MAHARASHTRA STATE BOARD OF TECHNICAL EDUCATION
(Autonomous)
(ISO/IEC - 27001 - 2005 Certified)
WINTER – 15 EXAMINATION
Model Answer

Subject Code: 17517 **Subject Name: SYSTEM PROGRAMMING**

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 judgment 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.

Marks

1. a) Attempt any three: **(4×3=12)**

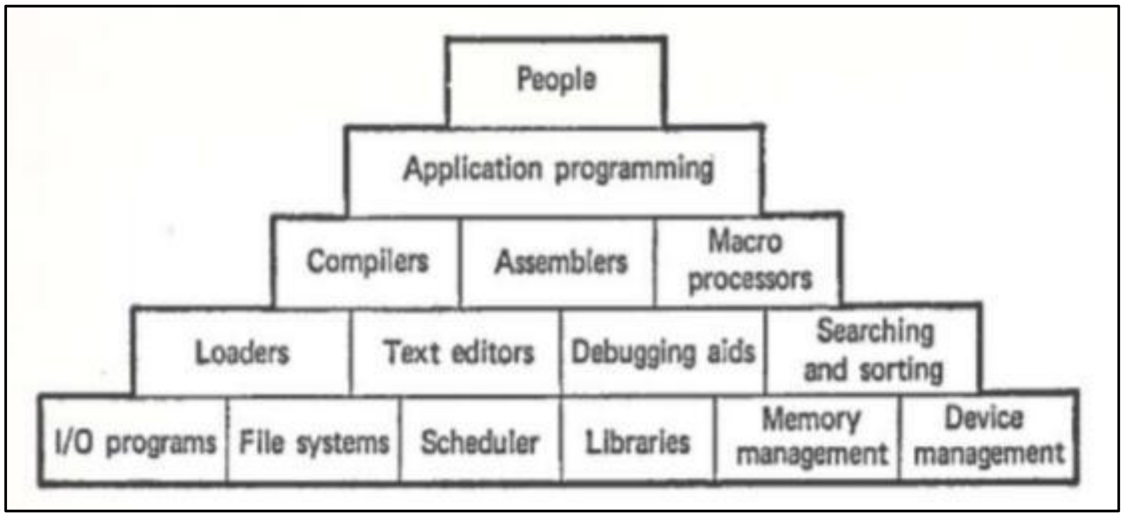
1) State functions of relocating loader.
(Each Function – 1 Mark)

Ans:

i) Provides multiple procedure segments, but only one data segment.
ii) Provides flexible intersegment referencing ability but does not facilitate access to the data segments that can be shared.
iii) The transfer vector linkage is only useful for transfers, and is not well suited for loading or storing external data.
iv) The transfer vector increases the size of the object program in memory

2) **Draw the foundation of system programming.**
(Correct Diagram – 4 Marks)

Ans:



3) **Explain general design of Assembler.**
(all design step - 4 Marks)

Ans:

Step 1: Specify the problem
This includes translating assembly language program into machine language program using two passes of assembler. Purpose of two passes of assembler are to determine length of instruction, keep track of location counter, remember values of symbol, process some pseudo ops, lookup values of symbols, generate instructions and data, etc.
Step 2: Specify data structures
This includes establishing required databases such as Location counter(LC), machine operation table (MOT), pseudo operation table (POT), symbol table(ST), Literal Table(LT), Base Table (BT), etc.
Step 3: Define format of data structures
This includes specifying the format and content of each of the data bases – a task that must be undertaken before describing the specific algorithm underlying the assembler design.
Step 4: Specify algorithm
Specify algorithms to define symbols and generate code
Step 5: Look for modularity
This includes review design, looking for functions that can be isolated. Such functions fall into two categories: 1) multi-use 2) unique
Step 6: Repeat 1 to 5 on modules

4) Explain the lexical phase of compiler.
(Task - 1 Mark; Database - 2 Marks; Algorithm - 1 Mark)

Ans:

Lexical Phase

TASKS

The three tasks of the lexical analysis phase are:

1.To phase the source program into the basic elements or tokens of the language
2.To build a literal table and an identifier table
3.To build a uniform symbol table

DATABASES

These tasks involve manipulations of five databases. Possible forms for these are:

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 (e.g. arithmetic operators , keywords, non-alphanumeric symbols).Each entry consists of the terminal symbol, an indication of its classification (operator ,break character),and its precedence (used in later phases)

		≠
Symbol	Indicator	Precedence

3. ***Literal table***- created by lexical analysis to describe all literals used in the source program. There Is one entry for each literal ,consisting of a value ,a number of attributes, an address denoting the location of the literal at execution time (filled in by a later phase), and other information (e.g., in some implementation we may wish to distinguish between literals used by the program and those used by the compiler ,such as the literal 31 in the expression BINARY FIXED (31)).The attribute ,such as data type or precision ,can be deduced from the literal itself and filled in by lexical analysis.

Literal	Bas	Scal	Precision	Other information	Address

Literal table entry

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. The pointer points to the name in a table of names .Later phases will fill in the data attributes and addresses of each identifier.

Name	Data attributes	Address

5. ***Uniform Symbol table*** – created by lexical analysis to represent the program as a string of tokens rather than of individuals characters. (Space and comments in the source are not represented by uniform symbol and are not used by future phase. There is one uniform symbol for every token in the program.)Each uniform symbol contains the identification of the table of which the token is a member (e.g., a pointer to the table or a code) and its index within that table.

Table	Index

ALGORITHM

The first task of the lexical analysis algorithm is to parse the input character string into tokens.
The second is to make the appropriate entries in the tables .A token is substring of the input string that represent a basic element of the language .It may contain only simple characters and may not include another token. To the rest of 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.

b) Attempt any one: (6×1=6)

1) Explain components of system software with examples.
(Description - 1 Mark each; Example - ½ Mark)

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 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) State use of macro with suitable example.
(Description - 4 Marks; Example - 2 Marks)

Ans:

Macro is used to give single line abbreviation to group of lines which are repeatedly used in program. These statements are combined and kept in macro. Whenever such single line abbreviation is encountered macro processor expands/ replaces this abbreviation with associated group of lines.

```
              .
              .
              .
A        1,DATA    Add contents of  DATA  to  register 1
A        2,DATA    Add contents of DATA to register 2
A        3 ,DATA   Add contents of DATA to register 3

              .
              .
              .
A        1,DATA    Add contents of DATA to register 1
A        2,DATA    Add contents of DATA to register 2
A        3 ,DATA   Add contents of DATA to register 3
```

```
                    .
                    .
                    .
DATA     DC           F'5'
                    .
                    .
                    .
```

Macro processor performs following task.

Recognizing macro definitions: A macro pre-processor must recognize macro definitions that are identified by the MACRO and MEND pseudo-ops. The macro definitions can be easily recognized, but this task is complicated in cases where themacro definitions appear within macros. In such situations, the macro pre-processor must recognize the nesting and correctly matches the last MEND with the first MACRO.

Macro Definition Table

		80 Bytes per entry	
Index		Card	
.		.	
.		.	
.		.	
15	&LAB	INCR	&ARG1, &ARG2, &ARG3
16	#0	A	1,#1
17		A	2,#2
18		A	3,#3
19		MEND	
.		.	
.		.	
.		.	

Saving the definitions: The pre-processor must save the macro instructions definitions that can be later required for expanding macro calls.

	8 Bytes	4 Bytes
Index	Name	MDT Index
.	.	.
.	.	.

.	.	.
3	"INCRbbbb"	15
.	.	.
.	.	.
.	.	.

Recognizing macro calls: The pre-processor must recognize macro calls along with the macro definitions. The macro calls appear as operation mnemonics in a program.

ARGUMENT LIST ARRAY

	8 bytes per entry
Index	Argument
0	"LOOP1bbb"
1	"DATA1bbb"
2	"DATA2bbb"
3	"DATA3bbb"

Replacing macro definitions with macro calls: The pre-processor needs to expand macro calls and substitute arguments when any macro call is encountered. The preprocessor must substitute macro definition arguments within a macro call.

2. Attempt any two : (8×2=16)

1) Write an algorithm for assembler first pass. Explain it in detail.
(Algorithm - 4 Marks; Description - 4 Marks)

Ans:

Algorithm for Pass 1 assembler:

```
STEP 1: BEGIN
STEP 2: LC = 0
STEP 3: READ CARD
STEP 4: SEARCH IN PSEUDO-OP TABLE
        STEP 4. 1 IF FOUND THEN
                IF CARD = DS/DC THEN
                        ADJUST LC TO PROPER ALIGNMENT
                        L = LENGTH OF DATA FIELD
                        GO TO STEP 4.2.4
                ELSE IF CARD = EQU THEN
                        EVALUATE OPERAND
```

```
                    ASSIGN VALUE IN SYMBOL TABLE
                    GO TO STEP 5
ELSE IF CARD = USING/DROP
GOTO STEP 5
                ELSE IF CARD = END THEN
                    ASSIGN STORAGE LOCATIONS TO LITERAL
                    REWIND COPY FOR PASS 2
        STEP 4. 2 ELSE
                STEP 4.2.1     SEARCH IN MACHINE-OP TABLE
STEP 4.2.2      L=LENGTH
                STEP 4.2.3     PROCESS FOR LITERALS
                STEP 4.2.4     IF SYMBOLS IN LABEL FIELD THEN
                                   ASSIGN CURRENT VALUE OF LC TO SYMBOL
                                   LC = LC +1
                               ELSE
LC = LC+1
STEP 5: WRITE COPY OF CARD ON FILE USE BY PASS 2.
STEP 6: GO TO STEP 3
```

PASS 1: DEFINE SYMBOLS

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

The simplest procedure occurs for USING and DROP Pass 1 is only concern with pseudo-ops that defines symbols (Labels) or affect the location counter; USING and DROP do neither assembler need only save the USING and DROP card for Pass 2.

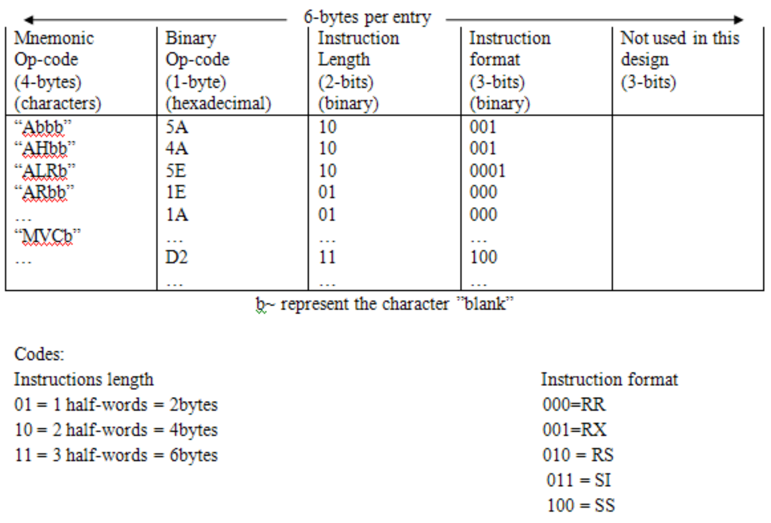
In case of EQU pseudo-op during Pass 1 We can concern only with defining the symbol in the label field this require evaluating the expression in the operand field (The symbol in the operand field and EQU statement must have been defined previously).
The DS and DC pseudo-ops can affects both the location counter and definition of symbols in **Pass 1.** The operand field must be examine to determine the number of bytes of storage require due to requirement for certain alignment conditions. It may be necessary to adjust the location counter before defining the symbol.
When the END Pseudo –op is encountered Pass 1 is terminated before transferring to control to **Pass 2.** There are various "housekeeping" operation that must be performed this including assigning location the literal that have been collected during Pass 1, a procedure that is very similar that for the DC pseudo-op, finally conditions are reinitialized for processing by Pass 2.

2) Explain the database used by pass I and Pass II of an assembler.
(Pass 1 Database - 4 Marks; Pass 2 Database - 4 Marks)

Ans:

Pass 1 data bases

- Input source program
- A LC to keep track of each instruction location
- A MOT (Machine Operation Table)



6-bytes per entry

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

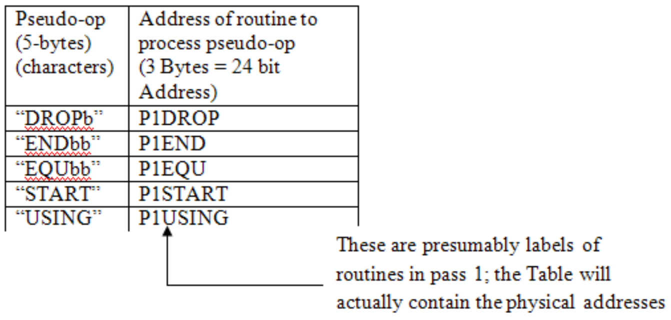
b~ represent the character "blank"

Codes:

Instructions length	Instruction format
01 = 1 half-words = 2bytes	000=RR
10 = 2 half-words = 4bytes	001=RX
11 = 3 half-words = 6bytes	010 = RS
	011 = SI
	100 = SS

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

- A POT (Pseudo operation Table)

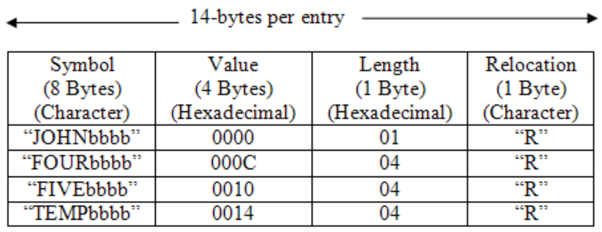


Pseudo-op (5-bytes) (characters)	Address of routine to process pseudo-op (3 Bytes = 24 bit Address)
"DROPb"	P1DROP
"ENDbb"	P1END
"EQUbb"	P1EQU
"START"	P1START
"USING"	P1USING

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

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

- A ST (Symbol Table)



14-bytes per entry

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

Symbol Table:

- A LT (Literal Table)

Variable tables

Symbol Table

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

Literal Table

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

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

Pass 2 databases

- Copy of source program input to pass 1
- LC: Same as Pass I
- MOT: Same as Pass I
- POT: Same as Pass I
- ST: Same as Pass I
- BT (Base table)

	Availability Indicator (1-byte) (character)	Designated relative – address contents of base register (3-bytes = 24-bit address) (hexadecimal)	
1	"N"	-	15 entries
2	"N"	-	
.	:		
14	"N"	-	
15	"Y"	00 00 00	

Code=
Availability
Y~ register specified in USING pseudo-op
N~register never specified in USING pseudo-op or subsequently made unavailable by the DROP pseudo-op

- Output in machine code to be needed by the loader

3) Explain simple machine independent optimization algorithm with an example.
(Description of Simple Machine Independent - 4 Marks; Example - 4 Marks; any relevant example shall be considered)

Ans:

Machine- independent optimization:

- When a sub-expression 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 sub-expression as shown in following table.
- 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

Example:

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

Matrix with common sub-expressions

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

Matrix after elimination of common sub-expressions

3. Attempt any four: (4×4=16)

1) List various applications of system software

(Any four applications – 4 Marks)

Ans:

1. It increases the productivity of computer which depends upon the effectiveness, efficiency and sophistication of the systems programs.
2. Compilers are system programs that accept people like languages and translate them into machine language.
3. Loaders are system programs that prepare machine language program for execution.
4. Macro processors allow programmers to use abbreviation.
5. Provides efficient management of various resources.
6. It manages multiprocessing, paging, segmentation, resource allocation.
7. Operating system and file systems allow flexible storing and retrieval of information.

2) Explain hash and random entry searching.

(Description of hash and random entry searching - 4 Marks)

Ans:

All Binary Search algorithms, which are fast, but can only operate on tables that are ordered and packed, i.e. tables that will have adjacent items ordered by keywords. Such search procedures may therefore have to be used in conjunction with a sort algorithm which both orders and packs the data.

Actually, it is unnecessary for the table to be ordered and packed to achieve good speed in searching. This is also possible to do considerably better with an unpacked, unordered table, provided it is sparse, i.e. the number of storage spaces allocated to it exceeds the number of items to be stored.

It is observed that the address calculation sort gives good results with a sparse table. However, having to put elements in order slows down the process. A considerable improvement can be achieved by inserting element in a random (or pseudo-random) way.

The random entry number K is generated from the key by methods similar to those used in address calculation. If the K the some other cell must be found for the insertion. The first problem is the generation of a random number from the key. It is to design a procedure that will generate pseudo-random, consistent table positions for keywords.

One fairly good prospect for four character EBCDIC keywords is to simply divide the keyword by the table length N and use the remainder. This scheme works well as long as N and the key size (32 bits in case) have no common factors. For a given group of M keywords the remainders should be fairly evenly distributed over)....(N-1).

3) Outline the algorithm for syntax analysis phase of compiler.
(4 steps - 1 Mark each)

Ans:

The algorithm for syntax analysis phase is as follows:

1. Reductions are tested consecutively for match between Old Top of Stack field and the actual Top of Stack, until match is found.
2. If match is found, the action routines specified in the action field are executed in order from left right.
3. When control return to the syntax analyzer, it modifies the Top of Stack to agree with the New Top of Stack field.
4. Step 1 is the repeated starting with the reduction specified in the next reduction field

4) Explain dynamic binder loading scheme.
(Description of dynamic binder loading scheme - 4 Marks)

Ans:

In dynamic linking, 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 linking 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

5) Explain the meaning of top down and bottom up parser.
(Top down parser – 2 Marks and bottom up parser – 2 Marks)

Ans:

Top-down Parser

The top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. It can be done using recursive decent or LL(1) parsing method. It cannot handle left recursion. It is only applicable to small class of grammar.

Bottom-up Parser

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. It starts from a sentence and then apply production rules in reverse manner in order to reach the start symbol. It is a table driven method and can be done using shift reduce, SLR, LR or LALR parsing method. It handled the left recursive grammar.
It is applicable to large class of grammar.

Consider the grammar
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

and the input string w = cad

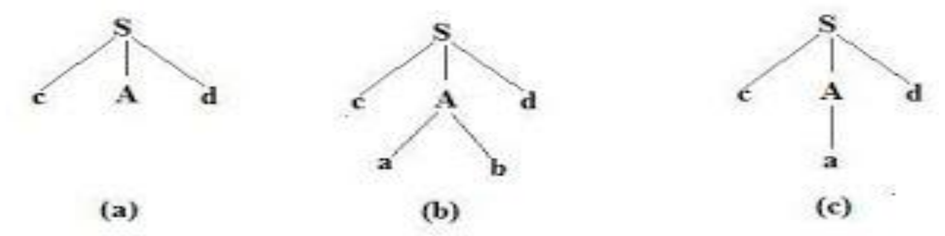


Fig: Top – Down Parsing

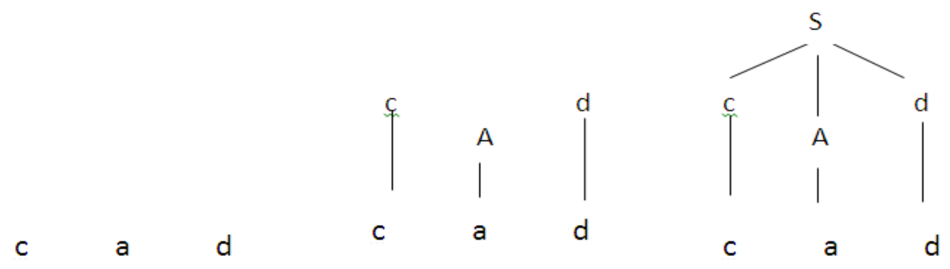


Fig:-Steps in bottom up parse

4. A) Attempt any three: **(4×3=12)**

1) How sub-routine linkage are applied in loaders?
(Description - 4 Marks)

Ans:

A main program A wishes to transfer to subprogram B. The programmer, in program A, could write a transfer instruction (e.g BAL 14 B) to subprogram B. However, the assembler does not know the value of this symbol reference and will declare it as an error (undefined symbol)Unless a special mechanism has been provided. This mechanism is typically implemented with a relocating or a direct linking The assembler pseudo-op EXTERN followed by a list of symbol indicates that these symbols are defined in other programs but referenced in the present program .Correspondingly, if a symbol is defined in one program and referenced in others, we insert it into symbol list following the pseudo-op ENTRY .In turn the assembler will inform the loader that these symbols may be referenced by other programs. For examples, the following sequence of instructions may be a simple calling sequence to another program:

```
MAIN        START
            ETRN SUBROUT
            ……………..
            ……………..
            …………….
            L 15=A(SUBROUT)…..CALL SUBROUT

            BAIR 14, 15
             ..
             ..
             ..
             ..
            END
```

The above sequence of instructions first declares SUBROUT as an external variable ,that is variable referenced but not defined in this program .The load instruction loads the address of that variable into 15. The BALR instruction branches to the contains of register 15, which is the address of SUBROUT ,and leaves the value of the next instruction in register 14.

2) Apply the optimization techniques for suitable example.
(2 Marks for each; Any 2 Techniques; description is optional)

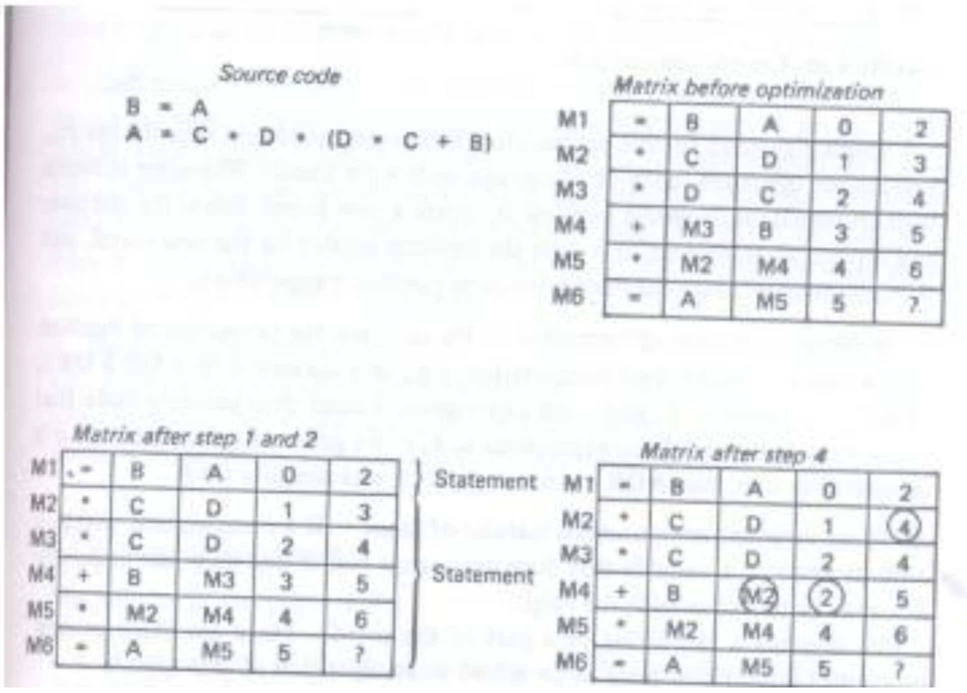
Ans:

1. Elimination of common sub expression:

The elimination of duplicate matrix entries can result in a more can use and efficient object program. The common subexpression must be identical and must be in the same statement.

The elimination algorithm is as follows:-

i) Place the matrix in a form so that common subexpression can be recognized.
ii) Recognize two subexpressins as being equivalent.
iii) Eliminate one of them.
iv) After the rest of the matrix to reflect the elimination of this entry.



Source code

```
B = A
A = C * D * (D * C + B)
```

Matrix before optimization

M1	=	B	A	0	2
M2	*	C	D	1	3
M3	*	D	C	2	4
M4	+	M3	B	3	5
M5	*	M2	M4	4	6
M6	=	A	M5	5	?

Matrix after step 1 and 2

M1	=	B	A	0	2	Statement
M2	*	C	D	1	3	Statement
M3	*	C	D	2	4	
M4	+	B	M3	3	5	
M5	*	M2	M4	4	6	
M6	=	A	M5	5	?	

Matrix after step 4

M1	=	B	A	0	2
M2	*	C	D	1	4
M3	*	C	D	2	4
M4	+	B	M2	2	5
M5	*	M2	M4	4	6
M6	=	A	M5	5	?

2. Compile time compute:-

Doing computation involving constants at compile time save both space and execution time for the object program.

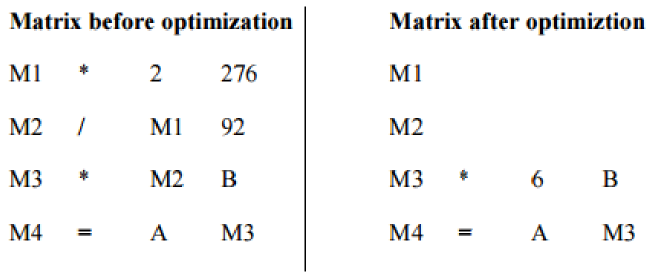
The algorithm for this optimization is as follows:-

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				Matrix after optimiztion			
M1	*	2	276	M1			
M2	/	M1	92	M2			
M3	*	M2	B	M3	*	6	B
M4	=	A	M3	M4	=	A	M3

3.Boolean expression optimization:- We may use the properties of boolean expression to shortentheir 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.

Original Code is:-

```
For y=0 to height-1
For x=0 to width-1

i=y*width+x
Process I
Next x
Next y
```

here y*width is loop invarient not change in inner loop

Modified code is:-

```
For y=0 to height-1
```

```
temp= y*width
For x=0 to width-1
i=temp+x
Process i
Next x
Next y
```

3) Explain four purposes of storage assignment phase of compiler.
(Each purpose - 1Mark)

Ans:

The purpose of this phase 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 result, e.g the results of matrix lines. These storage references were reserved by the interpretation phase and did not appear in the source code.
3. Assign storage to literals
4. Ensure that the storage is allocated and appropriate locations are initialized (Literals and any variables with the initial attribute)
   The storage allocation phase first scans through 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:

1. Updates the location counter with any necessary boundary alignment.
2. Assigns the current value of the location counter to the address field of the variable.
3. Calculate the length of the storage needed by the variable (by examining its attributes). 4. Updates the location counter by adding this length to it. Once it has assigned relative address to all identifiers requiring STATIC storage locations, this phase creates a matrix entry:

STATIC	Size	

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:

Initialize	Variable	

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 relative location for each entry. An “automatic” entry and a “controlled “entry are also made in the matrix. Code generation use the relative location entry to generate the address part of instructions. No storage is generated at compile time for automatic or controlled. However, the matrix entry automatic does cause code to be generated that allocates this storage at execution time, i.e., when the generated code is executed, it allocates automatic storage.

LIT	Size	

The literal table is similarly scanned and location are assigned to each literal, and a matrix entry is made. Code generation generates storage for all literals in the static area and initializes the storage with the values of the literals. Temporary storage is handled differently since each source statement may 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 generate code to create the proper amount of storage.

AUTOMATIC	Size	

Temporary storage is automatic since it is only referenced by the source program and only needed while the source program is active.

4) **Explain the concept of top down parser.**
(Description - 2 Marks; 2 Marks for Description of any two type of top down parser)

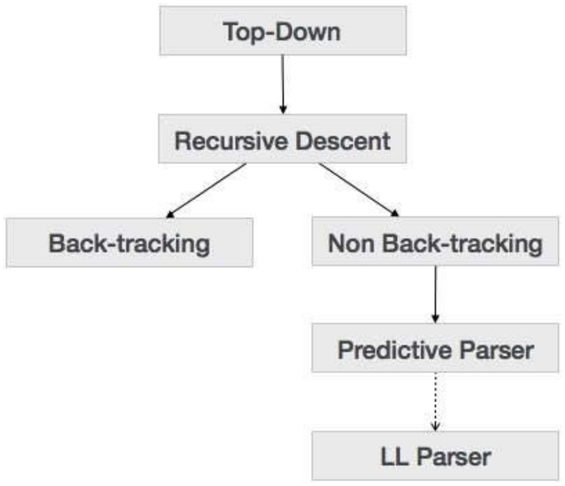
Ans:

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:



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:

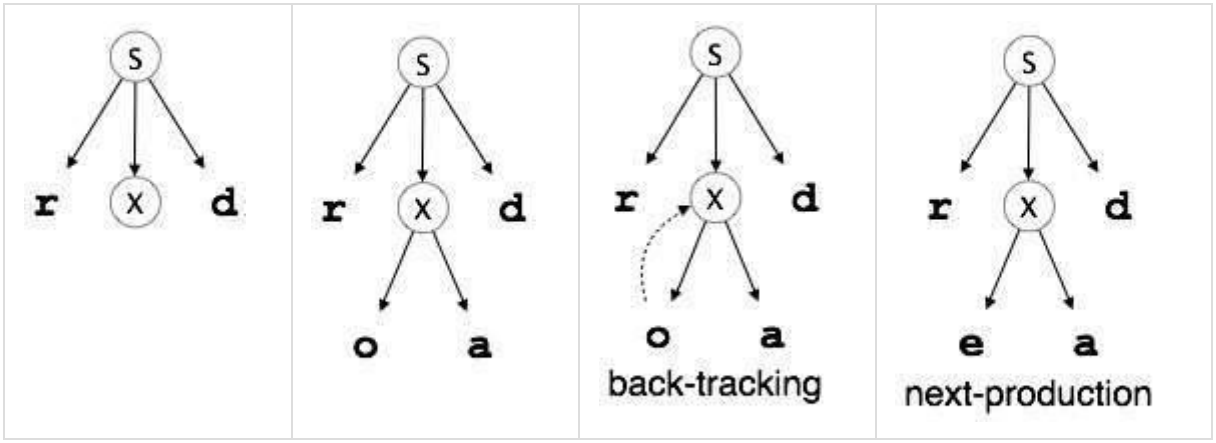
S →rXd|rZd

X →oa|ea

Z →ai

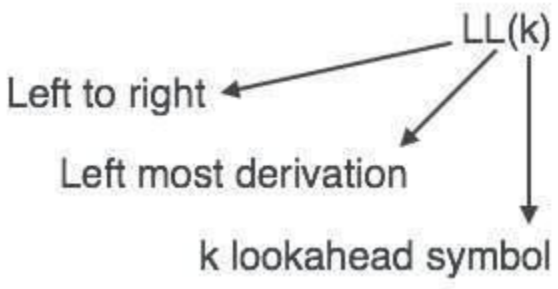
For an input string: read, a top-down parser, will behave like this:
It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘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 aheads. Generally k = 1, so LL(k) may also be written as LL(1).



B) Attempt any one: (6×1=6)

1. State and explain four basic task of macro processor.
(List - 2 Marks; Description of each functions -1 Mark each)

Ans:

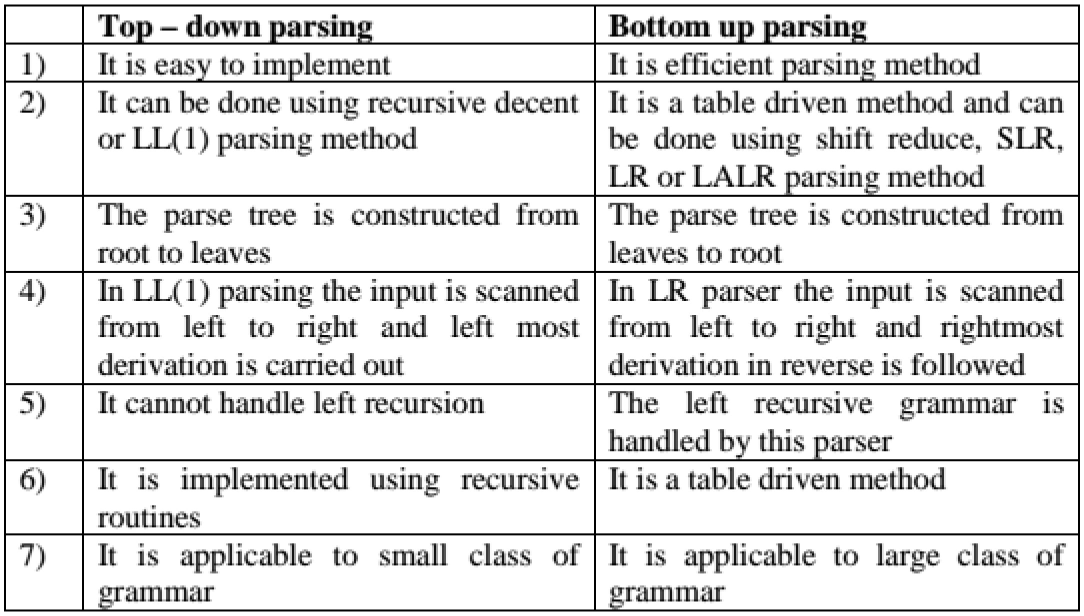
The 4 basic task of Macro processor is as follows:-

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 microprocessor must recognize macro definitions Identified by the MACRO and MEND pseudo-ops. When MACROS and MENDS are nested, the macroprocessor 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 appear 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.

2. **Compare advantages and disadvantages at top down and bottom up parser.**
(Any six points - 1 Mark each)

Ans:



	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

OR

Top down parser
Advantages:-
1. It is easy to implement
2. It never wastes time on subtrees that cannot have an S at the root. Bottom up parsing does this.

Disadvantages:-
1. It is not efficient parsing method as compare to bottom up parser
2. It cannot handle left recursion.
3. It is not applicable to large scale of grammar.

4. Wastes time on trees that don’t match the input (compare the first word of the input with the leftmost branch of the tree). Bottom-up parsing doesn’t do this.

Bottom up parser

Advantages:-

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

Disadvantages:-

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

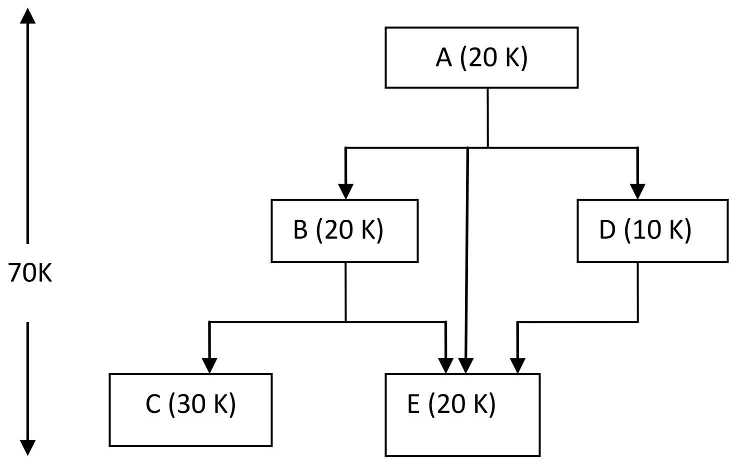
5. Attempt any two: (8×2=16)

1) Explain overlay structure in detail

(Explanation - 5 Marks, diagram/example - 3 Mark; any relevant diagram/example shall be considered)

Ans:

Usually the subroutines of a program are needed at different times for example pass1and pass2 Of an assembler are mutually exclusive .by explicitly recognizing which subroutine call other subroutine it is possible to produce an overlay structure that identifies mutually exclusive subroutine . A program consisting of five subprogram (A, B, C, D, AND E) that require 100k bytes of core. The arrows indicate that subprogram A only calls B, D and E subprogram B only calls C and E subprogram only calls E and subprogram C and E do not call any other routines. highlight the interdependencies between the procedures .note that procedures B and D are never in use at the same time neither are C and E .if we load only those procedure that are actually to be used at any particular time the amount of core needed is equal to longest path of overlay structure .this happens to be 70K for the example in procedure A,B and C. a storage assignment for each procedure consistent with the overlay structure.



In order for the overlay structure to work it is necessary for the module loader to load the various procedure as they are needed .we will not go into their to load the various procedure as they are needed .we will not g into their specific details but there are many binders capable of processing and allocating an overlay structure .the portion of the loader that actually intercepts the calls and load the necessary procedure is called the overlay supervisor or simply the flipper .this overall scheme is called dynamic loading or load –on-call (LOCAL).

2) Write the algorithm for elimination and common sub-expression. Apply it for the following Statements and show the designed matrix :

B=A

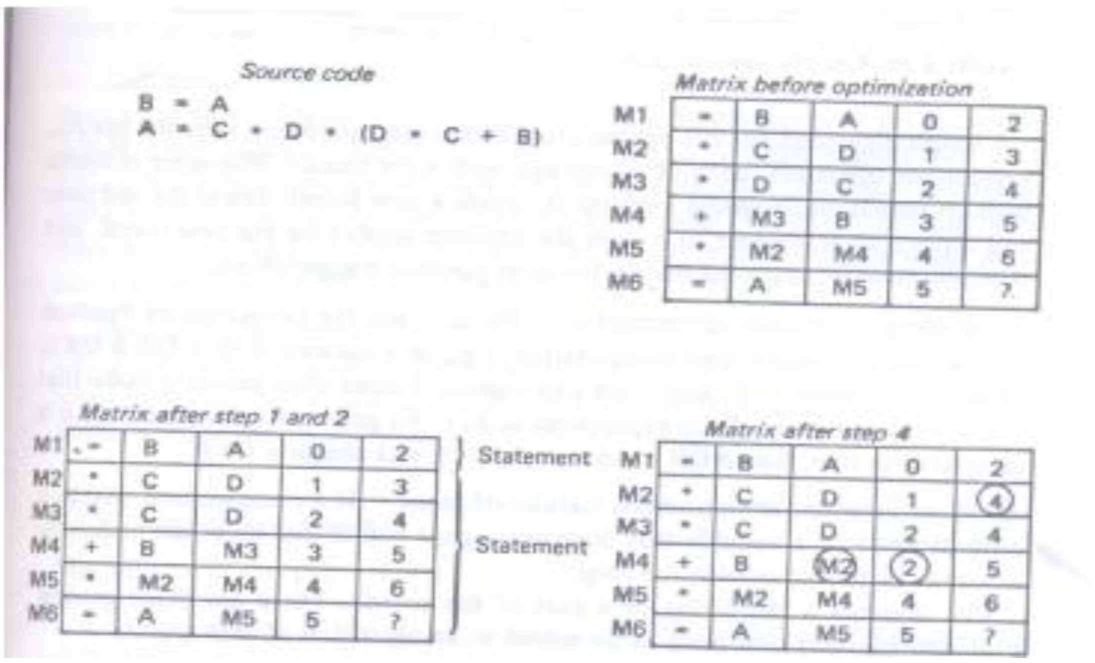
A=C*D*(D*C+B)

(Algorithm - 4 Marks; Optimization - 4 Marks)

Ans:

The elimination algorithm is as follows:-

i) Place the matrix in a form so that common sub expression can be recognized.
ii) Recognize two sub expressions as being equivalent.
iii) Eliminate one of them.
iv) After the rest of the matrix to reflect the elimination of this entry.



Source code

```
B = A
A = C * D * (D * C + B)
```

Matrix before optimization

M1	=	B	A	0	2
M2	*	C	D	1	3
M3	*	D	C	2	4
M4	+	M3	B	3	5
M5	*	M2	M4	4	6
M6	=	A	M5	5	7

Matrix after step 1 and 2

M1	., =	B	A	0	2	Statement
M2	*	C	D	1	3	
M3	*	C	D	2	4	Statement
M4	+	B	M3	3	5	
M5	*	M2	M4	4	6	
M6	=	A	M5	5	?	

Matrix after step 4

M1	=	B	A	0	2
M2	*	C	D	1	4
M3	*	C	D	2	4
M4	+	B	M2	2	5
M5	*	M2	M4	4	6
M6	=	A	M5	5	?

3) Show the result of each pass by using Radix Sort :
424,887,807,709,882,616,573,413,679,180,975,264.
(Pass 1 - 2 Marks; Pass 2 - 2 Marks; Pass 3 with Output - 4 Marks)

Ans:

Pass-1

0	1	2	3	4	5	6	7	8	9
180		882	413 573	264 424	975	616	807 887		679 709

Pass-2

0	1	2	3	4	5	6	7	8	9
709 807	413 616	424			573	264	679 975	180 882 887	

Pass-3

0	1	2	3	4	5	6	7	8	9
	180	264		413 424	573	616 679	709	807 882 887	975

Sorted elements are

180, 264, 413, 424, 573, 616, 679, 709, 807, 882, 887, 975.

6. Attempt any four of the following: (4×4=16)

1) Describe conditional macro expansion with suitable example.
(explanation -2 Marks , example - 2 Marks)

Ans:

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 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 macro processor to skip to the statement. Labeled .FINI if the parameter corresponding to & COUNT is a1; otherwise the macro processor is to continue with the statement following the AIF pseudo-ops. AIF is conditional branch pseudo ops it performs an arithmetic test and branches only if the tested condition is true. AGO is an unconditional branch pseudo-ops or 'Go to' statement. It specifies label appearing on some other statement. AIF & AGO controls the sequence in which the macro processor expands the statements in macro instructions.

2) How to improve the assembler design?
(Description - 4 Marks)

Ans:

1. The rules of assembly language states that the symbol should be defined same where in the program. Hence there may be same cases in which the reference is made to the symbol prior to its definition end such reference is called forward reference.
2. Due to forward reference, assembler cannot assemble the instructions and such a problem is called forward reference problem.
3. To solve the problem, assembler will make two phases (scan) once the I/P program.
4. The purpose of pass 1 is to define the symbols and the literals encounter in the program. The purpose of pass 2 is to assemble the instruction and assemble the data.

3) What is the algorithm for direct linking loader?
(Algorithm - 4 Mark; any 1 algorithm shall be considered)

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
```

```
        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.
```

4) Explain how to reduce different process in compiler?
(Description – 4 Marks)

Ans:

The syntax rules of the source language are contained in the reduction table. The syntax analysis phase is an interpreter driven by the reductions.

The general form of rule or reduction is:

Label: old top of stack/ Action routines/ new top of stack/ Next reduction.

Example:

```
/ /***/
<idn> PROCEDURE/bgn_proc/S1 ****/4
<any><any><any>/ERROR/S2S1*/2
```

These three reductions will be the first three of the set defined for the example. The interpretation is as follows:

1. Start by putting the first three uniform symbols from the UST onto the stack.
2. Test to see if top three elements are <idn>: PROCEDURE.
3. If they are, call the begin procedure (bgn_proc) action routine, delete the label and get the next four uniform symbols from the UST onto the stack and go to reduction
4. If not, call action routine ERROR, remove the third uniform symbol from the stack get one more from the UST, and go to reduction 2.

The reduction state that all programs must start with a „<label>: PROCEDURE... The syntax phase deletes the label and the „:., gets four more tokens and interprets reduction 4, which will start parsing of the body of the procedure.

If the first statement is not a <label>: PROCEDURE until a match is found or until all the symbols in the UST have been tried.

5) What is purpose of ID number on ESD cards? Why it is not needed for locally defined symbols?

(Purpose of ID - 2 Marks; reason - 2 Marks)

Ans:

Purpose of ID number on ESD:

Each SD and ER symbol is assigned a unique number by the assembler. This number is called as symbol identifier or ID which is used in conjunction with RLD Card.

Reason behind Not needed for locally defined Symbol:

The external symbol is used for relocation or linking is identified on RLD cards by means of an ID number rather than symbol name. The id number must match an SD or ER entry on ESD card. Since an entry of locally declared symbols are already known hence the unlike the case with GEST it is not necessary to search the LESA given an ID number the corresponding value is written as LESA(ID) can be immediately obtained.