MODEL ANSWER

SUMMER– 17 EXAMINATION

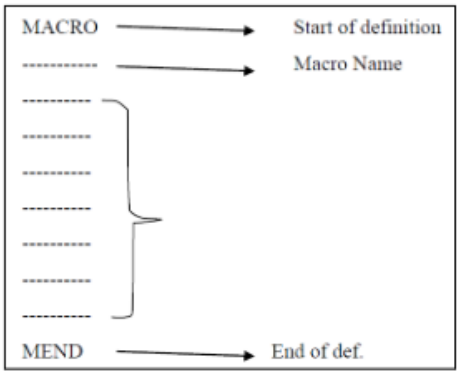
Subject Title: System Programming **Subject Code:** 17517

Important Instructions to examiners:

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

Q. No	Sub Q. N.	Answer	Marking Scheme
1.	**a)**	**Attempt any three:**	**4x3=12 Marks**
	1)	**State and explain the functions of loader.**	**4M**
	Ans:	a) **Allocation**: Allocate the space in the memory where the object programs can be loaded for execution. b) **Linking**: Resolving external symbol reference c) **Relocation**: Adjust the address sensitive instructions to the allocated space. d) **Loading**: Placing the object program in the memory in to the allocated space.	**(State: 1 mark and explain: 3 marks)**
	2)	**What is an assembler? What are its functions?**	**4M**
	Ans:	Assembler: Assembler 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. Functions of Assembler:- 1. Generate machine instructions • evaluate the mnemonics to produce their machine code • evaluate the symbols, literals, addresses to produce their equivalent machine addresses • convert the data constants into their machine representations 2. Process pseudo operations Assembly language is a machine dependent language so it is also called as low level programming language.	**(Definition: 2 marks and Functions: 2 marks)**

			
	3)	**Explain the design steps of assembler.**	**4M**
	Ans:	**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. 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. **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. 4. **Specify algorithm:** Specify algorithms to define symbols and generate code 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 6. Repeat 1 to 5 on modules	**(All steps: 4 marks)**
	4)	**How will you recognize basic elements in compiler?**	**4M**
	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: 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	**(Explanation : 4 marks)**

		classify it as a possible identifier or literal. Those tokens that satisfy the lexical rules for forming identifiers are classified as "possible identifiers".	
	b)	**Attempt any one:**	**6x1= 6Marks**
	1)	**What is macro? Enlist advantages of macro.**	**6M**
	Ans:	**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 advantage of macro facility where macro is defined as "Single line abbreviation for group of instructions". A macro instruction is a notational convenience for the programmer, It allows the programmer to write shorthand version of a program (module programming). The macro processor replaces each macro invocation with the corresponding sequence of statements (expanding) A macro represents a commonly used group of statements in the source programming language The macro processor replaces each macro instruction with the corresponding group of source language statement, this is called expanding macros.  **Advantages:** Macros are single line abbreviation for groups of instructions. For every occurrence of this one. Line macro instruction, the macro processing assemble will substitute the entire block. By defining the appropriate macro instruction can assembly language programmer can tailor his own higher level facility in a convenient manner, at no cost in control over the structured of his program. He can achieve the conciseness and ease in coding of high level language without losing the basic advantage of assembly language programming. Integral macro operation simplifies debugging and program modification and they facilitate standardization. • The speed of the execution of the program is the major advantage of using a macro. • It saves a lot of time that is spent by the compiler for invoking / calling the functions • It reduces the length of the program	**(Definition: 2 marks, four Advantages: 4 marks)**
	2)	**Enlist and explain the features of macro processor.**	**6M**
	Ans:	• Macro instruction argument. • Conditional macro expansion • Macro call within macros. • Macro instruction defining macros.	**(List:2 marks, Explanation of each:1 mark)**

Macro instruction Arguments: A macro definition is enclosed between a macro header statement and a macro end statement. Macro definitions are typically located at the start of a program. A macro definition consists of.

1. A macro prototype statement:-The macro prototype statement declares the name of a macro and the names and kinds of its parameters.
2. One or more model statements
3. Macro pre-processor statements

It has the following syntax
<macro name> [< formal parameter spec > [,..]]
Where <macro name> appears in the mnemonic field of an assembly statement and

<formal parameter spec> is of the form

&<parameter name> [<parameter kind>]

Parameter Substitution – Example

Source

```
STRG   MACRO       &a1, &a2, &a3
               STA     &a1
               STB     &a2
               STX     &a3
               MEND
       .
STRG   DATA1, DATA2, DATA3
       .
STRG   DATA4, DATA5, DATA6
       .
       .
```

Expanded souce

```
       .
       .
       .

       STA     DATA1
               STB     DATA2
               STX     DATA3
       .
               STA     DATA4
               STB     DATA5
               STX     DATA6
```

Conditional Macro Expansion: Most macro processors can also modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation. Conditional Assembly is commonly used to describe this feature. Two important macro processor pseudo-ops, AIF and AGO, permit conditional reordering of the sequence of macro expansion. This allows conditional selection of the machine instructions that appear in expansions of a macro call.

Consider the following program.

```
                .
                .
                .
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   D C   F '5'
DATA 2   D C   F '10'
DATA 3  D C   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
& ARGO VARY & COUNT, & ARG1, &ARG2, &ARG3
& ARGO   A          1, &ARG1
          AIF (& COUNT EQ1).FINI
          A          2, & ARG2
          AIF (& COUNT EQ2).FINI
          A          3,  & ARG3
       .FINI MEND
                .
                .
                .
LOOP1    VARY    3, DATA1, DATA2, DATA3   { loop 1 A1, DATA1
                .                           {      A2, DATA2
                .                           {      A3, DATA3
                .
LOOP2    VARY    1, DATA1                 { loop 2 A1,DATA3
                .                           {      A2, DATA2
                .
                .
      DATA 1  D C  F '5'
      DATA 2  D C  F '10'
      DATA 3  D C  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 macroprocessor to skip to the statement. Labelled .FINI if the parameter corresponding

to & COUNT is a1; otherwise the macroprocessor 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 macroprocessor expands the statements in macro instructions.

Macro calls within macros: 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.
Syntax:

```
MACRO
MACRO_NAME1
---
---
MEND
MACRO
MACRO_NAME2
MACRO_NAME1
--
--
MEND
```

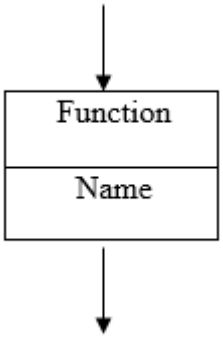
EXAMPLE:

```
MACRO
ADD         &A1
L           1,&A1
ST          1,&A1
MEND

MACRO
ADDITION    &A1,&A2
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.
Defining macro within macro definition: Macro definition is not defined called as callable until after the outer macro has been called. This is because the method by which the definition are implemented.
Example:-

```
 MACRO
 DEFINE      &SUB
```

		MACRO &SUB &Y L 1,&Y ST 1,&Y MEND MEND When user call above macro with the statement DEFINE COS Defines new macro CAS, the stamen expands into a new macro definition. The user might subsequently call the CAS macro as follows: COS AR And macro processor will generate calling sequence.	
2.		**Attempt any two:**	**8x2=16Marks**
	1)	**Explain the term: Look for modularity.**	**8M**
	Ans:	We know our design, looking for functions that can be isolated. Typically, such functions into two categories: (1) multi-use and (2) unique. In the flowchart for pass 1 and pass 2 .we examine each step as a candidate for logical separation. Likely choices are identified in the flowcharts by the shape.  Where "name" is the name assigned to the function (e.g., MOTGET ,EVAL ,PRINT). Listed below are some of the functions that may be isolated in the two passes: **Pass 1:** 1. READ1: Read the next assembly source card. 2. POTGET1: Search the pass 1 Pseudo- op Table (POT) for a match with the operation field of the current source card. 3. MOTGET1: Search the Machine-op Table (MOT) for a match with the operation of the current source card. 4. STSTO: Store a label and its associated value into the symbol Table (ST). if the symbol is already in the table, return error indication (multi-ply-defined symbol). 5. LTSTO: Store a literal into the Literal Table.(LT) : do not store the same literal twice .	**(Appropriate Description : 8 marks)**

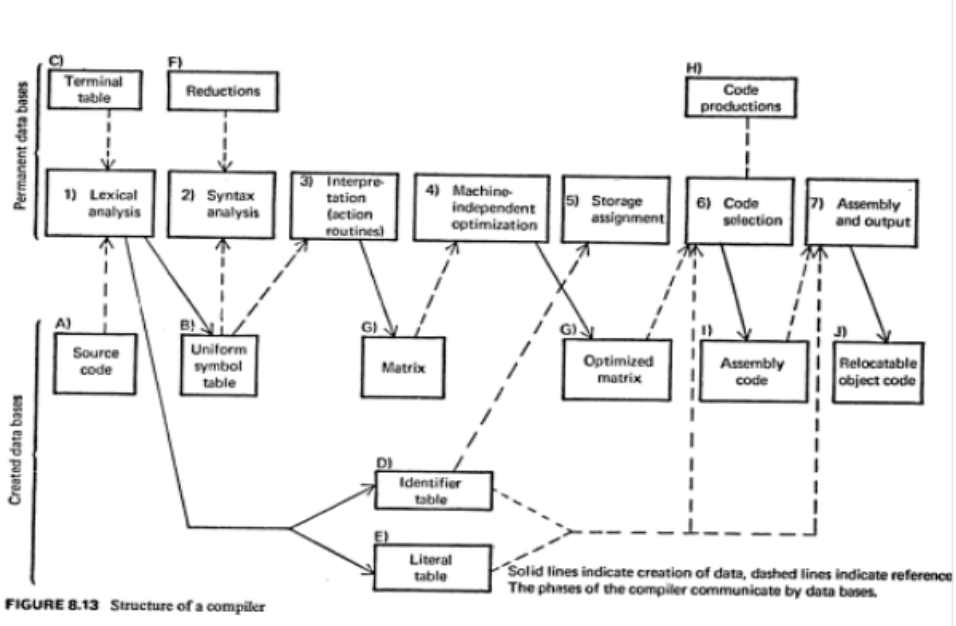
6. WRITE1: Write a copy of the assembly source card on a storage device for use by pass 2.
7. DELENGHT: Scan operand field DS or Dc pseudo-op to determine the amount of storage required.
8. EVAL: Evaluate an arithmetic expression consisting of constant and symbols (e.g 6, ALPHA, BETA+ 4 * GAMMA).
9. STGET: Search the Symbol TABLE (ST) for the entry corresponding to a specific symbol (used by STSTO, and EVAL).
10. LITASS: Assign storage locations to each literal in the literal table (may use DLENGHT)

PASS 2:

1. READ2: Read the next assembly source card from the file copy.
2. POTGET 2 : Similar to POTGET1 (search POT)
3. MOTGET2: Same as in pass 1 (search MOT)
4. EVAL: Same as in pass 1 (evaluate expressions).
5. PUNCH: Convert generated instruction to card format; punch card when it is filled with data.
6. PRINT--- Covert relative location and generated code to character format: print the line slong with copy of the source card.
7. DCGEN------- Process the fields of the DC to generate object code (uses EVAL and PUNCH).
8. DLENGTH--- Same as in pass 1.
9. BTSTO--- Insert data into appropriate entry of Base Table (BT)
10. BTDROP----- Insert “unavailable” indicator into appropriate entry of
11. BT.BTGET ----- Convert effective address into base and displacement by searching Base Table (BT) for available base registers.
12. LTGEN------Generate code for literals(uses DCGEN)

Each of these functions should independently go through the entire design process (problem statement, data bases, algorithm, modularity, etc.). These functions can implemented as separate external subroutines, as internal subroutines, or as sections of the pass 1 and pass 2 programs. In any case, the ability to treat functions separately makes it much easier to design the structure of the assembler and each of its parts. Thus, rather than viewing the assembler as a single program (of from 1,000 to 10,,, source statements typically), we view it as coordinated collection of routines each of relatively minor size and complexity. We will not attempt to examine all of these functional routines in details since they are quite straightforward. There are two particular observations of interest: (1) several of the routines involve the scanning or evaluation of fields (e.g, DLENGTH,EVAL,DCGEN): (2) several other routines involve the processing of tables by storing or searching (e.g. PRTGET1, POTGET2,MOTGET1, OTGET2, LTSTO,STSTO,STGET).The section of this book dealing with the implementation of compilers will discuss techniques for parsing fields and evaluating arithmetic expressions, many of which are also applicable to the functional modules of

		the assembler. Table processing, as discussed in regard to assembler implementation, is found in almost every type of system program, including compilers, loaders, file systems and operating systems, as well as in many application programs. The general topic of processing data structures and data organization plays a crucial role in systems programming. Since storing and searching for entries in tables often represent the largest expenditures of time in an assembler, the next section examiners some techniques for organizing these tasks.	
	2)	**Explain different data structures used by pass-II assembler.**	**8M**
	Ans:	The various data structure used is as follows:- i) Copy file:- It is prepared by pass 1 to be used by pass 2. ii) Location counter:- It is used to assign address to instruction and addressed to symbol defined in the program. iii) Machine operation Table (MOT) or Mnemonic Opcode Table (MOT):- It is used to indicate for each instruction. a) Symbolic mnemonic b) Instruction length c) Binary machine opcode. d) Format. iv) **Pseudo-operation Table (POT):-** It indicate for each pseudo-op the symbolic mnemonic and action to be taken in pass 2 **Or** It is consulted to process pseudo like DS, DC, Drop & using. v) **Symbol Table (ST):-** It is used to generate the address of the symbol address in the program. vi) **Base table (BT)**:- It indicate which registers are currently specified as base register by USING Pseudo-ops. vii) **INST workspace:-** It is used for holding each instruction and its various parts are being getting assembled. viii) **PUNCH CARD workspace:**- It is used for punching (outputting) the assembled instruction on to cards ix) **PRINT LINE workspace:-** It is used for generating a printed assembly listing for programmers reference. x) **Object card:-** This card contain the object program in a format required by the loader.	**(Any Eight: 1 mark each)**

	3)	**Draw the basic phases of compiler and explain each phase function.**	**8M**
	Ans:	 **FIGURE 8.13** Structure of a compiler The different phases of compiler are as follows:- **1) Lexical Phase:** i. 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. ii. The reading or parsing of source program is called as scanning of the source program. iii. It recognizes keywords, operators and identifiers, integers, floating point numbers, character strings and other similar items that form the source program. The lexical analyzer collects information about tokens in to their associated attributes. **2) Syntax Phase:** i. In this phase the compiler must recognize the phases (syntatic construction); each phrase is semantic entry and is a string of tokens that has meaning, and 2nd Interpret the meaning of the constructions. ii. 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). iii. There are many ways of recognizing the basic constructs and interpreting the meaning. iv. Syntax analysis uses rules (reductions) which specify the syntax form of source language. v. These reductions define the basic syntax construction and appropriate compiler routine (action routine) to be executed when a construction is recognized. vi. The action routine interprets the meaning and generates either code or intermediate form of construction. **3) Interpretation Phase:** This phase is typically a routine that are called when a construct is recognized. The purpose of these routines is to on intermediate form of source program and adds information to identifier table.	**(Diagram: 4 marks and function of each phase:½ mark)**

		4) Code optimization Phase: Two types of optimization is performed by compiler, machine dependent and machine independent. Machine dependent optimization is so intimately related to instruction that the generated. It was incorporated into the code generation phase. Where Machine independent optimization is was done in a separate optimization phase. **5) Storage Assignment:** The purpose of this phase is as follows:- i. Assign storage to all variables referenced in source program. ii. Assign storage to all temporary locations that are necessary for Intermediate results. iii. Assign storage to literals. iv. Ensure that storage is allocated and appropriate locations are initialized. **6) Code generation:** i. This phase produce a program which can be in Assembly or machine language. ii. This phase has matrix as input. iii. It uses the code production in the matrix to produce code. iv. It also reference the identifier table in order to generate address & code conversion. **7) Assembly phase:** The compiler has to generate the machine language code for computer to understand. The task to be done is as follows:- i. Generating code ii. Resolving all references. iii. Defining all labels. iv. Resolving literals and symbolic table.	
3.		**Attempt any four:**	**4x4=16Marks**
	1)	**What are the four components of system software?**	**4M**
	Ans:	**Assembler:** 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. **ALP → ASSEMBLER → Machine Language equivalent + Information required by the loader** **Loader:** Loader is a system program which places program into the memory and prepares for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code. Eg. Boot Strap loader. **Macro:** A macro is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to a replacement output sequence (also often a sequence of characters) according to a defined procedure. The mappings process that instantiates (transforms) a macro use into a specific sequence is known as macro expansion. A facility for writing macros may be provided as part of a software application or as a part of a programming language. In the former case, macros are used	**(1 mark for Each Component)**

to make tasks using the application less repetitive. In the latter case, they are a tool that allows a programmer to enable code reuse or even to design domain-specific languages.

```
MACRO
MACRO_NAME
…
…
…
MEND
```

Compiler: 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. Eg. Javac , TurboC, CC (used in Unix/Linux).

2) Apply radix sort on following numbers: 170,45,75,90,2,24,802,66. **4M**

Ans: **(Correct Solution: 4 marks)**

Original table	**First distribution**	**Result of pass 1**	**Second Distribution**	**Result of pass 2**	**Third Distribution**	**Result of pass 3**
	0) 170,90		0) 02, 802		0) 002, 024, 045, 066, 075, 090	
170	1)	170	1)	2	1) 170	2
45	2) 2, 802	90	2) 24	802	2)	24
75	3)	2	3)	24	3)	45
90	4) 24	802	4) 45	45	4)	66
2	5) 45, 75	24	5)	66	5)	75
24	6) 66	45	6) 66	170	6)	90
802	7)	75	7) 170, 75	75	7)	170
66	8)	66	8)	90	8) 802	802
	9)		9) 90		9)	

3) Give the examples of arithmetic and non-arithmetic statements which can be use in compiler operation. **4M**

Ans: The interpretation phase converts statements into matrix representation. There are two types of statements; Arithmetic statements and non-arithmetic statements. **(Arithmetic: 2 marks ,**

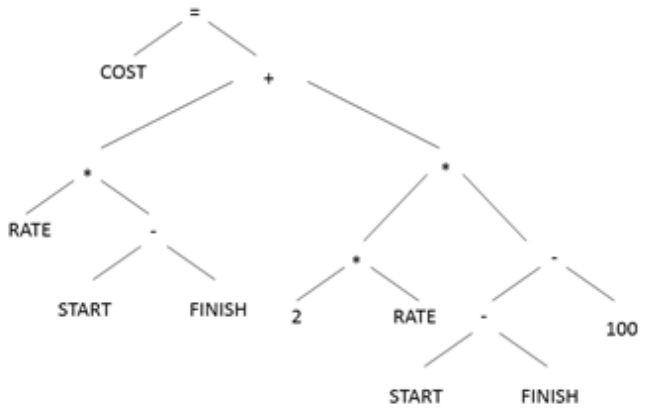
Arithmetic statements

•One form is to convert these statements into the **PARSE TREE**.

•The rules for converting these statements into parse tree are:

1. Any variable is a terminal node of the tree
2. For every operator, construct a binary tree (in order dictated by the rules of algebra), whose left branch is a tree for operand 1, and right branch is a tree for operand 2.

COST	=	RATE	*	(	START	-	FINISH	)	+	2	*	RATE	*	(	START	-	FINISH	-	100	)	;



Matrix of the parse tree:

- Linear representation of parse tree called **MATRIX**
- Operators of the program are listed sequentially in order they would be executed.
- Each matrix entry has one operator and two operands. The operands are uniform symbols denoting either variable, literals, or other matrix entries M_i
- Linear representation of parse tree called **MATRIX**
- Operators of the program are listed sequentially in order they would be executed.
- Each matrix entry has one operator and two operands. The operands are uniform symbols denoting either variable, literals, or other matrix entries M_i

Matrix line No.	Operator	Operand 1	Operand 2	Matrix entries
1	-	START	FINISH	M1
2	*	RATE	M1	M2
3	*	2	RATE	M3
4	-	START	FINISH	M4
5	-	M4	100	M5

Non-Arithmetic Statements: 2 marks)

6	*	M3	M5	M6	
7	+	M2	M6	M7	
8	=	COST	M7		

Non Arithmetic statements

- The non-arithmetic executable statements are replaced with the sequential ordering of individual matrix entries.

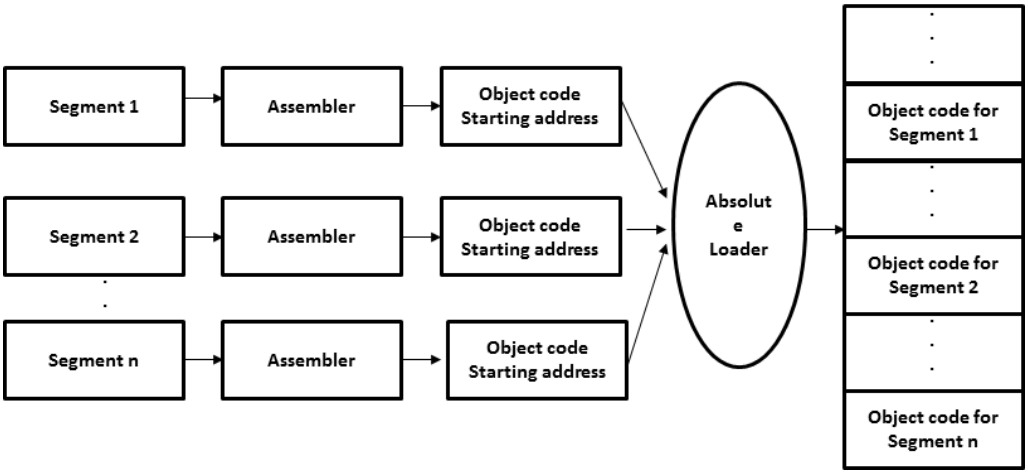
Example:
RETURN COST
END

Matrix representation:

Operator	**Operand 1**	**Operand 2**
Return	COST	
END		

4) **Explain absolute loader scheme.** **4M**

Ans: **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 are 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, its 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. **(Explanation : 4 marks)**



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

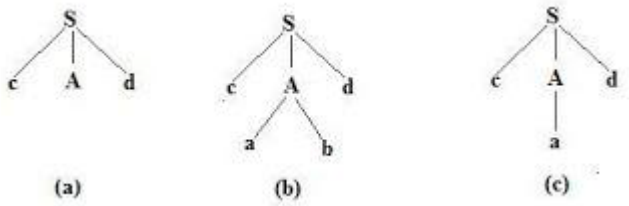
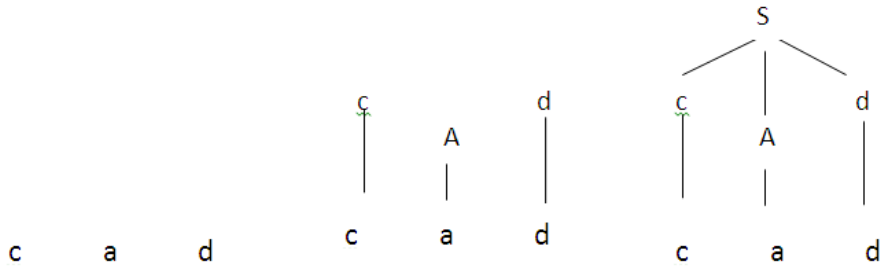
Advantages:

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:

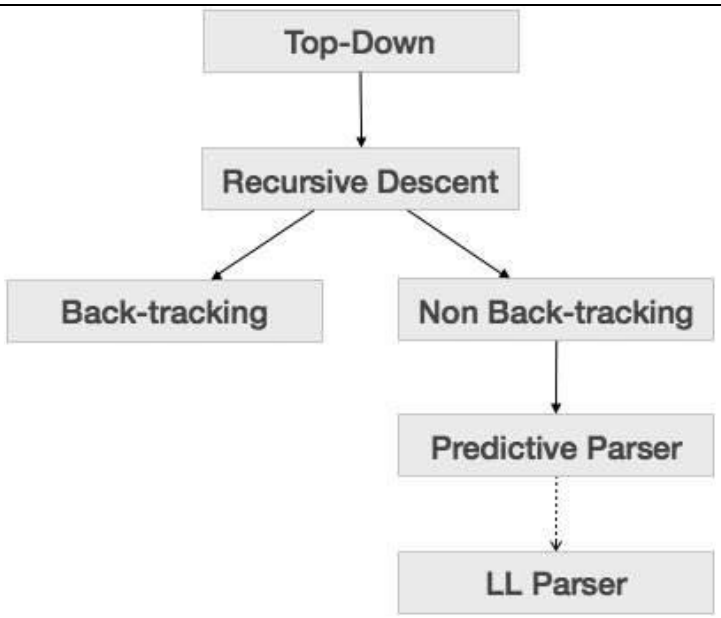
1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.

	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.	**(Top down parser: 2 marks, Bottom up Parser: 2 marks)**

		Consider the grammar S → cAd A → ab | a and the input string w = cad  **Top Down Parser**  **Bottom up parser**	
4.	**a)**	**Attempt any three:**	**4x3=12Marks**
	1)	**What is the algorithm for direct linking loader?**	**4M**
	Ans:	**Algorithm:-** **Pass1: Allocate segment and defines symbols.** 1. Start of pass 1 2. Initially program local address (PLA) is set to initial program load address (IPLA) 3. Read object card. 4. Write a copy of source card for pass2 5. Check card type A. If TXT or RLS card, no processing from pass1. So read next card (go to step 3) B. If an EDS card then, check type of external symbol I. If SD then VALUE = PLA, SLENGTH= LENGTH Ii. If ER then read next card go to step 3 Iii. If LD then VALUE = PLA+ADDR 6. If symbol is already in GEST	**(Algorithm:4 Mark, Any 1 algorithm shall be considered)**

		A. If yes then ERR: duplicate use of START and ENTERY name B. If no then the symbols and assigned values are stored in GEST C. Write symbol name and value for load map. 7. Stop. **Pass 2:** STEP 1: START STEP 2: PLA = IPLA STEP 3: EXADDR = IPLA STEP 4: READ CARD FROM FILE COPY STEP 5: CHECK CARD TYPE IF CARD == ESD THEN CHECK ESD CARD TYPE IF LD THEN GO TO STEP 4 ELSE IF SD THEN SLENGTH = LENGTH SET LESA (ID) = PLA) GOTO STEP 4 ELSE SEARCH GEST FOR SYMBOL IF FOUND SET LESA (ID) = VALUE GOTO STEP NO 4; ELSE PRINT ERROR ELSE IF CARD ==TXT THEN MOV BC BYTES FROM CARD COLOUMN 17-72 TO LOCATION (PLA +ADDR) GOTO STEP 4. ELSE IF CARD = RLD THEN GET VALUE FROM LESA(ID) IF FLAG == + THEN ADD VALUE TO CONTENTS OF LOCATION PLA + ADDRESS GO TO STEP 4. ELSE SUB VALUE TO CONTENTS OF LOCATION PLA + ADDRESS GO TO STEP 4. ELSE IF CARD == END THEN IF ADDR != NULL EXADDR = (PLA + ADDR) PLA = PLA+SLENGTH GO TO STEP 4 ELSE PLA = PLA+SLENGTH GO TO STEP 4 ELSE TRANSFER TO LOCATION EXADDR STEP 6: STOP.	
	2)	**Explain functions of lexical analyzers.**	**4M**
	Ans:	The three tasks of lexical analysis phase are 1. To parse the source program in to basic elements or tokens of the language. 2. To build a literal table and an identifier table. 3. To build a uniform symbol table. • The first task of lexical analyzer is to parse the input character string into tokens. o In this step, the input string is separated into tokens by break character. o The lexical analyzer recognizes three types of tokens: terminal symbols, possible identifiers, and literals. • The second is to make the appropriate entries in the tables. o It checks all tokens by first comparing them with the entries in terminal table. o Once the matches found, the token is classified as the terminal symbol and lexical analyzer creates a uniform symbol of type “TRM”, and insert it in uniform symbol table.	**(Appropriate Description: 4 Marks, Any relevant description shall be consider)**

		○ If token is not terminal symbol, lexical analyzer proceeds to classify it as possible identifier or literal ○ After token is classified as a "possible identifier", the identifier table is examined. If the particular entry is not in the table, then new entry is made. • The third is to build a uniform symbol table. ○ Based on the literals and identifiers identified in second step, uniform symbol entry is made in "uniform symbol" table.	
	3)	**Explain assembly phase of compiler in detail.**	**4M**
	Ans:	• After code generation phase, next phase is assembly phase. The task of assembly phase depends on how much has been done in code generation. • If a lot of work has been done in code generation, then the assembly phase must resolve labels references in object program, format the object deck, and format the appropriate information for the loader. • If code generation has simply generated symbolic machine instructions and labels, the assembly phase must (1) resolve label references, (2) calculate addresses, (3) generate binary machine instructions, and (4) generate storage, convert literals. **Databases**: 1. Identifier table: assembly phase uses this database to enter the value of all labels into identifier table. 2. Literal table: places the literal on appropriate TXT cards 3. Object code: the output of code generation. **Algorithm**: 1. A simple assembly phase scans the object code, resolving all label references and producing the TXT cards. 2. It then scans the identifier table to create the ESD cards. 3. The RLD cards are created using the object code, the ESD cards and the identifier table	**(Explanation : 2 marks, Databases:1 mark Algorithm: 1 mark)**
	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**: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production. Top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:	**(Description: 2 marks, 2 marks for Description of Any two type of top down parser)**



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 →ai

For an input string: read, a top-down parser, will behave like this: It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea). Now the parser matches all the input letters in an ordered manner. The string is accepted.

Predictive Parser: Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking. To accomplish its tasks; the

predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar. Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol **$**to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

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

b) Attempt any one: **6x1=6Marks**

1) Explain the database used by Pass-1 of two pass macro processor. **6M**

Ans: **(1 mark for Each Database)**

The various databases used by first pass are:
The input macro source deck.

1. The output macro source deck copy that can be used by pass 2.
2. The Macro Definition Table (MDT), which can be used to store the body of the macro definitions. MDT contains text lines and every line of each macro definition, except the MACRO line gets stored in this table. For example, consider the code described in macro expansion section where macro INC used the macro definition of INC in MDT. Table 2.1 shows the MDT entry for INC macro:

Macro Definition Table

& LAB	INCR	& ARG1,&ARG2,&ARG3
#0	A	1, #1
	A	2,#2
	A	3,#3
	MEND	

3. The Macro Name Table (MNT), which can be used to store the names of defined macros. Each MNT entry consists of a character string such as the macro name and a pointer such as index to the entry in MDT that corresponds to the beginning of the macro definition. Table 2.2 shows the MNT entry for INCR macro:

Macro Name Table

1	“INCRbbbb”	15
.	.	.
.	.	.
.	.	.
.	.	.

4. The Macro Definition Table Counter (MDTC) that indicates the next available entry in the MDT.
5. The Macro Name Table Counter (MNTC) that indicates the next available entry in the MNT.
6. The Argument List Array (ALA) that can be used to substitute index markers for dummy arguments prior to store a macro definition. ALA is used during both the passes of the macro pre-processor. During Pass 1, dummy arguments in the macro definition are replaced with positional indicators when the macro definition is stored. These positional indicators are used to refer to the memory address in the macro expansion. It is done in order to simplify the later argument replacement during macro expansion. The ith dummy argument on the macro name card is represented in the body of the macro by the index marker symbol #. The # symbol is a symbol reserved for the use of macro pre-processor.

Index	Argument
0	"bbbbbbbb" (all Blank)
1	"Data3bbb"
2	"Data2bbb"
3	"Data1bbb"

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

Ans:

Top Down Parser:

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 parse
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:

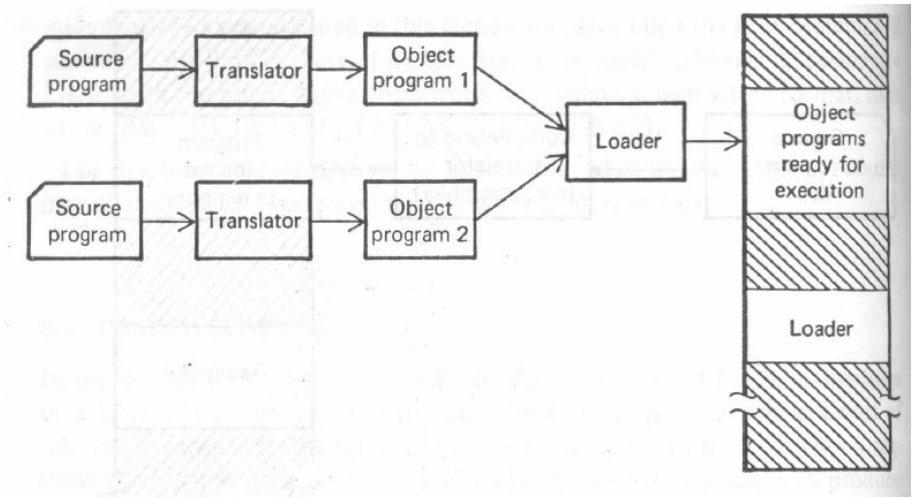
Advantages:-

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.

(Any Six points of comparison: 1 mark each)

5.		**Attempt any two:**	**8x2=16marks**
	1)	**Which disadvantages of compile-and-loader are removed in general loader scheme? Explain.**	**8M**
	Ans:	The Disadvantages of compile-and-Go Loaders A portion of memory is wasted because the core occupied by the assembler is unavailable to the object program. **General Loader scheme:** Outputting the instructions and data as they are assembled circumvents the problem of wasting core for the assembler. Such an output could be saved and loaded whenever the code was to be executed. The assembled programs could be loaded into the same area in core that the assembler occupied (since the translation will have been completed). This output form, which may be on cards containing a coded form of the instructions, is called an object deck. The use of an object deck as intermediate data to avoid one disadvantages of the proceeding compile and Go scheme requires the addition of a new program to the system a loader. The loader accepts the assembled machine instructions data, and other information present in the object format and places machine instructions and data in core in an executable computer form. The loader is assumed to be smaller than the assembler, so that reassembly is no longer necessary to run the program at a later date. Finally, if all the source program translators (assemblers and compilers) produce compatible object program deck formats and use compatible linkage conventions, it is possible to write subroutine in serval different languages since the object decks to be processed by the loader will all be in the same "language" (machine Language)  **Fig:General loader scheme**	**(Identifying Disadvantage: 2 marks, Description: 4 marks, Diagram: 2 marks)**
	2)	**Explain machine independent and machine dependent compiler optimization.**	**8M**
	Ans:	Two types of optimization is performed by compiler, machine dependent and machine independent. Machine dependent optimization is so intimately related to instruction that the generated. It was incorporated into the code generation phase. Whereas Machine independent optimization was done in a separate optimization phase.	**(Machine dependent:4 marks,Machine Independent:**

Machine- independent optimization: **4 marks)**

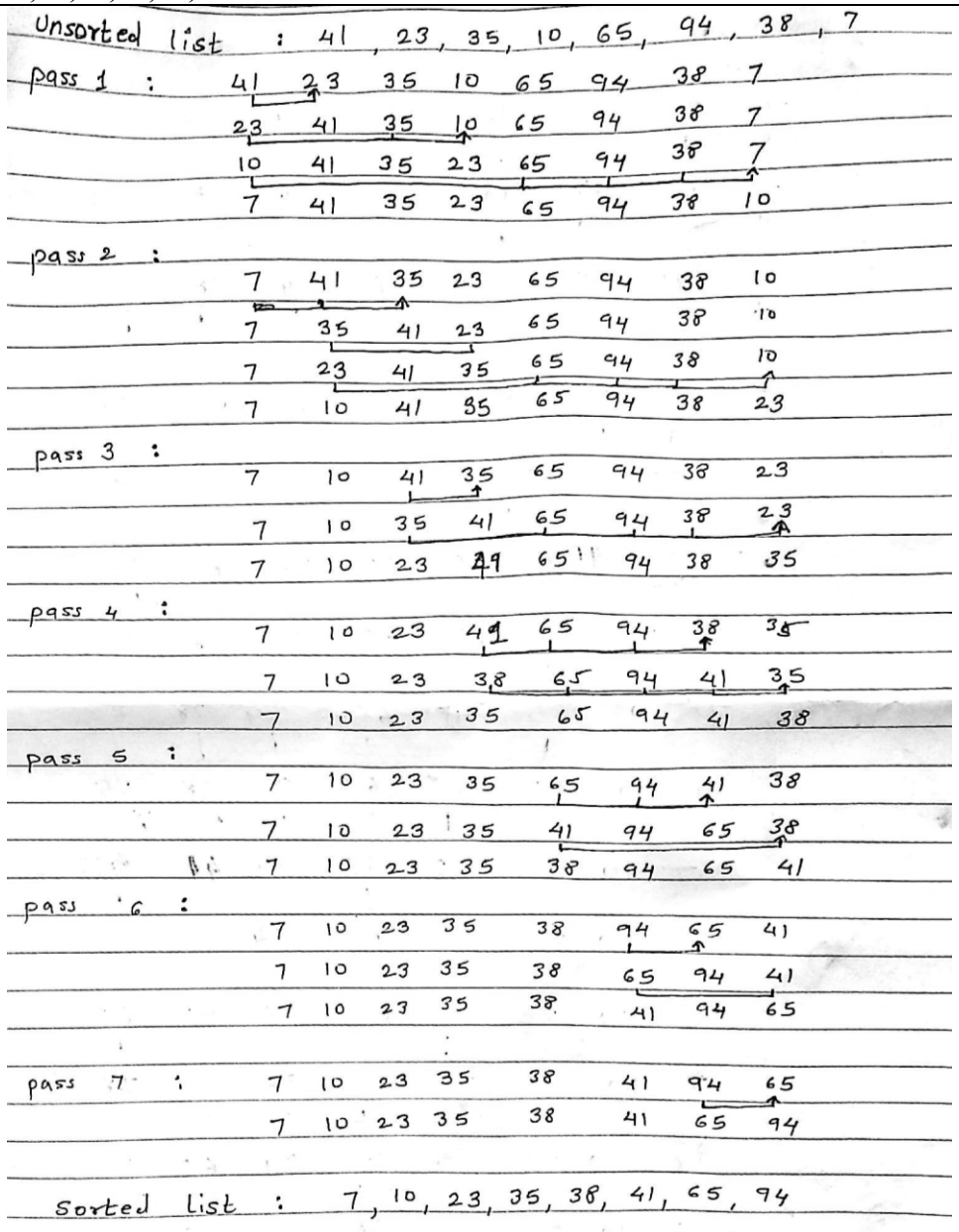
- When a subexpression occurs in a same statement more than once, we can delete all duplicate matrix entries and modify all references to the deleted entry so that they refer to the remaining copy of that subexpression as shown in following figure.
- Compile time computation of operations, both of whose operands are constants
- Movement of computations involving operands out of loops
- Use of the properties of Boolean expressions to minimize their computation
- Machine independent optimization of matrix should occur before we use the matrix as a basis for code generation

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

Machine dependent optimization:

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

Optimized Matrix				First try		Improved code				
				L	1, START	L	1, START			
1	-	STAR T	FINIS H	S	1, FINISH	S	1, FINISH	M 1	→	R 1
				S T	1, M1					
				L	1, RATE	L	3, RATE			
2	*	RATE	M1	M	0, M1	M R	2, 1	M 2	→	R 3
				S T	1, M2					
				L	1, =F'2'	L	5, = F'2'			
3	*	2	RATE	M	0, RATE	M	4, RATE	M 3	→	R 5
				S T	1, M3					
4										
				L	1, M1					
5	-	M1	100	S	1, =F'100'	S	1, =F'100'	M 5	→	R 1
				S T	1, M5					
				L	1,M3	LR	7, 5			
6	*	M3	M5	M	0, M5	M R	6, 1	M 6	→	R 7
				S T	1, M6					
				L	1, M2					
7	+	M2	M6	A	1, M6	A R	3, 7	M 7	→	R 3
					1, M7					
				L	1, M7	ST	3, COST			
8		M7	COST	S	1, COST					

				=			T											
	3)	**Apply interchange sort on following numbers: 41,23,35,10,65,94,38,7.**																**8M**
																		**(Appropriate solution: 8 marks)**
6.		**Answer any four of the following:**																**4x4=16Marks**
	1)	**What are the advantages and disadvantages of combining macro processor with the pass-1 of an assembler?**																**4M**
	Ans:	Advantages of combining macro processor with the pass 1 of assembler. 1. Many functions do not have to be implemented twice (E.g. read a card, test for																**(Advantages : 2 marks, Disadvantag**

		statement type). 2. There is less overhead 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. 3. More flexibility is available to the programmers in that he may use all the feature of the assembler is conjunction with macros. **Disadvantages are:** 1. The combined pass 1 of the assembler and the macro processor may be too large a program to fit into core of some machines. 2. The complexity of such program may be overwhelming. Typically, two separate programming groups implement pass 1 of the assembler and the macro processor. The combination of the two functions may be too much for one group or person to coordinate.	**es:2 marks)**
	2)	**Explain binary search with suitable 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. 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] upto list[5] Pass 1 : Low=0 and 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.	**(Algorithm Explanation: 2 marks, Example:2 marks)**
	3)	**How subroutine linkages are applied in loaders?**	**4M**
	Ans:	It is a mechanism for calling another subroutine in an assembly language. The scenario for subroutine linkage. 1. A main program A wishes to transfer to subprogram B. 2. The programmer in program A, could write a transfer instruction. Eg(BAL,14,B) to subprogram B. 3. But assembler does not know the value of this transfer reference and will declare it is	**(Description: 4 marks, Any relevant description shall be considered)**

		an error. 4. To handle this situation a special mechanism is needed. 5. To handle this mechanism is typically implemented with a relocation or direct linking loader. **Subroutine linkage uses following special pseudo-ops:** **ENTRY, and EXTRN** It is used to direct or to suggest loader that subroutine followed by ENTRY are defined in this program but they are used in another program. **For example**: the following sequence of instruction may be a simple calling sequence to another program. MAIN START ETRNSUBROUT …….. …….. ……. L 15=A(SUBROUT)…..CALLSUBROUTBAIR14, 15 .. .. END The above sequence of instructions first declares SUBROUT as an external variable, which is a variable referenced but not defined in this program. The load (L) instruction loads the address of that variable in to register 15.	
	4)	**Explain storage allocation concept in compiler.**	**4M**
	Ans:	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 w ere reserved by the interpretation phase and did not appearing the source code. 3. Assign storage to literals 4. Ensure th at the storage is allocated and appropriate locations are initialized (Literals and any variables with the initial attribute)The storage allocation phase first scans th rough the identifier table, assigning locations to 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 four Steps: a. Updates the location counter with any necessary boundary alignment. b. Assigns the current value of the location counter to the address field of the variable. c. Calculate the length of the storage needed by the variable (by examining its attributes). d. Update s the location count r by adding this length to it. Once it has assigned relative address to all identifiers requiring ST ATIC storage locations, this phase	**(Description: 4 Marks, Any relevant description shall be considered)**

		creates a matrix entry: 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: This tells code generation to put into the proper storage location the initial values that the action routines saved in the identifier table. A similar scan of the identifier table is made for automatic storage and controlled storage. The scan enters location for each entry. An "automatic" entry and a "controlled "entries are also made in the matrix. Code generation use the relative location entry to generate the ad dress part of instructions. No storage is generate d at compile time for automatic or controlled. However, the matrix entry automatic does cause code to be generate that allocates this storage at execution time, i.e., when the generated code is executed, it allocates automatic storage. The literal table is similarly s canned and locations are assigned to each literal, and a matrix entry is made. Code generation generates storage for a l literals in the static area and initializes the storage i^{th} the values of the literals. Temporary storage is handled differently since each source statement ay reuse the temporary storage (intermediate matrix result area) of the previous source statement. A computation is made of the temporary storage that is required for each source statement. The statement required the greatest amount of temporary storage determines the amount that will be required for the entire program. A matrix entry is made of the form this enables the code generation phase to gene ate code to create the proper amount of storage. Temporary storage is automatic since it is only referenced by the source program and only needed while the source program is active.	
	5)	**Explain the terms:** **a) Binder and** **b) Module loader.**	
	Ans:	**Binder:** A binder is a program that performs the same function as the direct linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck. This output file is in a format ready to be loaded and is typically called a load module. There are two major classes of a binders. The simplest type produces a load module that looks very much like a single absolute loader deck. And another one which is more sophisticated binder known as linkage editor. **Module Loader: -** The module loader merely has to physically load the module into core. The binder essentially performs the functions of allocations, relocation and linking; the module loader performs the function of loading.	**(Binder: 2 marks, Module Loader: 2 marks)**