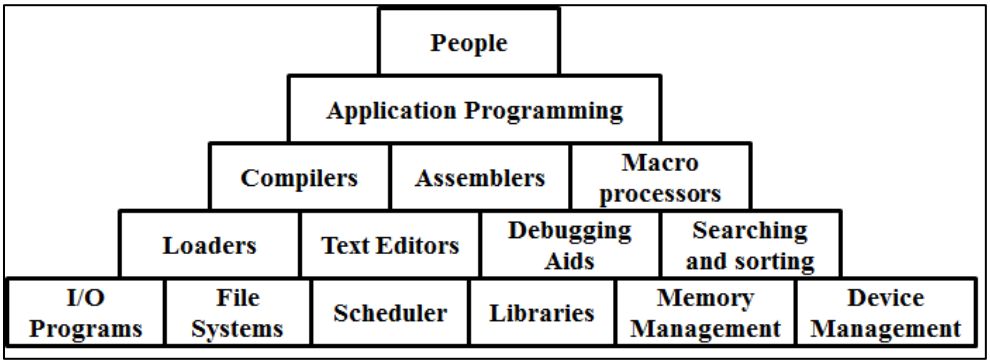
SUMMER– 18 EXAMINATION

Subject Name: System Programming **Model Answer** **Subject Code:** 17634

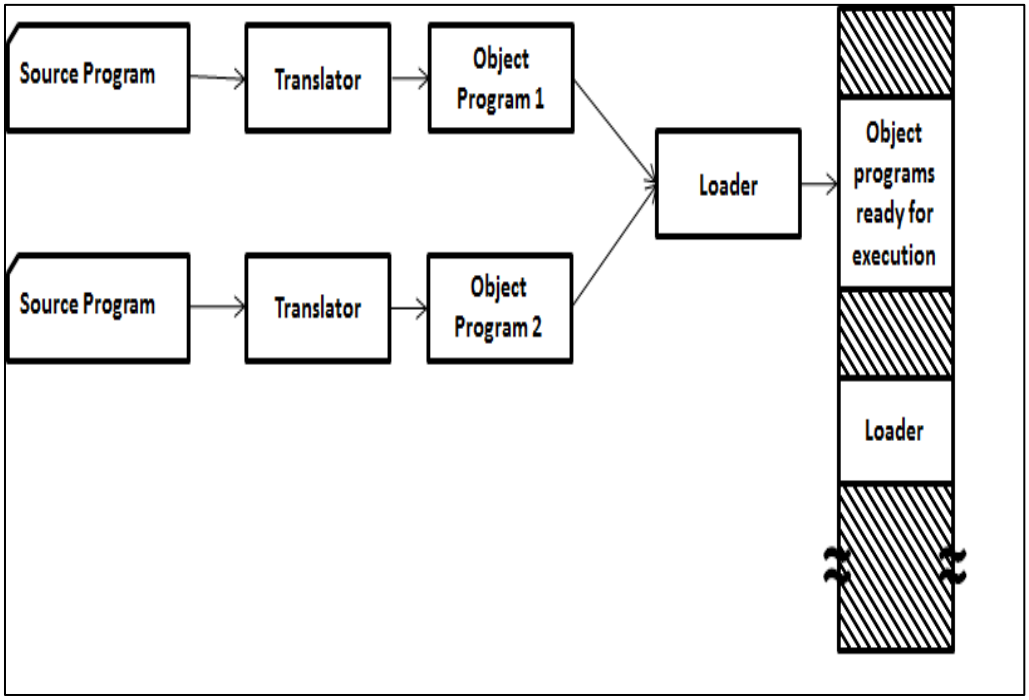
Important Instructions to examiners:

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.

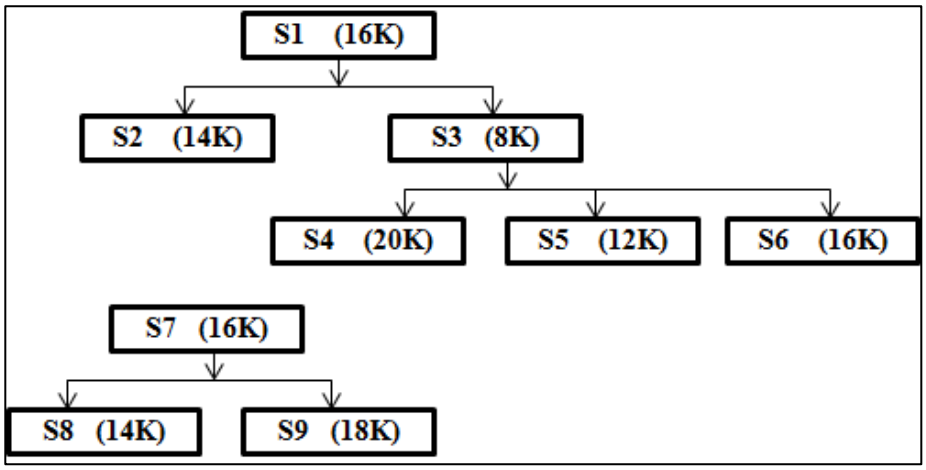
Q. No.	Sub Q. N.	Answers	Marking Scheme
1.		**Answer any FIVE of the following:**	**20 Marks**
	(a)	**Draw a neat labelled diagram of foundation of system programming.**	**4M**
	Ans:		**(Diagram :4 marks)**
	(b)	**What is binary search? Explain with an example.**	**4M**
	Ans:	**Binary Search Algorithm**: A more systematic way of searching an ordered table. This technique uses following steps for searching a keywords from the table. **1.** Find the middle entry (N/2 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.	**(Algorithm: 2 marks, Example: 2 marks)**

		4. The next action taken for each of these outcomes is as follows 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. **Example:** 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 7 **Pass 2** Low= (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.	
	(c)	**What is meant by implementation of macro call within macros? Give an example.**	**4M**
	Ans:	• The macro can be used within macro. The macro or macro calls are "abbreviations" of the sequence of instruction. • Therefore these "abbreviation" should be available within other macro definition. • To handle macros calls within macros, the macro processor must be able to work recursively. Recursively Procedures implemented with the help of stack. • Stack a storage scheme that allocates a separate storage area for variable associated with each call to the procedure **Syntax:** MACRO MACRO_NAME1 --- --- MEND MACRO MACRO_NAME2 MACRO_NAME1 //calling macro within macro -- -- MEND **Example:** MACRO ADD &A1 L 1,&A1 ST 1,&A1 MEND MACRO ADDITION &A1,&A2	**(Description: 2 marks, Example:2 marks)**

		ADD &A1 ADD &A2 MEND • Above code shows two macros ADD and ADDITION. • Within the definition of ADDITION macro, macro ADD is called two times with different parameter A1 and A2. • Use of macro within macro result in macro expansion on multiple levels. Such way the macro within macro involves several levels.	
	(d)	**State the six phases of compiler.**	**4M**
	Ans:	**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. **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. **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 are performed by compiler, machine dependent and machine independent. **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 machine language. **7) Assembly phase:-** • The compiler has to generate the machine language code for computer to understand.	**(6 phases of compiler: 4 marks)**

	(e)	**Draw a labelled diagram of general loading scheme.**	**4M**
	Ans:		**(Labeled diagram :4 marks)**
	(f)	**Mention the necessity of overlays in linking loader.**	**4M**
	Ans:	• Sometimes a program may require more storage space than the available one Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. • In such situations only those segments are resident in the memories that are actually needed at the time of execution. • But the question arises what will happen if the required segment is not present in the memory? • Naturally the execution process will be delayed until the required segment gets loaded in the memory. The overall effect of this is efficiency of execution process gets degraded. • The efficiency can then be improved by carefully selecting all the interdependent segments. The assembler cannot do this task. Only the user can specify such dependencies. • The inter dependency of the segments can be specified by a tree like structure called static overlay structures. • The overlay structure contains multiple root/nodes and edges. Each node represents the segment. The specification of required amount of memory is also essential in this structure. • The two segments can lie simultaneously in the main memory if they are on the same path. • Consider following example to understand the concept. In this, various segments along with their memory requirements are shown.	**(Relative Explanation: 4 marks)**

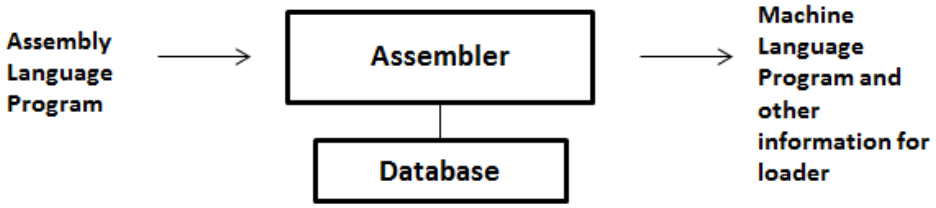
- Nodes S1, and S7 are root nodes.
- Nodes S2 and S3 are interdependent nodes; so, lie simultaneously in the main memory. Similarly, S4, S5, and S6 are on the same path therefore lie simultaneously in the main memory. In this way overlay structure is formed.



(g) Define the terms: Assemblers and Compiler.

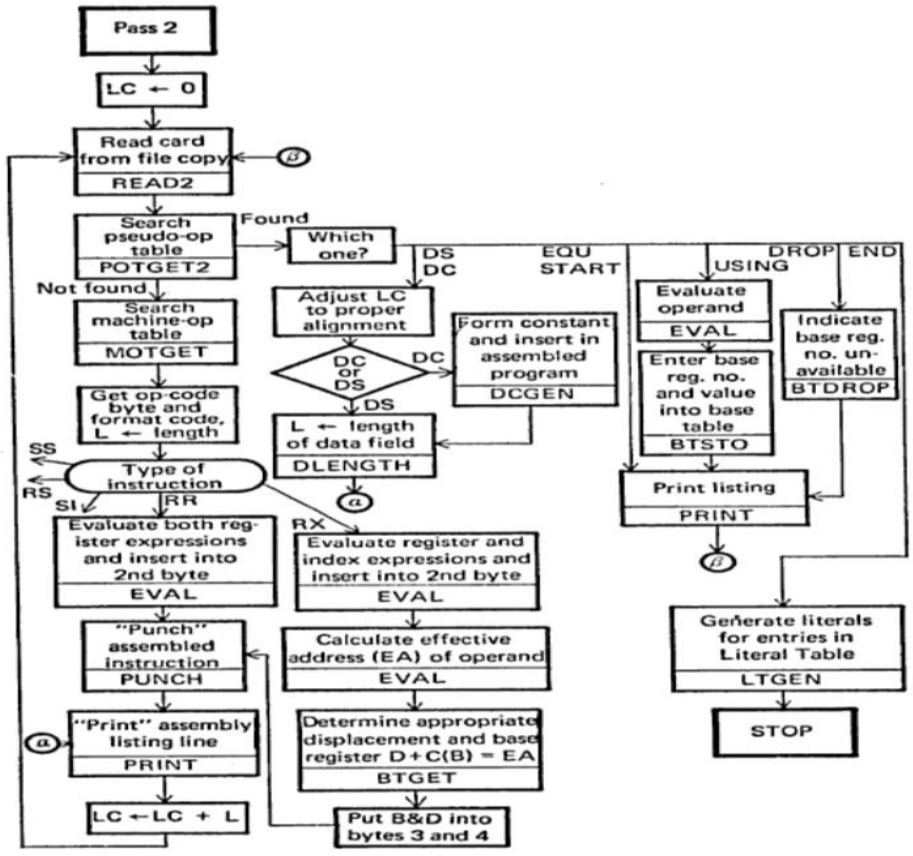
Ans:

- **Assemblers:**
  - The program known as assembler is written to automate the translation of assembly language to machine language. Input to the language is called as source program and output of assembler is machine language translation called as object program.



- **Compilers:**
  - A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).The most common reason for converting a source code is to create an executable program. E.g. Javac , TurboC, CC (used in Unix/Linux).

(Assembler: 2 marks, Compiler: 2 marks)

2.		**Answer any TWO of the following :**	**16 Marks**
	(a)	**Draw the flowchart for pass 2 of a two pass assembler.**	**8M**
	Ans:		**(Correct diagram:8 marks)**
	(b)	**Describe the design of absolute loader with respect to its performance based on (1) Allocation, (2) Loading, (3) Relocation, (4) Linking.**	**8M**
	Ans:	• Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. • **This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler.** • The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. • In this scheme, the programmer or assembler should have knowledge of memory management. • The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. • The programmer should take care of two things: o First thing is: specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next module, it's then	**(Description: 4 marks, example with performance parameters: 4 marks)**

the programmer's duty to make necessary changes in the starting addresses of respective modules.

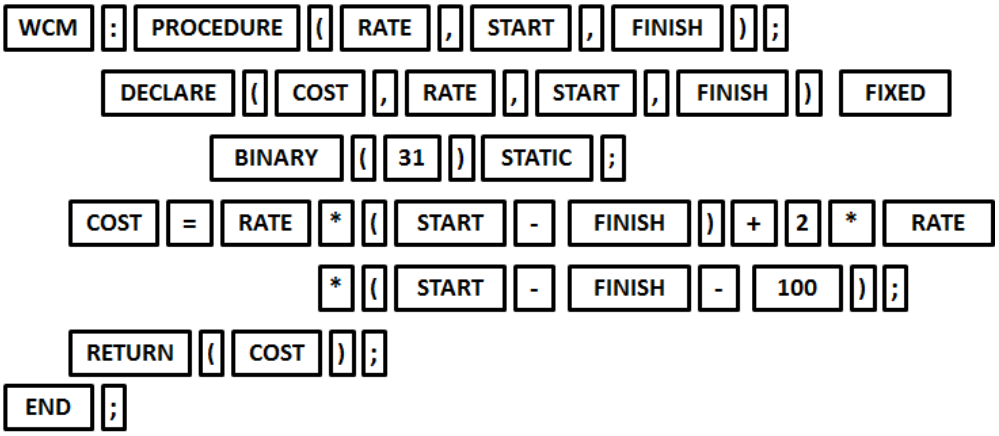
- Second thing is, while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction.

- **For example**

Line number

```
1        MAIN        START       1000
.                    .
.                    .
.                    .
1                    JMP         5000
16                   STORE       ;instruction at location 2000
                     END
1                    SUM         START 5000
2
20                   JMP         2000
21                   END
```

- In this example there are two segments, which are interdependent.
- At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN.
- Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment.
- The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. **(ALLOCATION)**
- During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of the mutually dependent dresses. **(LINKING)**
- As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. **(RELOCATION)**
- The task of assembler is to create the object codes for the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution.
- The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses. **(LOADING)**

	(c)	**Describe token with respect to lexical analysis with a suitable example and classify the tokens.**	**8M**
	Ans:	• The first phase of compiler is lexical analysis. It works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as: ***<token-name, attribute-value>*** • Algorithm of Lexical Analysis phase of compiler is as follows 1. The first tasks of the lexical analysis algorithm are to the input character string into token. 2. The second is to make the appropriate entries in the tables. 3. A token is a substring of the input string that represents a basic element of the language. It may contain only simple characters and may not include another token. To the rest of the compiler, the token is the smallest unit of currency. Only lexical analysis and the output processor of the assembly phase concern themselves with such elements as characters. Uniform symbols are the terminal symbols for syntax analysis. • Lexical analysis recognizes three types of token: Terminal symbols, possible identifiers, and literals. • It checks all tokens by first comparing them with the entries in the terminal table. Once a match is found, the token is classified as a terminal symbol and lexical analysis creates a uniform symbol of type "TRM" and inserts it in the uniform symbol table. If a token is not a terminal symbol, lexical analysis proceeds to classify it as a possible identifier or literal. Those tokens that satisfy the lexical rules for forming identifiers are classified as "possible identifiers". **Example:** **Consider following program** WCM: PROCEDURE (RATE, START, FINISH); DECLARE (COST, RATE, START, FINISH) FIXED BINARY (31) STATIC; COST = RATE * (START-FINISH) + 2*RATE*(START-FINISH-100); RETURN (COST); END; 	**(Description :4 marks, Example: 2 marks, Classification :2 marks)**

Class	PTR(TOKENS)
IDN	WCM
TRM	:
TRM	PROCEDURE
TRM	(
IDN	RATE

Uniform symbols of first statement

Classification of TOKENS:

Tokens are classified as

1. **Terminal symbols:** Keywords or assembler directives are referred to as terminal symbols. For e.g. in above program PROCEDURE, DECLARE, RETURN, END,*, (, etc are identified as terminal symbols.
2. **Possible identifiers:** a user defined string that identifies specific variables or procedures, etc. For e.g. in above program RATE, COST, START, FINISH are identifiers.
3. **Literals:** a constant value is referred as literal. For e.g. in above program 2, 100, 31 are literals.

3.		**Answer any FOUR of the following:**	**16 Marks**
	(a)	**Write in brief about any two components of system software.**	**4M**
	Ans:	The components of system software are 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. 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	**(Description of any 2 components : 2 marks each)**

		program (Higher level program) and generates the target program (Assembly language program or machine language program).	
	(b)	**Explain random entry searching with an example.**	**4M**
	Ans:	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. 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** 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 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.	(**Description: 2marks, Example: 2 marks)**

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 $T_n = 54/16 = 3.37$

(c) Describe the four tasks performed by Macro-processor. **4M**

Ans: The basic task of Macro processor is as follows:- **(Description of 4 tasks: 1 mark each)**

1) Recognize the macro definitions.
2) Save the Macro definition.
3) Recognize the Macro calls.
4) Perform Macro Expansion.

1) **Recognize the Macro definitions:** - A macro processor 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.

2) Save the Macro definition: - The processor must store the macro instruction definitions which it will need for expanding macro calls.

3) Recognize the Macro calls: - The processor must recognize macro call that

appears as operation mnemonics. This suggests that macro names be handled as a type of opcode.

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.

(d) For the following sub-expression, draw the intermediate code with optimization: z = (x + y) * (x + y) + 3 (x + y). 4M

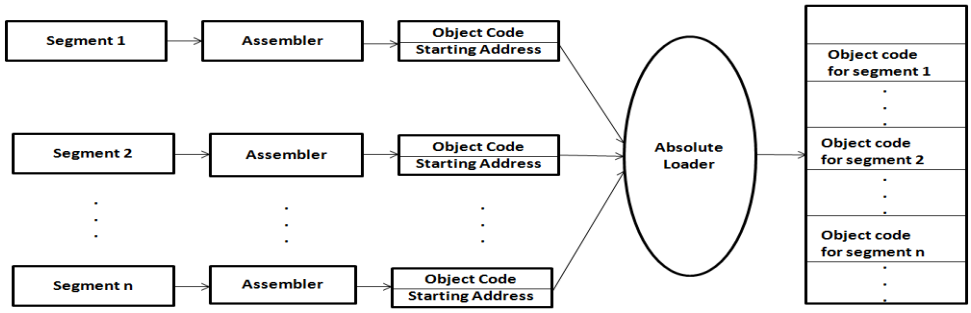
Ans:

MATRIX NO	OPERATOR	OP1	OP2	First Try	
M1	+	X	Y	L	1, X
				A	1, Y
				ST	1, M1
M2	*	M1	M1	L	1, M1
				M	1,M1
				ST	1,M2
M3	*	3	M1	L	1,=F'3'
				M	M1,3
				ST	M1,M3
M4	*	M2	M3	L	1,M2
				M	M2,M3
				ST	M2,M4
M5	=	Z	M4	L	1, M4
				ST	1,Z

(Optimized table:2 marks, Intermediate code: 2 marks)

(e) Describe the design of absolute loader. 4M

Ans:



Design of absolute loader

(Diagram: 2marks, Description: 2 marks)

Absolute Loader: Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines is the issues which need to be handled by the programmer. The programmer should take care of two things: first thing is: specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next. Modules, it's then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is, while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

```
Line number
1          MAIN      START          1000
.                      .
.                      .
.                      .
15                     JMP            5000
16                     STORE          ;instruction at location 2000
                       END
1          SUM       START          5000
2
20                     JMP            2000
21                     END
```

In this example there are two segments, which are interdependent. At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN. Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment. The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of the mutually dependent dresses. As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codes for the above segments and along with the information such as starting address of the

		memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses. Thus the absolute loader is simple to implement in this scheme 1) Allocation is done by either programmer or assembler 2) Linking is done by the programmer or assembler 3) Resolution is done by assembler 4) Simply loading is done by the loader 5) As the name suggests, no relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler **Advantages:** 1) It is simple to implement 2) This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution. 3) The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory. 4) The process of execution is efficient. **Disadvantages:** 1) In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.	
	(f)	**What are the data structures required to implement direct linking loader.**	**4M**
	Ans:	Databases required for Pass 1 and Pass 2 of direct linking loader with their purposes listed below: **Pass 1** 1. Input object decks 2. A parameter, the Initial Program Load Address (IPLA) supplied by the programmer or the operating system that specifies the address to load the first segment. 3. A Program Load Address (PLA) counter, used to keep track of each segment's assigned location. 4. A table, the Global External Symbol Table (GEST), that is used to store each external symbol and its corresponding assigned core address. 5. A copy of the input to be used later by pass 2. This may be stored on an auxiliary storage device, such as magnetic tape, disks or drum, or the original objects deck may be reread by the loader a second time for pass 2. 6. A printed listing, the load map that specifies each external symbol and its assigned value.	**(Any 6 Data structures: 4 marks)**

Pass 2

1) Copy of object program inputted to pass1
2) The Initial Program Load Address parameter (IPLA)
3) The Program Load Address Counter (PLA)
4) The Global External Symbol Table (GEST), prepared by pass1, containing each external symbol and its corresponding absolute address value.
5) An array, the Local External Symbol array (LESA), which is used to establish a correspondence between the ESD ID numbers, used on ESD and RLD cards, and the corresponding external symbols' absolute address value.

4. Answer any FOUR of the following: — 16 Marks

(a) Compare shell sort and address calculations sort. — 4M

Ans: **(Any 4 Points of Comparison: 1 mark each)**

Sr. No.	Shell Sort	Address Calculation Sort
1	Avg. Time(approx.) = $B*N*(\log_2(N))^2$	Avg. Time(approx.) = E*N
2	Extra Storage = none	Extra Storage = 2.2*N (approx.)
3	It requires floors and hence work in divide and conquer method.	It follows linear approach for solution.
4	It is slower than address calculation sort	It is faster than other method if space is available
5	It compare with their distances	It compare with their addresses

(b) Describe the issues in implantation of macroprocessor within an assembler. — 4M

Ans: **(Any two issues/Disadvantages: 2 marks each)**

{{Note: - Diagram is optional**}}**

Implementation within an assembler.

Issues of incorporating the macro processor into pass 1 of assembles.

- The program becomes too large to fit into the core of some machines.
- The program becomes complex in action with macros

The macro processor can be implemented within an assembler using two different ways, they are.

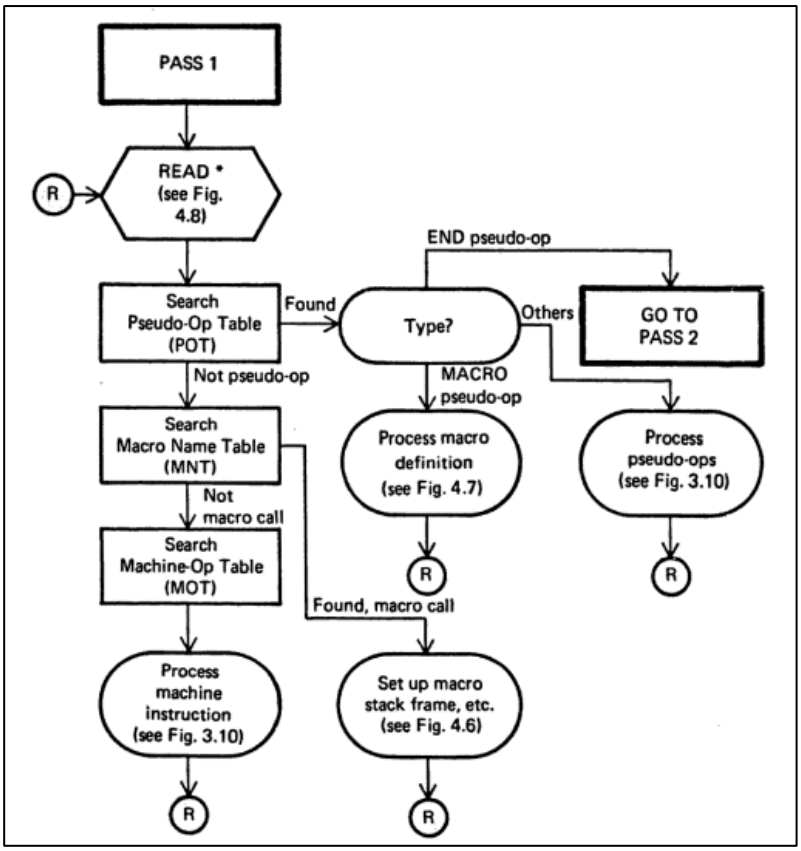
1. It can be added as a macro processor to an assembler, making a complete a pass over the input text before pass 1 of the assembler

Or

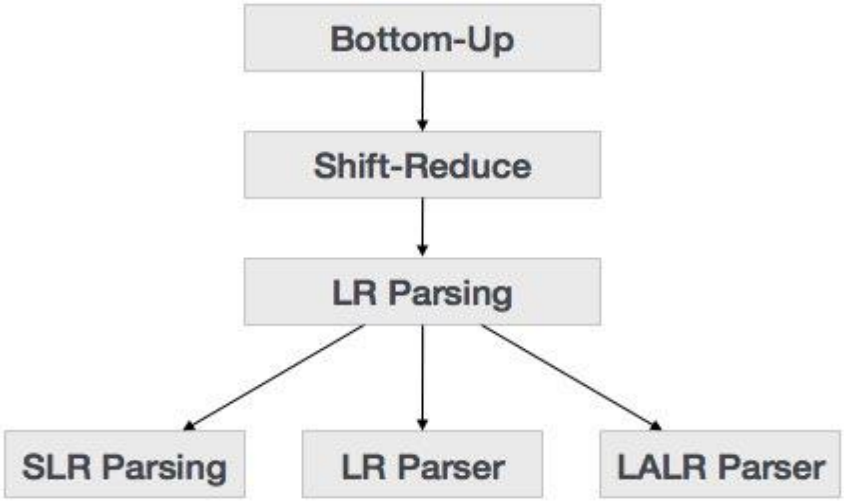
2. It can be implemented within pass 1 of the assembler.

Macro processor combined with assembler pass - 1.

The implementation of the macro processor within pass 1 eliminates the overhead of intermediate files, and can improve this integration of macro processor and assembler by combining similar functions. For example the assembler regular pseudo -ops

		handles can be used to identify MACRO pseudo -ops of the macro processor. The macro name table can be combined with the assembler machine op table or pseudo -op table.  **Flowchart of a macro processor combined with assembler part 1** **Advantage and incorporating the macro processor into pass 1 of assembler.** Many functions do not have to be implemented twice. • There is less over head during processing: functions are combined and it is not necessary to create intermediate files as output from the macro processor and input to the assembler. • Move flexibility is available to the programmer in that he may use all the features of the assembler in conjunction with macros.	
	(c)	**Write about bottom up parsing technique and how it differs from top down parsing.**	**4M**
	Ans:	**Bottom-up parsing:-** 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.	**(Description: 2 marks, Any 2 points of differentiatio**

n: 2 marks)



Shift-Reduce Parsing: Shift-reduce parsing use two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

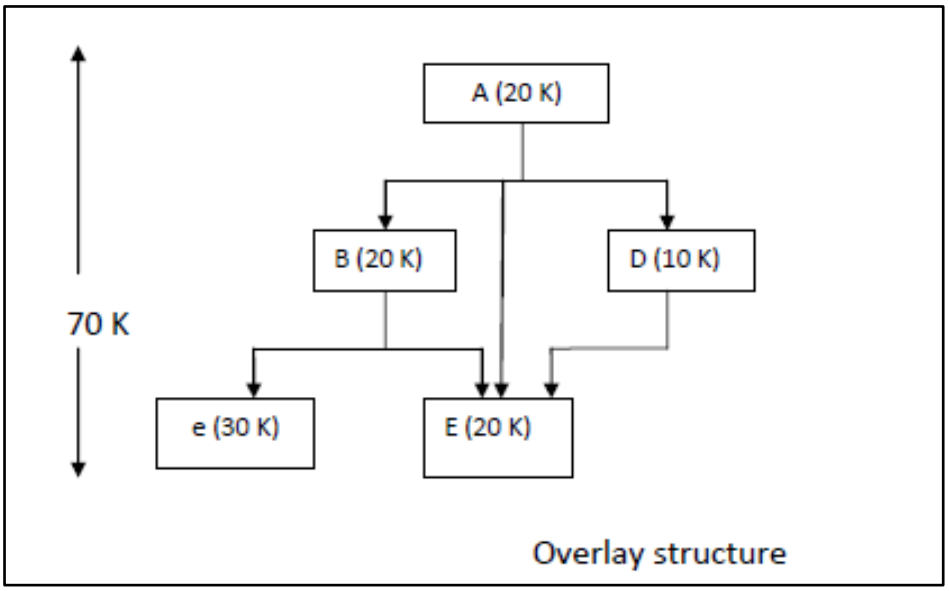
LR Parser: The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LRk parsers, where **L** stands for left-to-right scanning of the input stream; **R** stands for the construction of right-most derivation in reverse, and **k** denotes the number of look ahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

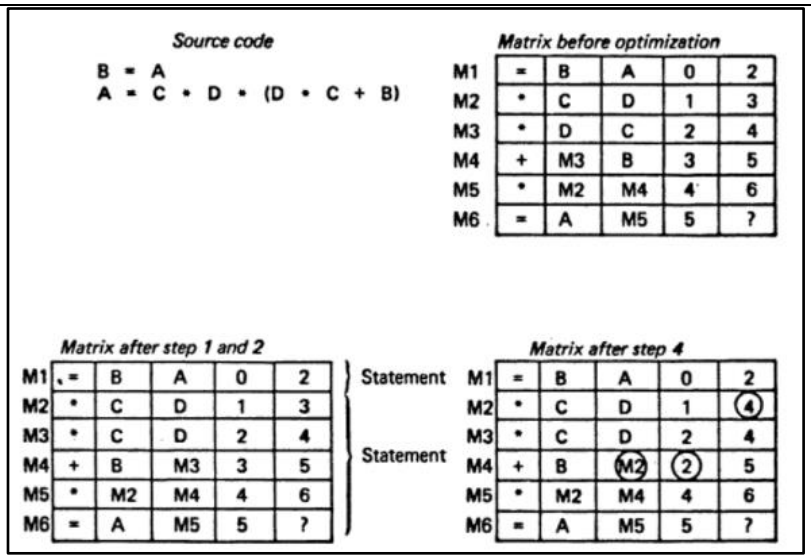
- SLR1 – Simple LR Parser:
- LR1 – LR Parser:
- LALR1 – Look-Ahead LR Parser:

Sr.	Top – down parsing	Bottom up parsing
1)	It is easy to implement	It is efficient parsing method
2)	It can be done using recursive decent or LL(1) parsing method	It is a table driven method and can be done using shift reduce, SLR, LR or LALR parsing method
3)	The parse tree is constructed from root to leaves	The parse tree is constructed from leaves to root
4)	In LL(1) parsing the input is scanned from left to right and left most derivation is carried out	In LR parser the input is scanned from left to right and rightmost derivation in reverse is followed
5)	It cannot handle left recursion	The left recursive grammar is handled by this parser
6)	It is implemented using recursive routines	It is a table driven method
7)	It is applicable to small class of grammar	It is applicable to large class of grammar

	(d)	**Write the purpose of storage allocation and interpretation phase.**	**4M**
	Ans:	**The purpose of storage allocation is to:** 1) Assign storage to all variables referenced in the source program. 2) Assign storage to all temporary locations that are necessary for intermediate results, 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. **The purpose of interpretation phase is to:** 1) This phase is typically a collection of routines that are called when a construct is recognized in syntactic phase. 2) The purpose of these routines is to on intermediate form of the source program and add information to identifier table. It interprets the precise meaning into the matrix or identifier table.	**(Purpose of storage allocation : 2 marks and interpretation phase: 2 marks)**
	(e)	**Describe what is dynamic binding .**	**4M**
	Ans:	In dynamic binding, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external ·reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is resumed. Thus dynamic binding both the loading and linking is done dynamically. **Advantages** 1) The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution. 2) The system can be dynamically reconfigured. **Disadvantages** 1) The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory	**(Description: 4 marks)**
	(f)	**Write what is meant by overlays. Explain with a diagram.**	
		Overlay: An overlay is a part of a program (or software package) which has the same load origin as some other part(s) of the program. Overlays are used to reduce the main memory requirement of a program. **Explanation:** The subroutines of a program are needed at different times. For e.g. Pass 1 and pass 2 of an assembler are call other subroutine it is possible to produce an overlay structure that identifiers mutually exclusive subroutines. In order for the overlay structure to work it is necessary for the module loader	**(Description: 2 marks, Diagram with its description: 2 marks)**

		to the various procedures as they are needed. The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is collect the overlay supervision or simply the upper.  Overlay structure Above program consisting of five subprogram (A, B, C, D & E) that require look bytes of core. The arrow indicate that subprogram A only calls B, D and E; subprogram B only calls C and E; subprogram D only calls E; and subprogram C and E do not call any other routine procedures B and D are never in use at the same time; neither are C and E. If are load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure. This happens to be 70k. Overlay reduces the memory requirement of a program. It also makes it possible to execute program where size exceeds the amount of memory which can be allocated to them. For the execution of overlay structured program, the root is loaded in memory and given control for the execution. Other overlays are loaded as and when headed. Loading of an overlay overwrite a previously loaded overlay with the same load origin.	
5.		**Answer any FOUR of the following:**	**16 Marks**
	(a)	**Explain the significance of System Programming.**	**4M**
	Ans:	**{{**Note: Any relevant answer shall be considered**}}** 1. System programming, as an operating system, compiler, or utility program that controls some aspect of the operation of a computer 2. It deals with computer components like registers and memory locations. 3. It is useful to control and manage computer systems 4. It is concerned with data transfer, reading from and writing to files, compiling, linking, loading, starting and stopping programs, and even fiddling with the individual bits of a small word of memory. 5. It deals with writing device drivers and operating systems, or at least directly using	**(Any 4 Significance: 1 mark each)**

		them; programmers exploit this low-level knowledge. 6. Generally feature extremely small run-time images, because they often have to run in resource constrained environments 7. If implemented properly, can be very efficient: to take advantage of the hardware. 8. System programs can sometimes be written to extend the functionality of the operating system itself and provide functions that higher level applications can use.	
	(b)	**Write the issues in implementation of a single pass macro processor.**	**4M**
	Ans:	1 It requires additional variables as Macro Definition Input (MDI) and Macro Definition Level Counter (MDLC) and its status needs to be maintained. 2 While performing Macro Definition pass simultaneously with "Macro Expansion pass there must be two separate Argument List Arrays maintained. 3 Separate Read Sub routine needs to be maintain.	**(Any 2 Issues: 2 marks each)**
	(c)	**Write four methods of machine independent optimization**	**4M**
	Ans:	The possible algorithm for four optimization techniques are as follows:- 1) Elimination of common sub expression 2) Compile time compute. 3) Boolean expression optimization. 4) Move invariant computations outside of loops. **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 sub-expression must be identical and must be in the same statement. i. The elimination algorithm is as follows:- ii. Place the matrix in a form so that common sub-expression can be recognized. iii. Recognize two sub-expressions as being equivalent. iv. Eliminate one of them. v. After the rest of the matrix to reflect the elimination of this entry.	(**4 methods: 1 mark each)**



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.- A = 2 * 276 / 92 * B

The compile time computation would be

Matrix Before optimization

M_1	*	2	276
M2	/	M_1	92
	*	M_2	B
M3	=	A	M_3
M4			

Matrix After optimization

M_1			
M_2			
M_3	*	6	B
M_4	=	A	M_3

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.

4) **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.

1. Recognition of invariant computation.
2. Discovering where to move the invariant computation.
3. Moving the invariant computation.

(d) Explain with an example how linear search is performed. **4M**

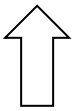
Ans: In Linear Search algorithm, it start with the first element in the list. Compare the current element to the key element, if the current element matches the key element then it is declare search found and stop. If the current element is not equal to the key element then set the current element to be the next element and repeat above sequence till end. At the end if element is not found then simply declare element is not exist in list. **(Description : 2 marks, Example: 2 marks)**

Example: -

78	64	54	75	47	34	46

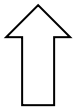
Search Number: 54 in the list, i.e. key = 54

78	64	54	75	47	34	46



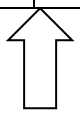
No Match; Take next number:

78	64	54	75	47	34	46

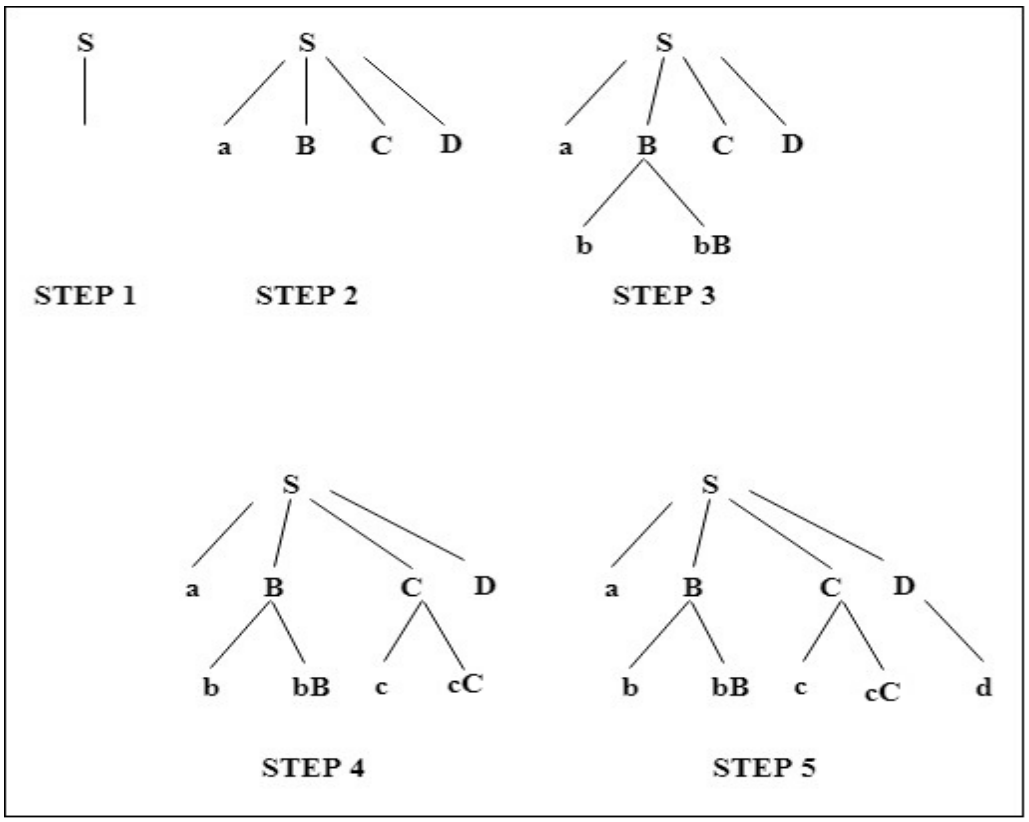


No Match: Take next number:

78	64	54	75	47	34	46



Match Found (Target = 54)
Search found Stop

	(e)	**Define Parser. Draw the parse tree for the string abbccd.**	
		Parser: - It is a system software that construct the parsing tree to identify syntactical errors in given statement. It can generate tree in top down approach or bottom up approach. Assume Given grammar as follows: - S ⟶ xyz | aBCD B ⟶ b|bB C ⟶ c|cC D ⟶ d  	**(Definition: 2 marks, Parse tree: 2 marks)**
	(f)	**Explain how Grammar is used for finding syntactic error in syntax analysis phase of compiler.**	
		1. Reductions are tested consecutively for match between Old Top of Stack field and the actual Top of Stack, until match is found. 2. When Match is found, the action routine specified in the action field are executed in order from left to right. 3. When control returns to the syntax analyser, it modifies the Top of the Stack to agree with the New Top of Stack field. 4. Step 1 is then repeated starting with the reduction specified in the next reduction field.	**(4 steps : 1 mark each)**

6.		Answer any TWO of the following:	16 Marks
	(a)	**Sort the given numbers in descending order using radix exchange sort. Show the steps:** **78, 387, 42, 09, 12, 881**	8M
	Ans:	**Pass 1:** **Step 1: -** Equalize numbers for 3 digit. 078, 387, 042, 009, 012, 881 **Step 2: -** Put Numbers in associated place. Consider LSB, i.e. unit position.	**(Pass 1: 3 marks, Pass 2: 3 marks, Pass 3: 2 marks)**

	0	1	2	3	4	5	6	7	8	9
078									078	
387								387		
042			042							
009										009
012			012							
881		881								

Step 3: - Retrieve the data in reverse sequence.
009, 078, 387, 042, 012, 881
Pass 2:
Step 4: - Put Numbers in associated place. Consider 10^{th} position (Since numbers needs to arrange in descending order arrange cells in descending order.

	0	1	2	3	4	5	6	7	8	9
009	009									
078								078		
387									387	
042					042					
012		012								
881									881	

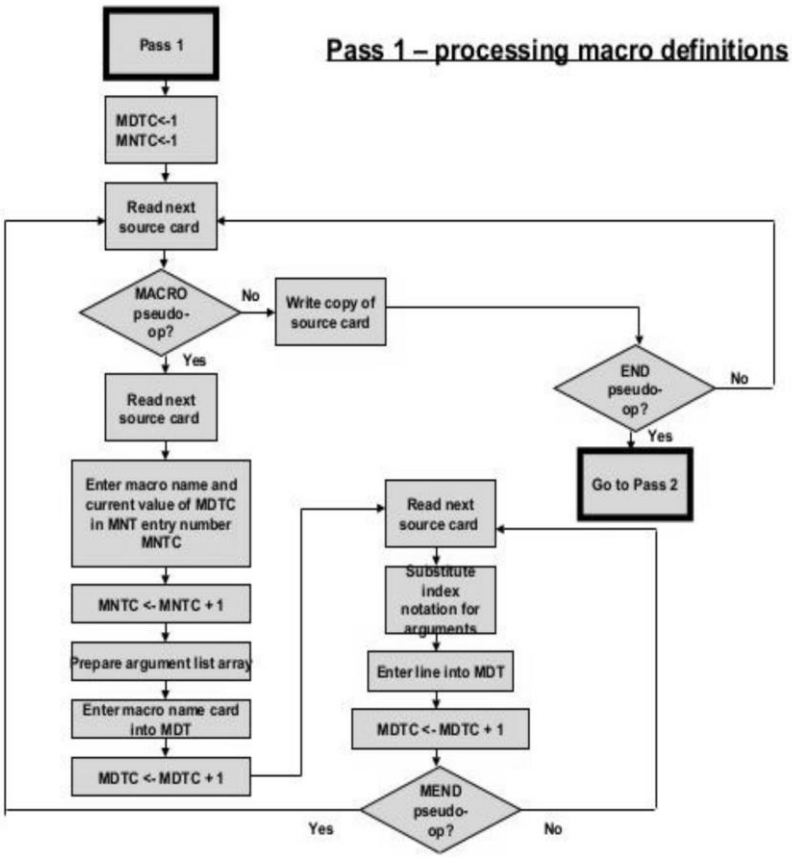
Step 5: - Retrieve the data in reverse sequences.
387, 881, 078, 042, 012, 009
Pass 3:
Step 6: - Put Numbers in associated place. Consider 100^{th} position (Since numbers needs to arrange in descending order arrange cells in descending order.

	0	1	2	3	4	5	6	7	8	9
387				387						
881									881	
078	078									
042	042									
012	012									
009	009									

Step 7: - Retrieve the data in reverse sequences.
881, 387, 078, 042, 012, 009

(b) Draw a flow chart of Pass I of a two pass macroprocessor. **8M**

Ans: **(Correct flow chart: 8 marks)**



	(c)	**Describe the databases used in lexical, syntactic and Symantic phases of compiler.**	**8M**
	Ans:	**Databases used in Lexical Phase:** 1. Source Program: Original form of program; appears to the compiler as a string of characters. 2. Terminal table: A permanent database that has an entry for each terminal symbol. Each entry consists of the terminal symbol, an indication of its classification and its precedence. 3. Literal Table: - Created by lexical analysis to describe all literal 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. 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 the identifier into that entry. Since in many languages identifiers may be from 1 to 31 symbols long the lexical phase may enter a pointer in the identifier table for efficiency of storage. 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 the token is a member. **Databases used in Syntactic and Semantic Phase of Compiler:-** 1. Uniform Symbol Table: - Created by lexical phase and containing the source program in the form of uniform symbols. It is used by the syntax and interpretation phases as the source of input to the stack. Each symbol from the UST Enters the stack only once.	**(Database for Lexical Phase: 5 marks, Syntactic and Semantic Phase Database: 3 marks)**

Symbol	Indicator	Precedence

Literal	Base	Scale	Precision	Other Information	Address

Name	Data Attributes	Address

Table	Index

Table	Index

		2. Stack: - The stack is the collection of uniform symbols that is currently being worked on by the syntax analysis and interpretation phases. Additions to or deletions from the stack are made by the phases that use it. The stack is organized on LIFO basis. Term Top of stack refers to the most recent entry and Bottom of Stack refers to the oldest of entry. 3. Reductions: - The syntax rules of the source language are contained in the reduction table. The syntax analysis phase in an interpreter driven by the reductions. The general form of the rule of reduction is ***Label: Old Top of Stack / Action Routines/ New Top of Stack/ Next Reduction***	