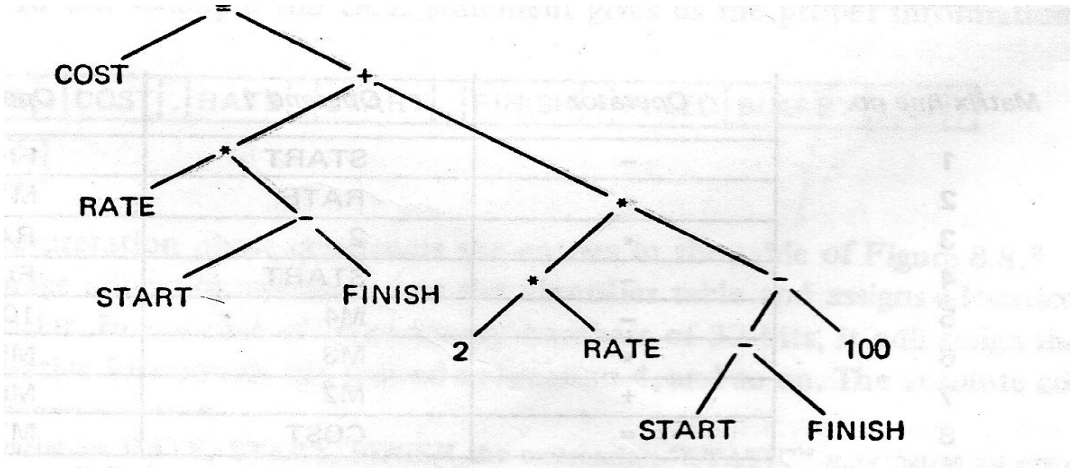
Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters RLD
5-18	Blank
19-20	Relative address of first data byte
21	Blanks
22-24	Byte Count (BC) = number of bytes of information in cc. 17-72
25-72	Blank
73-80	Card sequence number

The END card specifies the end of the object deck.
END card format

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters END
5	Blank
6-8	Start of execution entry (ADDR), if other than beginning of program
9-72	Blanks
73-80	Card sequence number

2) Generate the parse tree for following expression. **4M**
Cost = rate*(start - finish) + 2 * Rate * (start-finish)-100

Ans:



(Correct Pass Tree : 4 marks)

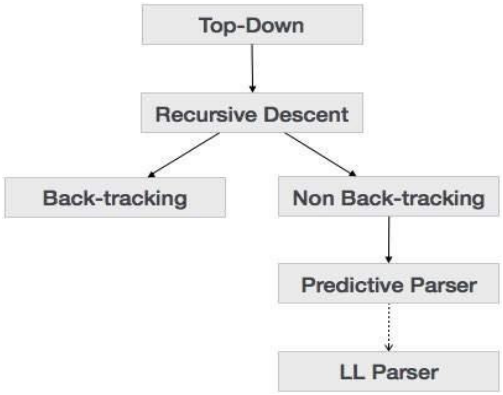
3) Write the matrix for the following expression. **4M**
Cost = rate*(start - finish) + 2 * Rate * (start-finish)-100

Ans:

Matrix line no.	Operator	Operand 1	Operand 2
1	-	START	FINISH
2	*	RATE	M1
3	*	2	RATE
4	-	START	FINISH
5	-	M4	100
6	*	M3	M5
7	+	M2	M6
8	=	COST	M7

(Correct Matrix : 4 marks)

4) Explain the concept of top down parser. **4M**

	Ans:	**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. Recursive descent parsing: It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking. Backtracking 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. The types of top-down parsing are depicted below:  **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. **Back-tracking** 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→ rXd | rZd X → oa | ea Z → a | i	***(Description : 2 marks; 2 marks for Description of any two type of top down parser)***
	b)	**Attempt any one:**	**(1×6=6)**
	1)	**Explain conditional macro expansion with the help of example.**	***6M***
	Ans:	Two important macro-processor pseudo-ops AIF and AGO permit conditional reordering of the sequence of macro expansion. This allows conditional selection of	***(explanation:4 marks , example***

the machine instructions that appear in expansions of Macro call. Consider the following program. . *: 2 marks)*

```
        Loop 1    A1, DATA 1
                  A2, DATA 2
                  A3, DATA 3
                  .
                  .
        Loop 2    A1, DATA 3
                  A2, DATA 2
                  .
                  .
        Loop 3    A1, DATA1
                  .
        .
         DATA 1 DC F‘5’
         DATA 2 DC F’10’
         DATA 3 DC F’15’
```

In the below example, the operands, labels and the number of instructions generated change in each sequence. The program can written as follows:-

```
                  .
                  .
                  .
          MACRO
&ARG0     VARY    &COUNT,&ARG1,&ARG2,&ARG3
&ARG0     A       1,&ARG1
          AIF     (&COUNT EQ 1).FINI
          A       2,&ARG2
          AIF     (&COUNT EQ 2).FINI
          A       3,&ARG3
.FINI     MEND                                   EXPANDED SOURCE
                  .                                      .
                  .                                      .
                  .                                      .
LOOP1     VARY    3,DATA1,DATA2,DATA3            LOOP1   A   1,DATA1
                  .                                      A   2,DATA2
                  .                                      A   3,DATA3
                  .                                      .
LOOP2     VARY    2,DATA3,DATA2                  LOOP2   A   1,DATA3
                  .                                      A   2,DATA2
                  .                                      .
                  .                                      .
LOOP3     VARY    1,DATA1                        LOOP3   A   1,DATA1
                  .
                  .
                  .
DATA1     DC      F’5’
DATA2     DC      F’10’
DATA3     DC      F’15’
```

Labels starting with a period (.) such as .FINI are macro labels and do not appear in the output of the macro processor. The statement AIF (& COUNT EQ1) .FINI direct the macro processor to skip to the statement. Labeled .FINI if the parameter corresponding to & COUNT is a1; otherwise the macro processor is to continue with the statement following the AIF pseudo-ops. AIF is conditional branch pseudo ops it performs an arithmetic test and branches only if the tested condition is true. AGO is an unconditional branch pseudo-ops or 'Go to' statement. It specifies label appearing on some other statement. AIF & AGO controls the sequence in which the macro processor expands the statements in macro instructions.

2) Compare advantages and disadvantages of top down and bottom up parser. ***6M***

Ans:

(Any 6 Points of comparison: 1 mark each)

Top down parser

Top Down Parser	Bottom-up Parser
Easy to implement.	Difficult to implement.
It never waste time on sub trees which does not have 'S' symbol at root.	It waste time on sub trees for which start state is not defined.
It can back track while creating parse tree.	Bottom up parser cannot back track.
It cannot handle left recursion.	It can handle left recursion
Not applicable for complex grammar.	It can handle complex grammar.
It is not efficient parsing method	It is highly efficient parsing method.

OR

Advantages:-

1. It is easy to implement
2. It never wastes time on sub trees that cannot have an S at the root. Bottom up parsing does this.

Disadvantages:-

1. It is not efficient parsing method as compare to bottom up parser
2. It cannot handle left recursion.
3. It is not applicable to large scale of grammar.
4. Wastes time on trees that don't match the input (compare the first word of the input with the leftmost branch of the tree). Bottom-up parsing doesn't do this.

Bottom up parser

Advantages:-

1. It is efficient parsing method.
2. Left recursion framer is handled by bottom up parser.
3. It is applicable to large scale of grammar.

Disadvantages:-

1. It wastes time on sub trees that cannot have an S at the root.
2. Bottom-up parse postpones decisions about which production rule to apply until it has more data than was available to top-down.

5.		Attempt any two:	(2×8=16)
	1)	**What are the specifications of data structures and formats of data bases used in direct linking loader?**	**8M**
	Ans:		***(Any 4 Data Structure : 2 marks Each)***

1. The ESD cards contain the information necessary to build the external symbol. The external symbols are symbols that can be referred beyond the subroutine level. The normal label in the source program is used only by the assembler.

ESD card format:

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters ESD
5-14	Blank
15-16	ESD identifier (ID) for program name (SD) external symbol(ER) or blank for entry (LD)
17-24	Name, padded with blanks
25	ESD type code (TYPE)
26-28	Relative address or blank
29	Blank
30-32	Length of program otherwise blank
33-72	Blank
73-80	Card sequence number

2. The TXT card contains the blocks of data and the relative address at which data is to be placed. Once the loader has decided where to load the program, it adds the Program Load Address (PLA) to relative address. The data on the TXT card may be instruction, non- related data or initial values of address constants.

TXT card format:

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters TXT
5	Blank
6-8	Relative address of first data byte
9-10	Blanks
11-12	Byte Count (BC) = number of bytes of information in cc. 17-72
13-16	Blank
17-72	From 1 to 56 data bytes
73-80	Card sequence number

1. **The RLD cards** contain the following information
2. The location and length of each address constant that needs to be changed for relocation or linking.
3. The external symbol by which the address constant should be modified.
4. The operation to be performed.

RLD card format:

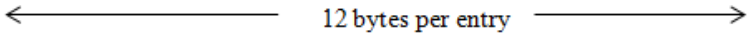
Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters RLD
5-18	Blank
19-20	Relative address of first data byte
21	Blanks
22-24	Byte Count (BC) = number of bytes of information in cc. 17-72
25-72	Blank
73-80	Card sequence number

1. **The END card** specifies the end of the object deck.

END card format:

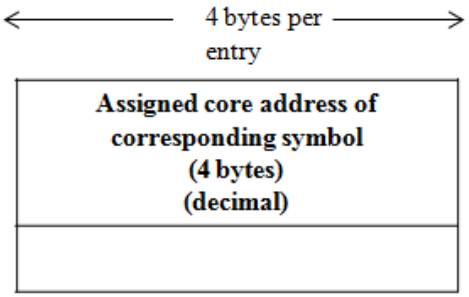
Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters END
5	Blank
6-8	Start of execution entry (ADDR), if other than beginning of program
9-72	Blanks
73-80	Card sequence number

2. **GEST** specifies the Global External Symbol Table format.



External symbol (8 bytes) (characters)	**Assigned core address (4 bytes) (decimal)**

3. **The LESA** specifies the local External Symbol Array.



2) Explain code generation phase of compiler with respect to database and algorithms. **8M**

Ans: *(4 marks for databases, 4 marks for algorithm)*

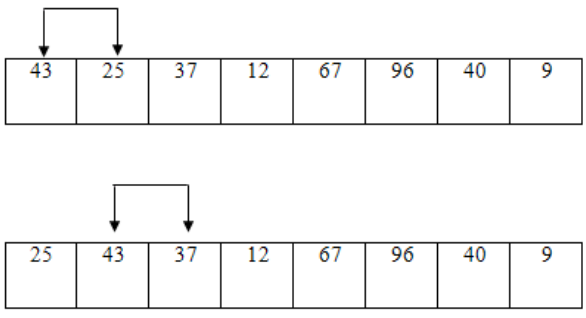
CODE GENERATION:

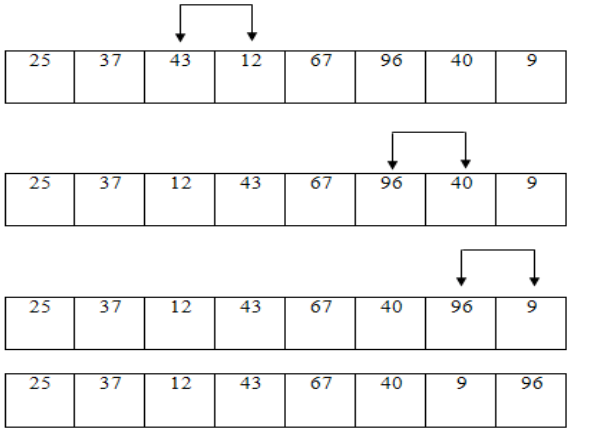
The purpose of the code generation phase is to produce the appropriate code (assembly or machine language). The code generation phase has the matrix as input. It uses the code productions (macro definitions) which define the operators that may appear in the matrix to produce code. It also references the identifier tables and literal tables in order to generate proper address and code conversions.

Databases used:
Matrix: each entry has its operator defined in the code pattern database.
Identifier table and literal table: they are used to determine the data type and locations of the variables so that proper accessing code with the correct address is generated.
Code productions (macro definition): a permanent data base defining all possible matrix operators.

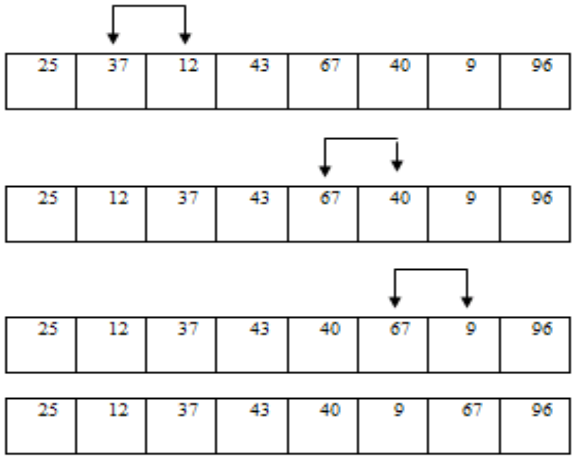
Standard code definition for -,+,*,=

```
-   L     1, & OPERAND 1
    S     1, & OPERAND 2
          1, M&N
    ST
*   L     1, & OPERAND 1
    M     0, & OPERAND 2
          1, M&N
    ST
+   L     1, & OPERAND 1
    A     1, & OPERAND 2
          1, M&N
    ST
=   L     1, & OPERAND 2
          1, & OPERAND 1
    ST
```

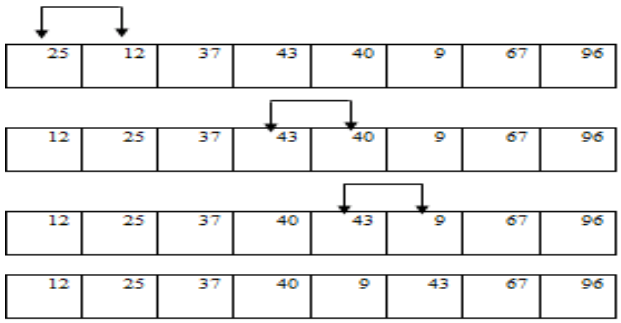
		By using above standard definition for +,-,*, =, the code will be generated for respective arithmetic statement. **Code generation algorithm:** 1. The code generation process is implemented in a way that is used by the assembler macro processor. 2. Prepare Matrix for each entry of the code. 3. The operation field of each matrix line is treated as "macro-call" and the matrix operands (M_i) on a line are used as "macro-argument". 4. Next step is to optimize a code using either "Machine-dependent" or "machine-independent" optimization techniques. 5. The code generation optimization can be done using three possible ways: i) during matrix optimization, ii) during code generation, or iii) during post-pass after assembly code is generated. 6. Code generation during matrix optimization: a. Load operands M_i to registers according to the matrix entry. b. If an operand, M_i, is (at execution time) already in the register, it is not reloaded. Else, "next matrix line code generation" stores the temporary matrix as it is the only line that knows whether the previous temporary matrix will be needed immediately	
	3)	**Apply interchange sort on following numbers:** **43, 25, 37, 12, 67, 96, 40, 9**	**8M**
	Ans:	**1st Iteration :** 	***(8 marks for correct answer)***



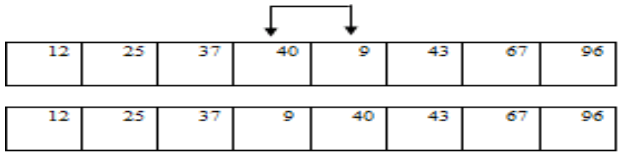
2nd Iteration :

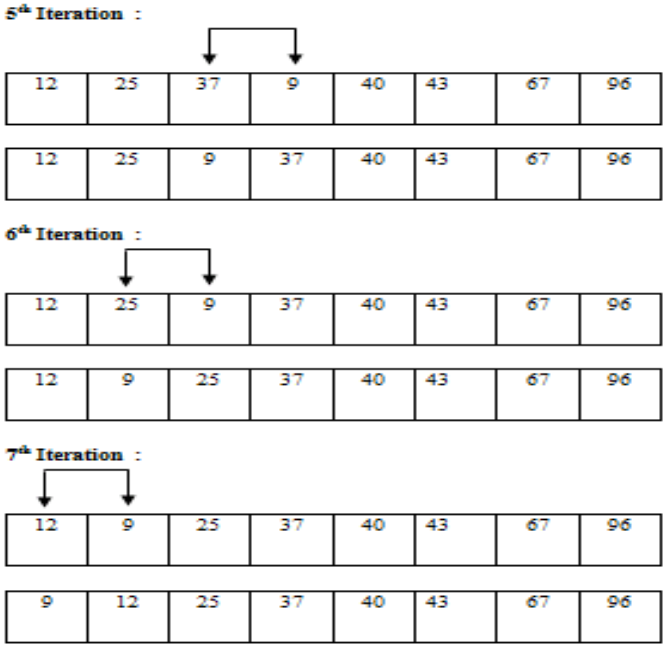


3rd Iteration



4th Iteration :





Sorted List : 9, 12, 25, 37, 40, 43, 67, 96

6.		**Attempt any four:**	**(4×4=16)**
	1)	**Explain a single pass algorithm for macro processor.**	**4M**
	Ans:	If we wanted to provide for macro definitions within macros. The basic problem here is that inner macro is defined only after the outer one has been called in order to provide for any use of the inner macro we would have to repeat both the macro-definition and the macro-call passes. However there is a simpler solution that has added advantages of reducing all macro processing to a single pass. There are two additional variables introduced in the one –pass design a macro definition input (MDI) indicator and a Macro Definition Level Counter (MDLC). The MDI and MDLC are switches (counters) used to keep track of macro calls and macro definitions. The MDI indicator has the value "ON" during expansion of a macro call and the value "OFF" at all other times. The actual expansion of macro calls is performed in the read box. READ tests the switch MDI. If it is "ON", lines are read from the Macro Definition Table (MDT). The reading of a MEND line indicates the end of a macro and terminates expansion of call: MDI is reset to "OFF" and the next line is obtained from the regular input stream. Note that lines returned by READ may include macro definition's; expanded macro code comes out of READ looking just like any other code and may therefore include macro definitions. The macro definition level counter is incremented by 1 when a MACRO pseudo-op is encountered and decremented by 1 when a MACRO pseudo-op occurs. The MDLC is used to insure that the entire macro definition, including MACROs and MENDs, gets stored in MDT.	***(Description of Working of Single pass Macro Processor : 4 marks)***

Working of Macro

1. Start
2. Initialize MDTC = 1
   MNTC =1
   MDI = OFF
   MDLC = 0
3. Read- perform read operation of macro.
4. Search MNT for match found of operation code.
5. Is macro name found ? macro call
   If yes MDI ='ON'
   MDTP = MDT Index from MNT entry
   Setup ALA
   Goto step 3
   If no go to step 6
6. Is macro pseudo-op ?macro definition,
   If yes MDLC = MDLC + 1
   If no then read code again.
7. Store macro name and current value of MDTC in MNT entry number in MNTC.
8. Increment MNTC by 1
   Prepare ALA
9. Store macro name card in MDT
   Increment MDTC by 1
   Increment MDLC by 1
10. Read - perform read operation of macro.
11. Substitute index notation for arguments in definition.
    Enter line is MDT
    Increment MDTC by 1
12. Is MACRO pseudo-op
    If yes increment MDLC by 1 and goto step 10.
    If no
    Is MEND pseudo-op ?
    If yes
    Decrement MDLC by 1
    Goto step 13.
    If no, goto step 10
13. If MDLC = 0
    If yes, goto step 3
    If no, goto step 10
14. Write expanded code in source file card.
15. Is END pseudo-op ?
    If yes
    Supply expanded source file to assembler processing

		If no, goto step 3.	
	2)	**Illustrate the algorithm for hash search.**	**4M**
	Ans:	**1) Hashing:** Hashing is the transformation of a string of characters into usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find shorter hashed key than to find it using the original value. **2)** Binary search algorithms are operated on tabled that are ordered and packed. Therefore it has to be used in conjunction with sort algorithms which both ordered and pack the data. So a considerable improvement can be achieved by inserting elements in a random way. The random entry number K is generated from the key. If the K^{th} position is valid, then the new element is put there; if not then some other cell must be found for the insertion. **3)** Here the first problem is to generate a random number from the key. This can be achieved by dividing a four character keyword by the table length N and use the remainder. Another method is to treat a keyword as a binary fraction and multiply it by another binary fraction: L 1, SYMBOL M 0, RHO **4)** The result is 64 bit product in registers 0 and 1. If RHO is chosen carefully, the low order 31 bits will be evenly distributed between 0 and 1, and the second multiplication by N will generate number uniformly distributed over 0…(N-1). This is known as power residue method. The second problem is the procedure to be followed when the first trial entry results in a filled position. This problem can be resolve by using one of the following methods: **1) Random entry with replacement:** A sequence of random numbers is generated from the keyword. From each of these a number between 1 and N is formed and the table is probed at that position. Probing are terminated when a void space is found. **2) Random entry without replacement:** this is the same as above expect that any attempt to probe the same position twice is bypassed. **3) Open addressing:** if the first probe gives a position K and that position is filled, then the next location K+1 is probed and so on until a free position is found. If the search runs off the bottom of the table, then it is renewed at the top. **Example:** Consider a table of 17 positions (N=17) in which the following 12 numbers are to be stored. 19, 13, 05, 27, 01, 26, 31, 16, 02, 09, 11, 21 These items are to be entered in the table at the position defined by the remainder after division by 17; if that position is filled, then the next position is examined, etc. The following table shows progress entry for the 12 items. The column ‘probes to find’ gives the number of probes necessary to find the corresponding item in the tables; thus it takes 3 probes to find item 09, 2 probes to find item 11 and 1 to find item 26. The	***(3 marks explanation: 1 mark example)***

column 'probes to find' gives the number of probes necessary to determine that the item is not in the table; thus the search for the number 54 would give an initial position of 3 and it would take 4 probes to find that the item is not present.

Position	Item	probes to find	probes to find not
0			1
1	01	1	6
2	19,02*	1	5
3	02	2	4
4	21	1	3
5	05	1	2
6			1
7			1
8			1
9	26, 09*	1	7
10	27, 09*	1	6
11	09, 11*	3	5
12	11	2	4
13	13	1	3
14	31	1	2
15			1
16	16	1	1
		16	54

Length of the table	N = 17
Items stored	M=12
Density	p = 12/17 = 0.705
Probes to store	$T_s = 16$
Average probes to find	$T_p = 16/12 = 1.33$
Average probes to find	$T_n = 54/16 = 3.37$

3) What are the uses of binders, linking loader overlays and dynamic binders? **4M**

Ans: **Uses of**

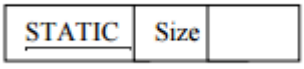
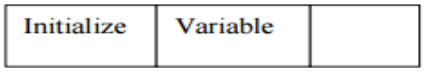
1. Binders

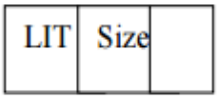
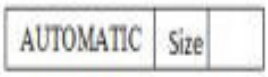
1. It is a program which performs the same function as the direct-linking loader in "binding" subroutines together.
2. Instead of placing the relocated and linked text directly into memory, it outputs the text as a file or card deck. This file is called as load module because it is in a form of ready to load.
3. It performs the function of allocation, relocation, linking.

2. Linking loader 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

(1 mark binders: 2 marks linking loader: 1 mark dynamic binders for each)

		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 two 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. **3. Dynamic binders** The dynamic binder loads only those subroutines in the program which are needed for execution thereby not loading all the subroutines of a certain program. This results in reduced usage of memory.	
	4)	**Explain storage allocation concept in compiler.**	**4M**
	Ans:	The storage allocation phase first scans through the identifier table, assigning locations to each entry with a storage class of static. It uses a location counter, initialized at zero, to keep track of how much storage it has assigned. **Whenever it finds a static variable in the scan, the storage allocation phase does the following steps:** 1. Updates the location counter with any necessary boundary alignment. 2. Assigns 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 (by examining its attributes). 4. Updates the location counter by adding this length to it. 5. Once it has assigned relative address to all identifiers requiring STATIC storage locations, this phase creates a matrix entry:  6. This allows code generation to generate the proper amount of storage. For each variable that requires initialization, the storage allocation phase generates a matrix entry:  7. This tells code generation to put into the proper storage location the initial values that the action routines saved in the identifier table. 8. A similar scan of the identifier table is made for automatic storage and controlled storage. The scan enters relative location for each entry. An "automatic" entry and a "controlled"entry are also made in the matrix. Code generation use the relative location entry to generate the address part of instructions. No storage is generated at compile time for automatic or controlled. However, the matrix entry automatic does cause code to be generated that allocates this storage at execution time, i.e., when the generated code is executed, it allocates automatic storage.	***(correct explanation: 4 marks)***

		  **The purpose of this phase is to: (optional)** 1. Assign storage to all variables referenced in the source program. 2. Assign storage to all temporary locations that are necessary for intermediate result, e.g. the results of matrix lines. These storage references were reserved by the interpretation phase and did 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)	
	5)	**Explain the concept of subroutine linkages.**	**4M**
	Ans:	• A main program 'A' wishes to transfer to subprogram 'B'. The programmer, in program 'A', could write a transfer instruction (e.g. BAL 14 B) to subprogram 'B'. However, the assembler does not know the value of this symbol reference and will declare it as an error (undefined symbol) unless a special mechanism has been provided. • This mechanism is typically implemented with a relocating or a direct linking. • The assembler pseudo-op EXTRN followed by a list of symbol indicates that these symbols are defined in other programs but referenced in the present program. • Correspondingly, if a symbol is defined in one program and referenced in others, we insert it into symbol list following the pseudo-op ENTRY. • In turn the assembler will inform the loader that these symbols may be referenced by other programs. • For examples, the following sequence of instructions may be a simple calling sequence to another program: MAIN START EXTRN SUBROUT …….. …….. L 15=A(SUBROUT)…..CALL SUBROUT BAIR 14, 15 .. .. END • The above sequence of instructions first declares SUBROUT as an external variable, that is variable referenced but not defined in this program. • The load instruction loads the address of that variable into 15. • The BALR instruction branches to the contents of register 15, which is the address of SUBROUT, and leaves the value of the next instruction in register 14.	***(correct explanation: 4 marks)***