MAHARASHTRA STATE BOARD OF TECHNICAL EDUCATION
(Autonomous)
(ISO/IEC - 27001 - 2005 Certified)

SUMMER-16 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
12

1. a)Attempt any three :

i) Draw and describe" Compile and go" loaders.
(Diagram - 1 mark; Description - 3 marks)

Ans:

One method of performing the loader functions is to have the assembler run in one part of memory and place the assembled machine instructions and data, as they are assembled, directly into their assigned memory locations.

As a usual practice one method of performing the loader functions is to have to assemble run in one part of memory and place the assembled machine instructions and data they are assembled, directly into their assigned memory locations.

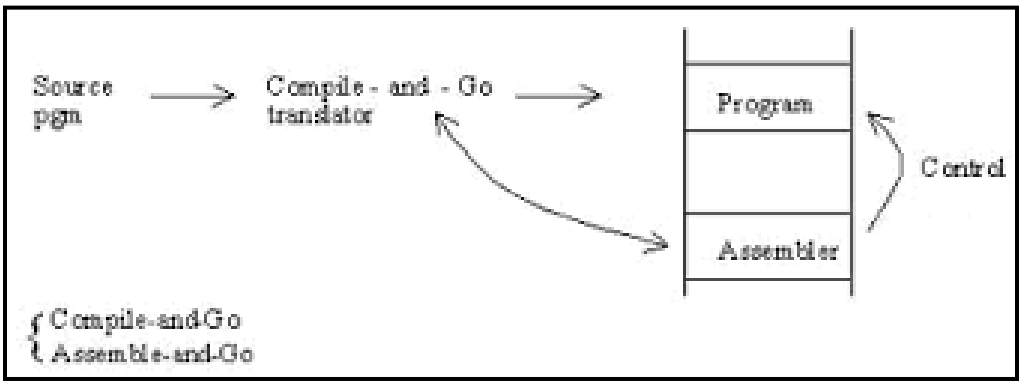
When the assembly is completed the assembler causes transfer to the instruction of the program. This is a simple solution, involving no extra procedures. It is used by the WATFOR FORTRAN compiler and several other language processors.

Such a loading scheme is commonly called "compile-and-go" or "assembler – and –go". It is relatively easy to implement. The assembler simply places the code into core, and the "loader" consists of one instruction that transfers to the starting instruction of the newly assembled program.

Disadvantages:

1. A portion of memory is wasted because the core occupied by the assembler is unavailable to the object program.
2. It is necessary to retranslate (assemble) he user's program code every time it is run.
3. It is very different to handle multiple subroutines in assembly language and another subroutine in any programming language.

This last disadvantage makes it very difficult to produce orderly modular programs in the design of assemblers. For example assembler is one of the type of compile and go loader which can be depicted in the following figure:



ii) What is system software? State the goals of system software.
(System software - 2 marks; Goals - 2 marks)

Ans:

These are the programs that help in the effective execution of the application programs and allow the application programmer to focus on the application to be developed without concerning about the internal detail of the system

e.g.
Assembler
Macro-processor
Loader
Linker
Compiler
Editor
Interpreter
Operating System

Goal of System software

- To achieve efficient use of available resources.
- To achieve efficient performance of the system.
- To make effective execution of general user program.
- To make available new better facilities.
- User convenience - provide convenient methods of using a computer system.
- Non-interference - prevent interference in the activities of its user.

iii) Describe linear search algorithm with suitable example.
(Description - 1 mark; Algorithm - 1 mark; Example - 2 marks)

Ans:

Linear search is an algorithm in which an element is search in given data structure in sequential manner. The searching begins from one end and it will check each element in given data structure for the availability of key element. The algorithms compare each element with key element. If element exist then it displays location of element or else appropriate error message will be displayed. Since each element is checked for availability of desired element this algorithm runs on data structure which is not sorted. It has best case time complexity as O(1), with average and worst case time complexity as O(N).

Advantage:-

Simple to implement.
Works efficient on small data structure.

Disadvantage: -

Slow in execution.
As the data structure increases efficiency of algorithm reduces.

Algorithm:-

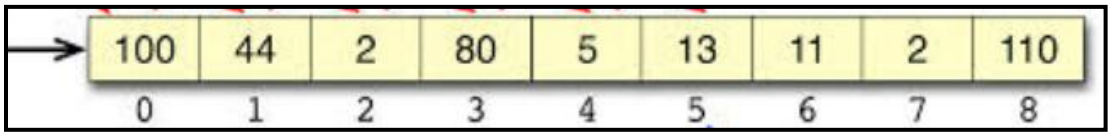
STEP 1 – START
STEP 2 – GENERATE DATA STRUCTURE DS[ ] .
STEP 3 – ACCEPT KEY ELEMENT TO BE SEARCHED.
STEP 4 – SET COUNTER = 0
STEP 5 – WHILE COUNTER < N
    STEP 5.1 – IF (DS[COUNTER] IS EQUAL TO KEY)
        PRINT ELEMENT FOUND AT "COUNTERth" LOCATION
        GOTO TO STEP 6
    STEP 5.2 – ELSE

COUNTER ++;

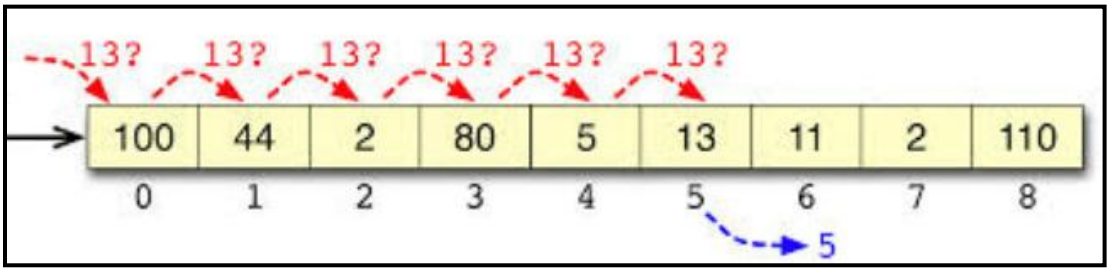
STEP 6 – END

Example:

Consider the following data structure



Now we want to search element 13 in given data structure, the algorithm search sequentially as follows:



iv) Differentiate between machine independent and machine dependant optimization.
(Any 4 points of comparison - 1 mark each)

Ans:

Machine Independent	Machine Dependent Optimization
It reduces only memory.	It reduces both, memory and space.
It is done after generating code	It is done while generating code.
It does not make efficient use of temporary storage.	It makes efficient use of temporary storage.
It does stores intermediate results.	It does not stores intermediate results unless it is essential.
It has higher number of loads and stores operations.	It reduces number of loads and stores operations.
It is comparatively slower to machine dependent.	It is faster as compared to machine independent.

b) Attempt any one: 6

i) Describe the Evolution of system software and operating system.
(Evolution of System software - 3 marks; Evolution of Operating System - 3 marks)

Ans:

Evolution of System Software

1. The earliest computers were entirely programmed in the machine language.
2. Programmers would write out the symbolic program on sheets of paper, hand-assemble into machine code and then toggle the machine code into the computer.
3. Assemblers solved the problem by allowing the programmers to write program in terms of symbolic names and binding the names to machine addresses.
4. The relocating loader allowed the users of the sub-programs to write each sub-program as it starts at location zero.
5. Linkers and loaders divided up the work, with linker doing part of address binding, assigning address with each program and the loader doing a final relocation step to assign.
6. Linkers had to deal with object code generated by high level programming languages such as FORTRAN.
7. Multiple copies of the same program were frequently run at the same time, compilers and the assemblers were modified to create object code in multiple sections with one section for read only code another code for writable data.

Evolution of Operating System:

1. As the demands for computer till memory, devices, and files increased, the efficient management of these resources become more critical. These resources are valuable, and inefficient management of them can be costly. The management of each resource has evolved as the cost and sophistication of its use increased.
2. In simple batched system, the memory resource was allocated totally to a single program. Thus, if a program did not need the entire memory, a portion of that resource was wasted.
3. Multiprogramming operating systems with partitioned core memory were developed to circumvent this problem. Multiprogramming allows multiple program to reside in separate areas of core at the same time. Programs were given a fixed portion of core.
4. Often in such partitioned memory system some portion could not be used since it was too small to contain a program. The problem of “holes” or unused portions of core is called “fragmentation”. Fragmentation has been minimized by the technique of relocatable partitions.
5. Paging is a method of memory allocation by which the program is subdivided into equal portions of pages, and core is subdivided into equal portions or blocks. The pages are loaded into blocks.

6. In simple paging all the pages of a program must be in core for execution. In demand paging a program can be executed without all pages being in core.
7. The traffic controller coordinates the processors and the processes. The resource of processor time is allocated by a program known as the scheduler.
8. The processor concerned with I/O is referred to as the I/O processor, and programming this processor is called I/O programming.

ii) Mention the features of macro facility.
(List of Features - 2 marks; Description - 1 mark each)

Ans:

Following are features of a MACRO facility:

1. Macro instruction Arguments.
2. Conditional Macro Expansion
3. Macro calls within Macros
4. Macro instruction Defining Macros

Macro Instruction Arguments:

The macro facility presented is capable of inserting block of instructions in place of macro calls. All of the calls to any given macro will be replace by identical blocks. This lacks flexibility: there is no way for a specific macro call to modify the coding that replaces it. An important extension of this facility consists of providing for arguments.

An important extension of this facility consists of providing for arguments or parameters in macro calls. Corresponding macro dummy arguments will appear in macro definitions.

```
        .
        .
        .
A       1,DATA1
A       2,DATA1
A       3,DATA1
        .
        .
        .
A       1,DATA2
A       2,DATA2
A       3,DATA2
        .
```

```
                .
                .
DATA 1 DC       F'5'
DATA 2 DC       F'10'
```

In this case the instruction sequences are very similar but not identical. The first sequences performs an operation using DATA1 as operand; the second using DATA2. They can be considered to perform the same operation with a variable parameter, or argument. Such parameter is called a macro instruction argument or dummy arguments. It is specified on the macro name line and distinguished by the ampersand which is always its first character.

```
MACRO                   Macro INCR has One Argument
   INCR     &ARG
   A        1,&ARG
   A        2,&ARG
   A        3,&ARG
  MEND
     .
     .
     .
   INCR     DATA1      Use DATA1 as operand
     .
     .
     .
   INCR     DATA2      Use DATA2 as operand
     .
     .
     .
DATA1 DC    F'5'
DATA2 DC    F'10'
     .
     .
     .
```

Conditional Macro Expansion:

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.

AIF is conditional branch pseudo-o; it performs an arithmetic test and branches only if the tested condition is true. The AGO is an unconditional branch pseudo ops or 'go to' statement. It specifies a label appearing on some other statement in the macro instruction definition; the macro processor continues sequential processing of instruction with the indicated statement. These statements are directives to the macro processor and do not appear in macro expansion.

Macro call within Macros:

Since macro calls are "abbreviations" of instruction sequences, it seems reasonable that such abbreviations should be available within other macro definitions.

Macro calls within macros can involve several levels. In fact, conditional macro facilities make it possible for a macro to call itself. So long as this does not cause an infinite loop so long as at some point the macro having been called for the nth time, decides not to call itself again perfectly.

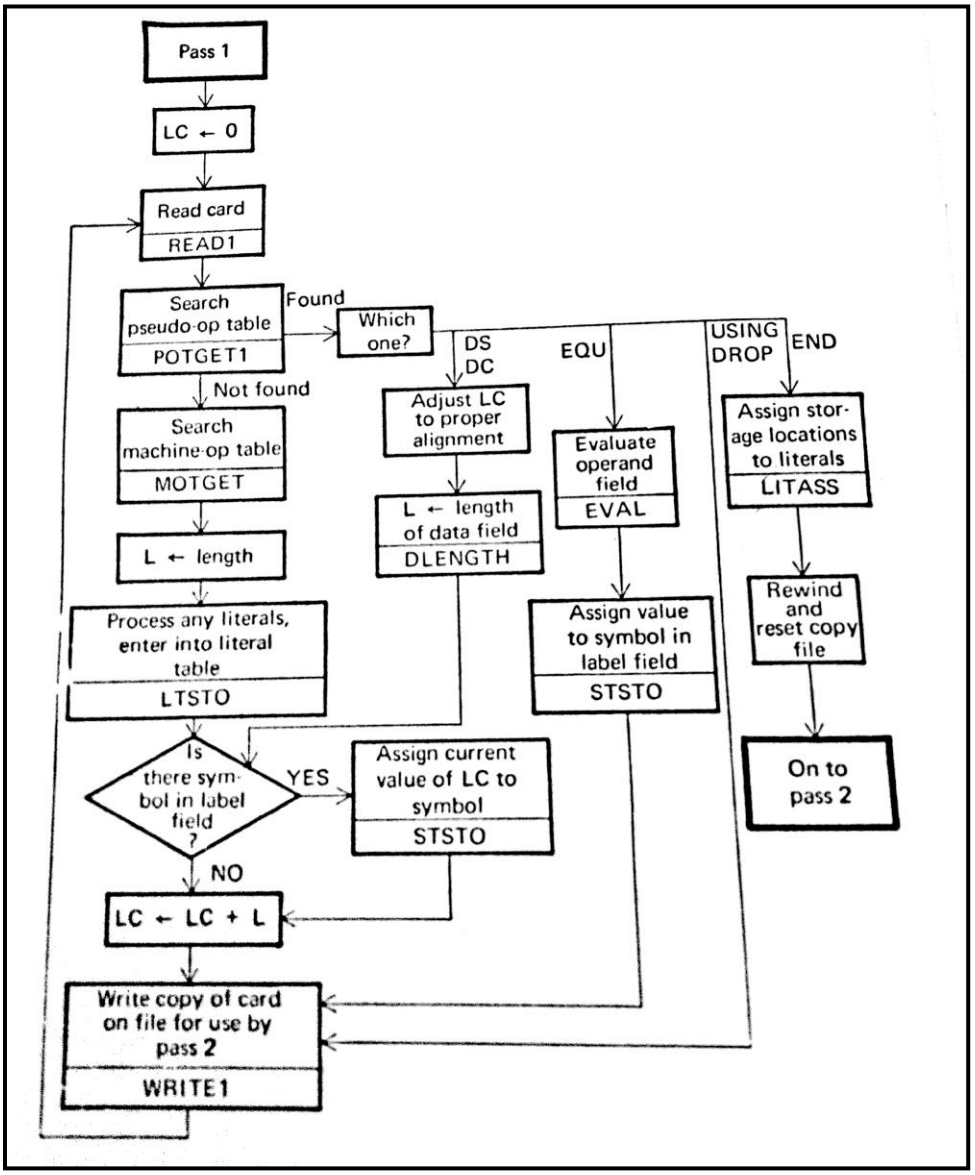
Macro Instructions Defining Macros

Macros are generalized abbreviations for instruction sequences, nothing that it seems reasonable to permit any valid statements in the abbreviated sequence, including macro definitions. In this manner a single macro instruction might be used to simplify the process of defining a group of similar macros.

2. Attempt any two: **16**

i) Explain pass I of assembler.
(Description - 5 marks; flowchart - 3 marks)

Ans:



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.

ii) Sort the following numbers using Radix exchange sort.Show all the passes.

102,501,704,222,112,555

710,113,200,211,222,111

(Pass I - 3 marks; Pass II - 3 marks; Pass III - 2 marks)

Ans:

Pass 1:-

	0	1	2	3	4	5	6	7	8	9
102			102							
501		501								
704					704					
222			222							
112			112							
555						555				
710	710									
113				113						
200	200									
211		211								
222			222							
111		111								

After Pass I:-
710, 200, 501, 211, 111, 102, 222, 112, 222, 113, 704, 555

Pass II:-

	0	1	2	3	4	5	6	7	8	9
710		710								
200	200									
501	501									
211		211								
111		111								
102	102									
222			222							
112		112								
222			222							
113		113								
704	704									
555						555				

After Pass II:-
200, 501, 102, 704, 710, 211, 111, 112, 113, 222, 222, 555
Pass 1II:-

	0	1	2	3	4	5	6	7	8	9
200			200							
501						501				
102		102								
704								704		
710								710		
211			211							
111		111								
112		112								
113		113								
222			222							
222			222							
555						555				

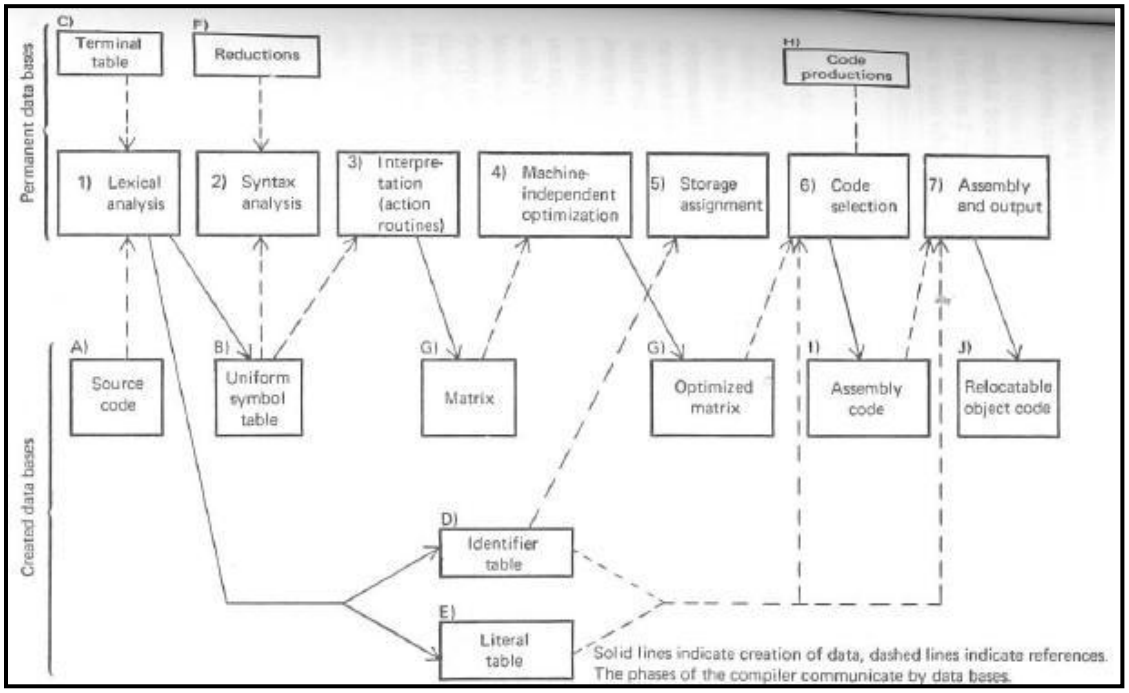
After Pass III

102, 111, 112, 113, 200, 211. 222, 222, 501, 555, 704, 710

iii) Draw and explain the general model of compiler.
(Diagram - 4 marks; Description - 4 marks)

Ans:

In analyzing the compilation of simple program there are seven distinct logical problems as follows and summarized in figure below.



1. **Lexical analysis –** Recognition of basic elements of creation of uniform symbols.
2. **Syntax analysis –** Recognition of basic syntactic constructs through reductions.
3. **Interpretation –** Definition of exact meaning, creation of matrix and tables by action routines.
4. **Machine Independent Optimization –** Creation of more optimal matrix.
5. **Storage Assignment –** Modification of identifier and literal tables. If makes entries in the matrix that allow code generation to create code that allocates dynamic storage and that also allow the assembly phase to reserve the proper amounts of STATIC storage.
6. **Code Generation –** Use of macro processor to produce more optimal assembly code.
7. **Assembly And Output –** Resolving symbolic addresses and generating machine language.

3. Attempt any four : 16

i) Define and describe assemblers and compilers.
(Assembler - 2 marks; compiler - 2 marks)

Ans:

- **Assembler:**
  It is a language translator that takes as input assembly language program (ALP) and generates its machine language equivalent along with information required by the loader.

ALP → Assembler → Machine language equivalent + Information required by the loader

- **Compiler:** Compiler is a language translator that takes as input the source program(Higher level program) and generates the target program (Assembly language program or machine language program)

Source Program → Compiler → Target program

↓

Error Messages

Since the process of compilation is very complex it is divided in to several phases.

(A phase is a logical unit of work that takes as input one representation and generates another representation.)

The phases are as follows

1. Lexical Analysis
2. Syntax analysis
3. Semantic Analysis
4. Intermediate code generation
5. Code optimization
6. Code Generation

ii) Describe Binary search algorithm with suitable example.

(Algorithm explanation - 2 marks; example - 2 marks)

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

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.

iii) Describe the lexical phase of compiler.
(4 marks for explanation)

Ans:

- The first phase of compiler is lexical analysis. It works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

- Algorithm of Lexical Analysis phase of compiler is as follows
  - The first tasks of the lexical analysis algorithm are to the input character string into token.
  - The second is to make the appropriate entries in the tables.
  - A token is a substring of the input string that represents a basic element of the language. It may contain only simple characters and may not include another token. To the rest of the compiler, the token is the smallest unit of currency. Only lexical analysis and the output processor of the assembly phase concern themselves with such elements as characters. Uniform symbols are the terminal symbols for syntax analysis.

- Lexical analysis recognizes three types of token: Terminal symbols, possible identifiers, and literals.
- It checks all tokens by first comparing them with the entries in the terminal table. Once a match is found, the token is classified as a terminal symbol and lexical analysis creates a uniform symbol of type „TRM., and inserts it in the uniform symbol table. If a token is not a terminal symbol, lexical analysis proceeds to classify it as a possible identifier or literal. Those tokens that satisfy the lexical rules for forming identifiers are classified as “possible identifiers”.

- **Databases used in lexical analysis are:**

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

Symbol	Indicator	Precedence

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

Literal	Base	Scale	Precision	Other information	Address

4) **Identifier table:** created by lexical analysis to describe all identifiers used in the source program. There is one entry for each identifier. Lexical analysis creates the entry and places the name of identifier into that entry. The pointer points to the name in the table of names. Later phases will fill in the data attributes and address of each identifier.

Name	Data attributes	address

5) **Uniform Symbol table:** created by lexical analysis to represent the program as a string of tokens rather than of individual characters. Each uniform symbol contains the identification of the table of which a token is a member.

Table	Index

iv) Describe about dynamic binders.
(Explanation - 4 marks)

Ans:

Dynamic binders:

In many cases linkage editing process takes as much time as the complication an execution of the job. In a segmented system it is not necessary to relocate the address within a subroutine segment. It is only necessary to convert symbolic references between segments to numeric segment numbers. By virtue of the presence of a hardware – detected indication in each intersegment reference instructions to differentiate between unlinked reference (in form of character string) as generated by the compiler and actual linked segment numbers, the binding will be performed only when an instruction is encountered that requires the linkage . This is called dynamic binding.

- The dynamic binders are also called as 'linkage editor'.

- It keeps track of the relocation information so that the resulting load module can be further relocated and loaded anywhere in the core.
- In this case the module loader must perform additional allocation and relocation as well as loading.
- It does not have to worry about the complex problems of linking.

v) Compare top down parser and bottom up parser.
(1 marks for each point (any four points.))

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

4. a) Attempt any three : 12

i) Describe the format of database of direct linking loaders.
(1 mark for each)

Ans:

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

ESD card format:

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

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

TXT card format:

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

13-16	Blank
17-72	From 1 to 56 data bytes
73-80	Card sequence number

- **The RLD cards** contain the following information

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

RLD card format:

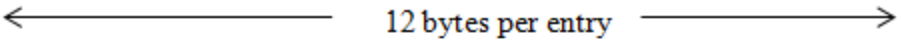
Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters RLD
5-18	Blank
19-20	Relative address of first data byte
21	Blanks
22-24	Byte Count (BC) = number of bytes of information in cc. 17-72
25-72	Blank
73-80	Card sequence number

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

END card format:

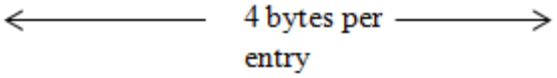
Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters END
5	Blank
6-8	Start of execution entry (ADDR), if other than beginning of program
9-72	Blanks
73-80	Card sequence number

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



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

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



Assigned core address of corresponding symbol (4 bytes) (decimal)

ii) Describe the Interpretation phase algorithm of compiler.
(*Purpose - 2 marks; example - 4 marks; types of statements - 2 marks)*

Ans:

INTERPRETATION PHASE:

This phase is typically a collection of routines that are called when a construct is recognized in syntactic phase. The purpose of these routines is to on intermediate form of the source program and adds information to identifier table. It interprets the precise meaning into the matrix or identifier table.

Databases used:

- **Uniform symbol table:** created by lexical analysis phase and containing the source program in the form of uniform symbols.
- **Stack:** contains tokens currently being parsed by the syntax and interpretation phase.

- **Identifier table:** initialized by lexical analysis to completely describe all identifiers used in the source program.
- **Matrix:** a simple form of matrix entry consist of a triplet where the first element is a uniform symbol denoting the terminal symbol or operator and other two elements are operands.

The compiler creates an intermediate form of source program. It affords 2 advantages

- It facilitates optimization of object code.
- It allows a logical separation between machine independent and machine dependent.

The form depends on syntactic construction

- Arithmetic statements
- Non arithmetic statements
- Non-executable statements

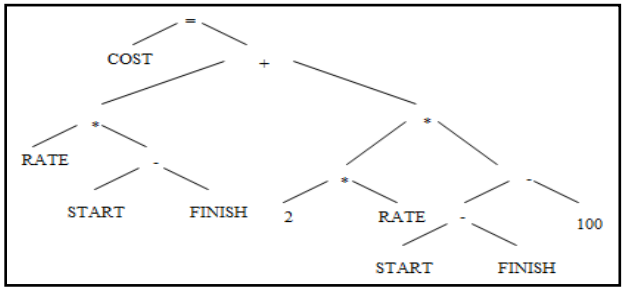
- **Arithmetic statements**
- A source code may contains statements like

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

- These statements are called as arithmetic statements.
- One form is to convert these statements into the PARSE TREE.
- The rules for converting these statements into parse tree are:

i. Any variable is a terminal node of the tree

ii. 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

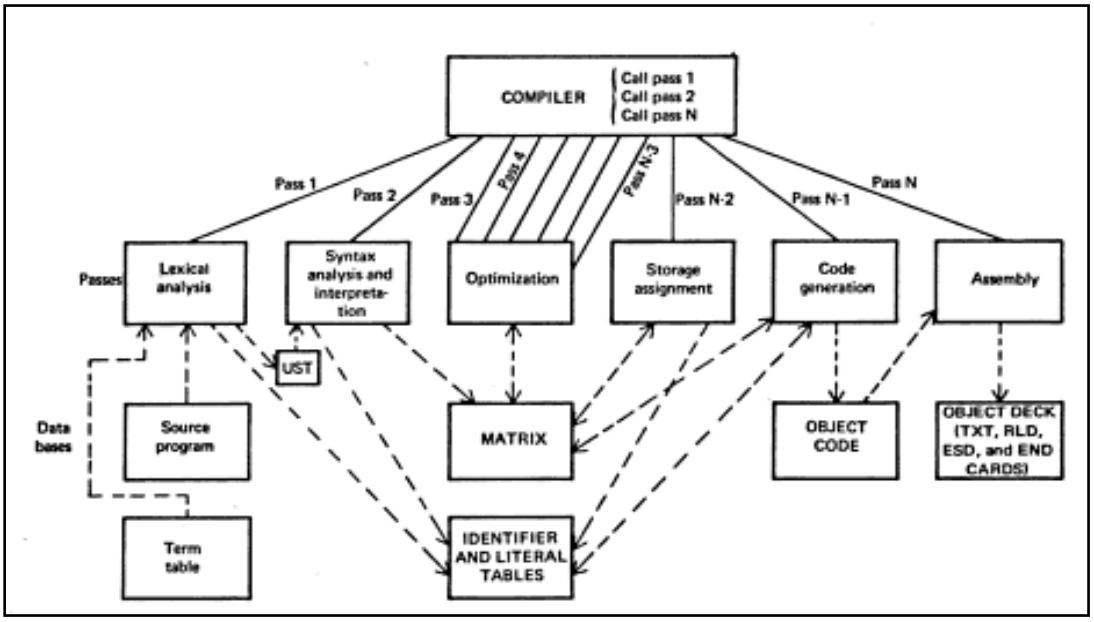


- **Non Arithmetic Statements**
  - The non-arithmetic executable statements are replaced with the sequential ordering of individual matrix entries.

Operator	Operand 1	Operand 2
Return	COST	
END		

iii) Describe the passes of a compiler
(Diagram - 2 marks; explanation - 2 marks)

Ans:



Passes is a logical execution of the compilation process.

1. **Pass 1** This pass is corresponds to lexical analysis phase. This pass scans the source code and creates an identifier, literal and uniform symbol table.
2. **Pass 2** corresponds to syntactic and interpretation phase. It scans the uniform symbol table, produces the matrix and place information about identifier into the identifier table.

3. **Passes 3 to N-3** corresponds to the optimization phase. Each separate type of optimization may required several passes over the matrix.
4. **Pass N-2** corresponds to the storage assignment phase. This is a pass over the identifier and literal tables rather than program itself.
5. **Pass N-1** corresponds to the code generation phase. It scans the matrix and creates the first version of the object deck.
6. **Pass N** corresponds to the assembly phase. It resolves the symbolic addresses and creates information for the loader.

iv) Give the advantages and disadvantages of top down parser.
(Advantages - 2 marks; Disadvantages - 2 marks)

Ans:

Top-down parsers
Advantages

- It is easy to implement
- It is easy to debug
- Smaller code
- Can tokenise quickly

Disadvantages

- It cannot handle left recursion
- Faster Handles left recursion
- It is applicable to small class of grammar

b) Attempt any one: **6**

i) Describe the two pass algorithm of macro.
(Explanation for pass 1 - 3 marks; explanation for pass 2 - 3 marks)

Ans:

PASS – 1 MACRO DEFINATION: The algorithm for pass 1 tests each input line. If it is a MACRO pseudo – op the entire macro definition that follows is saved in the next available locations in the Macro Definition Table (MDT). The first line of the definition is the macro name line. The name is entered into the Macro Name Table (MNT), along with a pointer to the first

location of the MDT entry of the definition. When the end pseudo –op is encountered, all of the macro definition have been processed so control transfers to pass 2 in order to process macro calls

PASS 2 – MACRO CALLS AND EXPANSION: the algorithm for pass 2 tests the operation mnemonic of each input line to see if it is a name in the MNT. When a call is found, the call processor sets a pointer, the macro Definition Table Pointer (MDTP), to the corresponding macro definition stored in the MDT. The initial value of the MDTP is obtained from the "MDT index" field of the MNT entry. The macro expander prepares the Argument List Array (ALA) consisting of a table of dummy argument indices and corresponding arguments to the call. This list is simply a succession of symbols ordered to match the dummy arguments on the name card. Arguments not represented in a call are considered blank, and superfluous arguments are ignored. In the case of argument reference by position, this scheme is completely straightforward. For reference by name, the macro processor locates the dummy argument on the macro name line in order to determine the proper index.

ii) Describe the role of a lexical and syntax analyzer.
(Lexical analyzer - 3 marks; syntax analyzer - 3 marks)

Ans:

Role of a lexical analyzer

1. Identify the lexical units in a source statement
2. Classify units in to different lexical classes
3. Build a token for each lexical unit
4. Ignore comments in the source program,
5. Detect tokens which are not part of the language.

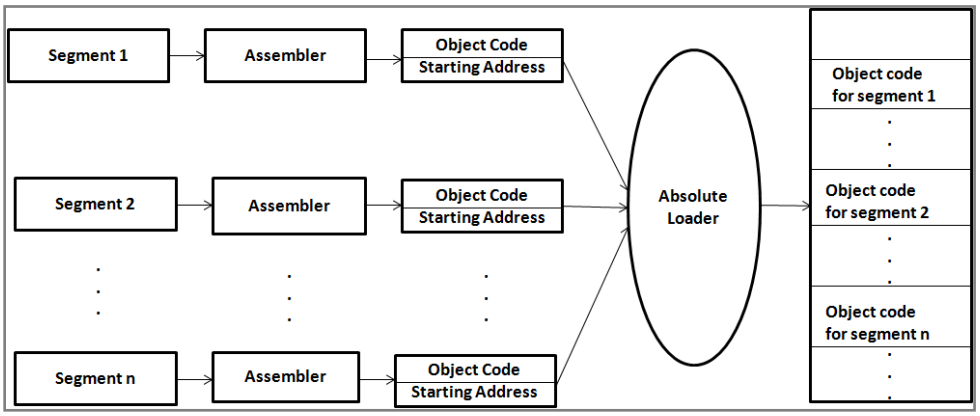
Role of a syntax analyzer

1. Obtain tokens from lexical analyzer
2. Check whether the expression is syntactically correct
3. Report syntax error if any
4. Determine the statement class i.e is it an assignment statement, a conditional statement, etc
5. Group token into statements
6. Construct a parse tree which represents syntactic structure of the program.

5. Attempt any two: **16**

i) Draw and describe the design of absolute loaders.
(Diagram - 3 marks; explanation - 5 marks)

Ans:



Design of absolute loader

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, it's then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is, while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

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

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

Thus the absolute loader is simple to implement in this scheme

1. Allocation is done by either programmer or assembler
2. Linking is done by the programmer or assembler
3. Resolution is done by assembler
4. Simply loading is done by the loader

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.

ii) Describe the optimization phase of a compiler.
(Machine dependent - 4 marks; machine - independent - 4 marks)

Ans:

- Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).
- 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 - independent optimization:

- 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
4	-	START	FINISH	M4
5	-	M4	100	M5
6	*	M3	M5	M6
7	+	M2	M6	M7
8	=	COST	M7	

Matrix with common subexpressions

	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	

Matrix after elimination of common subexpressions

Machine dependent optimization:

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

Optimized Matrix				First try		Improved code				
				L	1, START	L	1, START			
1	-	START	FINISH	S	1, FINISH	S	1, FINISH	M1	→	R1
				ST	1, M1					
				L	1, RATE	L	3, RATE			
2	*	RATE	M1	M	0, M1	MR	2, 1	M2	→	R3
				ST	1, M2					

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

iii) Describe Hashing with suitable example.
(Definition - 2 marks; explanation - 6 marks)

Ans:

- Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.
- 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 th position is void, then the new element is put there, if not then 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).

6. Attempt any four: **16**

i) Describe Macro call within macros with suitable example.
(Explanation - 2 marks; Example - 2 marks)

Ans:

Macro calls are "abbreviations" of instruction sequences, it seems reasonable that such "abbreviations" should be available within other macro definitions. For example,

```
MACRO
ADD1        &ARG
L           1, &ARG
A           1, =F'1'
ST          1, &ARG
MEND
MACRO
ADDS        &ARG1, &ARG2, &ARG3
ADD1        &ARG1
ADD1        &ARG2
ADD1        &ARG3
MEND
```

Source	Expanded source (level 1)	Expanded source (level 2)
. . .		
MACRO ADD1 &ARG L 1, &ARG A 1, = F '1' ST 1, &ARG MEND		
MACRO ADDS &ARG1, &ARG2, &ARG3 ADD1 &ARG1 ADD1 &ARG2 ADD1 &ARG3 MEND		
. . .	. . .	(Expansion of ADD1)
ADDS DATA1, DATA2, DATA3	(Expansion of ADDS) ADD1 DATA1 ADD1 DATA2 ADD1 DATA3	L 1, DATA1 A 1, = F '1' ST 1, DATA1 L 1, DATA2 A 1, = F '1' ST 1, DATA2 L 1, DATA3 A 1, = F '1' ST 1, DATA3
DATA1 DC, F '5' DATA2 DC, F '10' DATA3 DC, F '5'		DATA1 DC, F '5' DATA2 DC, F '10' DATA3 DC, F '5'

Within the definition of the macro 'ADDS' are three separate calls to a previously defined macro 'ADD1'. The use of the macro 'ADD1' has shortened the length of the definition of 'ADDS' and thus had made it more easily understood. Such uses of macros result in macro expansions on multiple 'levels'.

ii) Describe the data structure of assembler.
(2 marks for pass 1; 2 marks for pass 2)

Ans:

Data Structures used in assembler pass 1

- Input source program
- A Location Counter (LC), used to keep track of each instruction's location.
- A table, the Machine-operation Table (MOT), that indicates the symbolic mnemonic, for each instruction and its length (two, four, or six bytes)
- A table, the Pseudo-Operation Table (POT) that indicates the symbolic mnemonic and action to be taken for each pseudo-op in pass 1.
- A table, the Symbol Table (ST) that is used to store each label and its corresponding value.
- A table, the literal table (LT) that is used to store each literal encountered and its corresponding assignment location.
- A copy of the input to be used by pass 2.

Data Structures used in assembler pass 2

- Copy file:- it is prepared by pass 1 to be used by pass 2.
- Location counter: - It is used to assign address to instruction and addressed to symbol defined in the program.
- Machine operation Table (MOT) or Mnemonic Opcode Table (MOT):- It is used to indicate for each instruction.
    - Symbolic mnemonic
    - Instruction length
    - Binary machine opcode.
    - Format.
- 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.
- Symbol Table (ST):- It is used to generate the address of the symbol address in the program.
- Base table (BT):- It indicates which registers are currently specified as base register by USING Pseudo-ops.
- INST workspace: - It is used for holding each instruction and its various parts are being getting assembled.
- PUNCH CARD workspace:- It is used for punching (outputting) the assembled instruction on to cards.
- PRINT LINE workspace: - It is used for generating a printed assembly listing for programmer's reference.
- Object card: - This card contain the object program in a format required by the assembler.

iii) Explain about Relocating loaders.
(For explanation - 4 marks)

Ans:

- Relocating loader avoids possible reassembling of all subroutines when a single subroutine is changed and perform the tasks of allocation and linking for the programmer.
- The BSS loader allows many procedure segments yet only one data segment.
- The assembler assembles each procedure segment independently and passes on to the loader the text and information as to relocation and intersegment references.
- The o/p of a relocating assembler using a BSS scheme is the object program and information about all other program it references.
- For each source program the assembler o/p a text prefixed by a transfer vector that consist of addresses containing names of the subroutines referenced by the source program.
- The assembler would also provide the loader with additional information such as the length of the entire program and the length of the transfer vector position.
- After loading the text and the transfer vector into core, the loader would load each subroutine identified in the transfer vector.
- It would the place a transfer instruction to the corresponding subroutine in each entry in the transfer vector.
- The BSS loader scheme is other used on compiler with a fixed length direct address instruction format.
- The relocation bit solves the problem of relocation, the transfer vector is used to solve the problem of linking and the program length information solves the problem of allocation.

iv) Describe the syntax phase of compiler.
(Explanation - 4 marks)

Ans:

The function of the syntax phase is to recognize the major constructs of the language and to call the appropriate action routines that will generate the intermediate form to matrix for the constructs. Databases involved in syntax analysis are as follows:

Uniform Symbol Table ("UST"): It is created by the lexical analysis phase and containing the source program in the form of uniform symbols. It is used by the syntax and interpretation phases as the source of input to the stack. Each symbol from the UST enters the stack only once.

Stack: the stack is the collection of uniform symbols that is currently being worked on by the stack analysis and interpretation phase. The stack is organized on a Last In First Out (LIFO) basis. The term "Top of Stack" refers to the most recent entry and "Bottom of Stack" to the oldest entry.

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

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.

v) How subroutine linkages are applied in loaders?
(Explanation - 4 marks)

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 (BA L, 14,B) to subprogram B.
3. But assembler does not know the value of this transfer reference and will declare it is an error.
4. To handle this situation a special mechanism is needed.
5. To handle it mechanism is typically implemented with a relocation or direct linking loader.

Subroutine linkage uses following special pseudo ops:

1. **EXTRN :**
   It is used to direct or to suggest loader that subroutine followed by EXTERN are used in the main program but defined in some other program.
2. **ENTRY:**
   It is used to direct or to suggest loader that subroutine followed by ENTRY are defined in the main program but used in some other program.

Example: the following sequence of instructions may a simple calling sequence to another program.

```
MAIN    START
        EXTERN    SOUBROUT
        ------------------
        ------------------
        L     15 = A(SOUBROUT)       //calling soubroutine
        BALR 14, 15
        .
        .
        .
        .
        END
```

The above sequence of instructions first declares SOBROUT as an external variable that is a variable referenced but not defined in this program.
The Load (L) instruction loads the address of that variable to the register '15'.