MAHARASHTRA STATE BOARD OF TECHNICAL EDUCATION
(Autonomous)
(ISO/IEC - 27001 - 2005 Certified)
SUMMER-16 EXAMINATION
Model Answer

Subject Code: 17634 **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) Answer any THREE of the following: **12**

(i) Compare assembler and compiler.
(Any 4 points of comparison - 1 mark each)

Ans:

Assembler	Compiler
It Converts Machine Manipulation Code Directly into Binary Machine Instruction.	It Converts Human Developed Codes into Machine Executable Codes.
It Gives Most Efficient Executable.	Does not always produce the most efficient executable.
Some How Difficult to Work with.	Easiest for humans to program,
Assembler doesn't searches for errors.	A compiler searches all the errors of a program and lists them.
Stores variable in Tables	Trace variable in program
Compilers usually produce the machine executable code directly from a high level language	Assemblers produce an object code which might have to be linked using linker programs in order to run on a machine

(ii) Explain different data structure used by Pass - II of assembler.
(Any 4 Data Structures - 1 mark each)

Ans:

- Copy of source program input to pass 1
- LC: Location Counter: - To keep track of each instruction location
- MOT: Mnemonic Op. 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
01 = 1 half-words = 2bytes
10 = 2 half-words = 4bytes
11 = 3 half-words = 6bytes

Instruction format
000=RR
001=RX
010 = RS
011 = SI
100 = SS

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

- **POT: Pseudo Op. 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

- **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:

- **BT (Base table):-**

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

15 entries

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**

(iii) State and explain four basic tasks of macro processor.
(1 mark each task of Macro Processor)

Ans:

1. 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 the macro 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.

MDT

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

2. Saving the definitions:

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

MNT

Macro Name Table		
1	"INCRaaaa"	15
.	.	.
.	.	.
.	.	.

3. 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.

ALA

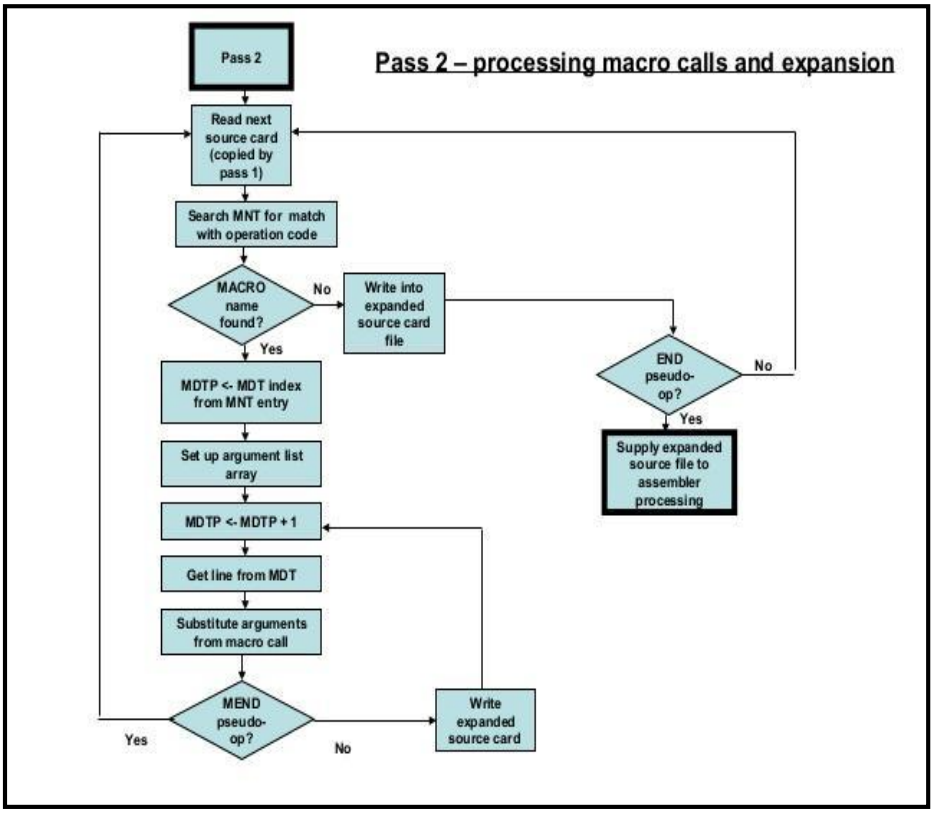
Index	**Argument**
0	"bbbbbb" (All Blank)
1	"Data3bbb"
2	"Data2bbb"
3	"Data1bbb"

4. Replacing macro definitions with macro calls:

The pre-process or needs to expand macro calls and substitute arguments when any macro call is encountered. The pre-processor must substitute macro definition arguments within a macro call.

(iv) Draw flowchart for processing macro calls and expansion in II-Pass macro processor.
(Correct Flowchart - 4 marks)

Ans:



b) Answer any ONE of the following: 6

(i) Explain different components of system software.
(List - 1 mark; description - 1 mark each)

Ans:

Components of system software are:

1. Assembler
2. Macros
3. Loader
4. Linker
5. Compiler

1. Assembler:
It is a language translator that takes as input assembly language 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

2. Macros:
The assembly language programmer often finds that certain set of instructions get repeated often in the code. Instead of repeating the set of instructions the programmer can take the advantage of macro facility where macro is defined to be as "Single line abbreviation for a group of instructions".
The template for designing a macro is as follows

```
MACRO //Start of definition
Macro Name
----------
----------
MEND //End of def.
```

3. Loader:
It is responsible for loading program into the memory, prepare them for execution and then execute them. Loader is a system program which is responsible for preparing the object programs for execution and start the execution. Functions of loader:
Allocation: Allocate the space in the memory where the object programs can be loaded for execution.
Linking: Resolving external symbol reference
Relocation: Adjust the address sensitive instructions to the allocated space.
Loading: Placing the object program in the memory in to the allocated space.

4. **Linker**:
   A linker which is also called binder or link editor is a program that combines object modules together to form a program that can be executed. Modules are parts of a program.
5. **Compiler**:
   Compiler is a language translator that takes as input the source program (Higher level program) and generates the target program (Assembly language program or machine language program)

(ii) What is the difference between:

1) **Phase and pass**
2) **Syntax analysis and semantic interpretation**
3) **Token and uniform symbol**

(Phase and pass - 2 marks; Syntax analysis and semantic interpretation - 2 marks; Token and uniform symbol - 2 marks)

Ans:

a. Phase and Pass

A pass is a single time the compiler passes over (goes through) the sources code or some other representation of it. Typically, most compilers have at least two phases called front end and back end, while they could be either one-pass or multi-pass.

Phase is used to classify compilers according to the construction, while pass is used to classify compilers according to how they operate.

b. Syntax analysis and semantic interpretation

Syntax Analysis phase takes as input the tokens generated by lexical phase and if the syntax of the statement is correct it generates a parse tree representation.

Semantic interpretation phase performs the check on the meaning of the statement and performs the necessary modifications in the parse tree representation.

c. Token and uniform symbol

Token:-

The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

Uniform Symbol:-

There is one uniform symbol for every token of which the token is member and its index within that table.

2. Answer any TWO of the following: 16

a) Write the content of symbol table,literal table POT and MOT after Pass I of assembler for following code:

```
SIMPLE    START
          BALR     15, 0
          USING    *, 15
LOOP      L        R1, TWO
          A        R1, TWO
          ST       R1, FOUR
          CLI      FOUR +3, 4
          BNE      LOOP
          BR       14
R1        EQU      1
TWO       DC       F'2'
FOUR      DS       F
          END
```

(Symbol Table - 2 marks; Literal Table - 2 marks; POT - 2 marks; MOT - 2 marks)

Ans:

Symbol Table

Symbol	Value	Length	Relocation
SIMPLE	0	1	R
LOOP	2	1	A
R1	3	1	A
TWO	4	1	A
FOUR	5	1	R

Literal Table

Literal	Value	Length	Relocation
A	0	1	R
F'2'	2	1	A
F	3	1	A

Machine Op Table

Mnemonic Op-Code	Binary Op-Code	Instruction Length	Instruction Format	Not used in the design
Lbbb	D2	10	100	
Abbb	5A	10	001	
ST	3E	03	100	
CLI	6B	01	100	
BNE	1D	03	001	
BR	8C	03	000	

Pseudo-Op Table

Pseudo-Op	Address of routine to process Pseudo-Op
START	P1START
USING	P1USING
ENDbb	P1END
EQUbb	P1EQU
DCbbb	P1DC
DSbbb	P1DS

b) Explain the following terms:
(i) Parameter passing in macro
(ii) Nested macro calls
(iii)Conditional macro
(iv)Procedure

Ans:

(i) Parameter passing in Macro:

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
.		

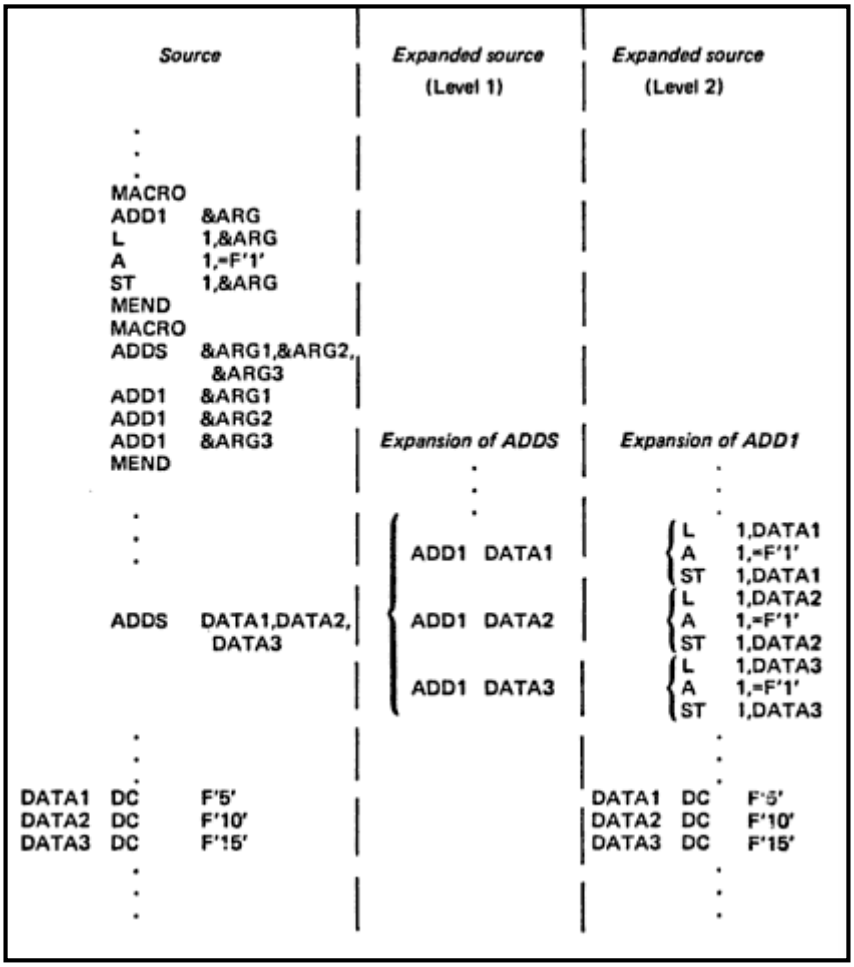
.		
.		
DATA1 DC	F'5'	
DATA2 DC	F'10'	
.		
.		
.		

(ii) Nested Macro calls:

Since 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
```

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 has made it more easily understood.



Macro call within macros can involve several levels. For example, the macro ADDS might be called within the definition of another macro. 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. It makes perfectly good sense.

(iii) Conditional Macro:

Two important macro processor pseudo ops, AIF and AGO, permit conditional reordering of the sequence of macro expansion. This allows conditional selection of the machine instructions that appear in expansions of a macro call.

AIF is a conditional branching pseudo-op; it performs an arithmetic test and branches only if the tested condition is true. The AGO is an unconditional branch pseudo-op or 'Go to' statement. If 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.

(iv) Procedure:

Procedure may be either internal or external. An internal procedure is included as part of a single program module. An external procedure is a separate program module that s compiled separately and is linked together with the other program modules by means of the loader. Furthermore procedures may have multiple entry points. This feature can be very useful for subroutines that are very similar such as the SINE and COSINE routines. It can be written as follows.

```
N_FACTORIAL : PROCEDURE(IN) RECURSIVE;
        IF N=0 THEN RETURN(1)
        ELSE RETURN(N*N_FACTORIAL(N-1));
END;
```

c) Explain how symbolic names of subroutines are used in relocation and linking in BSS loader and dynamic loader.

(Appropriate description - 8 marks)

Ans:

The changes that are necessary for use of a relocatable loader have implications for the assembly language which produces the input for the loader. The MIXAL language which is used on the MIX computer is an absolute assembly language; it is used in connection with an assembler which produces loader input for an absolute loader. It has no provisions for generating relocatable loader code. A relocatable MIXAL would vary in several respects.

One variation would be in the ORIG statement; it would be allowed in only limited ways, if at all. A programmer would no longer be able to ORIG her code to an arbitrary absolute address. The only uses which would be permitted would be relative ORIGs, like ORIG *+10, ORIG N*2+*, and so forth. Many assemblers thus replace the ORIG statement with a BSS statement. The BSS statement stands for "Block Storage Save" and takes one expression in its operand field. A BSS 10 is identical to an ORIG 10+*, a BSS N*2 to an ORIG N*2+*, and so on.

Another change is in allowed address expressions. Each symbol is either absolute or relocatable. Expressions can also be typed as absolute or relocatable. An absolute symbol is an absolute expression, and a relocatable symbol is a relocatable expression. The sum, difference, product, and quotient of absolute expressions are absolute expressions. Relocatable expressions are more difficult to define. A relocatable expression is one whose final value depends upon the base address in the same way as a relocatable symbol. The binding from relocatable to absolute should be a simple addition of a base. Let R be a relocatable symbol or expression, and let A be an absolute symbol, constant, or expression. Then the expressions, R+A, R-A, and A+R are relocatable. An expression like R-R is absolute. Expressions like R+R, R*R, R/R, A*R, or R/A are neither relocatable nor absolute and hence are not allowed.

Notice that either R or A can be an expression, and so expressions like R-R-A+R+A-R+R are allowed (and are relocatable). To determine if an expression is either relocatable or absolute,

replace all relocatable symbols by R and all absolute symbols by A. Then check that no relocatable symbols are involved in multiplications or divisions. Finally combine sub-expressions such as R+A, A+R, R-A and substitute with R, and A+A, A-A, A/A, A*A, and R-R, substituting with an A until the entire expression is reduced as far as possible. If the result is either R or A, the expression is valid and of the indicated type; otherwise it is illegal.

One other change is concerned with externals and entry points. To the assembler of a segment, any reference to an external will appear to be a reference to an undefined symbol, which is not what is meant. One possible approach to this would be to treat all undefined symbols as externals, but this would result in truly undefined symbols being treated incorrectly. Thus a new pseudo-instruction is introduced which is used to notify the assembler that a symbol is meant to be an external symbol, not an undefined one. This new pseudo-instruction, EXT, could be of the form

EXT <list of external symbols>
or alternatively

<symbol> EXT <operand field ignored>

Many assemblers require external symbols to be declared as such before their first use in the segment and often place restrictions on the use of externals in address expressions, allowing only an external by itself, or plus or minus an absolute expression.

Corresponding to the externals are entry points and they are treated in much the same way. In order for the loader to correctly link externals with the corresponding entry point, the loader must know where the entry points are. Thus assembly languages often include an entry point declaration pseudo-instruction ENT. The form of this pseudo instruction is often

ENT <list of entry point symbols>

Any symbol listed in an ENT pseudo-instruction must be defined as a relocatable symbol elsewhere in the segment, and can then be used by the loader for linking.

In addition to these programmer-visible changes in assembly language programs, the assembler itself must also produce relocatable, not absolute, loader code.

3. Answer any FOUR of the following: **16**

a) Outline the algorithm for syntax analysis phase of complier.

(Explanation with example - 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.

b) Describe the I/P and O/P of the macro processor. How it is dependent upon the assembler source code.

(I/P -1 mark; O/P -1 mark; Description - 2 marks)

Ans*:*

Input to Macro Processor:

- Source code with macro call;

Output of Macro Processor:

- Source code with macro expansion.

The process of replace is called expanding the macro. Notice that the macro definition itself does not appear in the expanded source code. The definition is saved by the macro processor. The occurrence in the source program of the macro name, as an operation mnemonic to be expanded, is called macro call.

The macro processor can be added as a pre-processor to an assembler, making a complete pass over the input text before pass 1 of the assembler. The macro processor can also be implementing within pass 1 of the assembler.

The implementation of the macro processor within pass 1 eliminates the overhead of intermediate files, and we can improve this integration of macro process and assembler by combining similar functions.

c) Write a binary search algorithm with suitable example.

(Algorithm - 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.

d) Define operating system. Enlist its functions.
(Definition - 1 mark; any 3 functions - 1 mark each)

Ans:

The operating system is the core software component of the computer system. It is an interface between the computer software and hardware components.

Functions of operating system:

1) Controls peripheral devices connected to the computer.
2) Transfers files between main memory and secondary memory storage, manages file folders, allocates the secondary storage space, and provide file protection and recovery.
3) Allocates the use of RAM to the requesting processes.
4) Allow computer to run other applications.

e) Name the machine independent and dependent phase of complier and justify your answer.

(Machine independent - 1 mark; machine dependent - 1 mark; justification - 2 marks)

Ans:

The compiler having seven phases of the compiler model as follow

Machine independent:-

1. Lexical phase
2. Syntax phase
3. Interpretation phase
4. Machine independent Optimization

Machine dependent:-

1. Storage Assignment
2. Code generation
3. Assembly Phase

A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of sub processes called phases.

Marks

4. a) Answer any THREE of the following: 12

(i) Explain the database used in lexical phase.
(1 mark for each database)

Ans:

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.

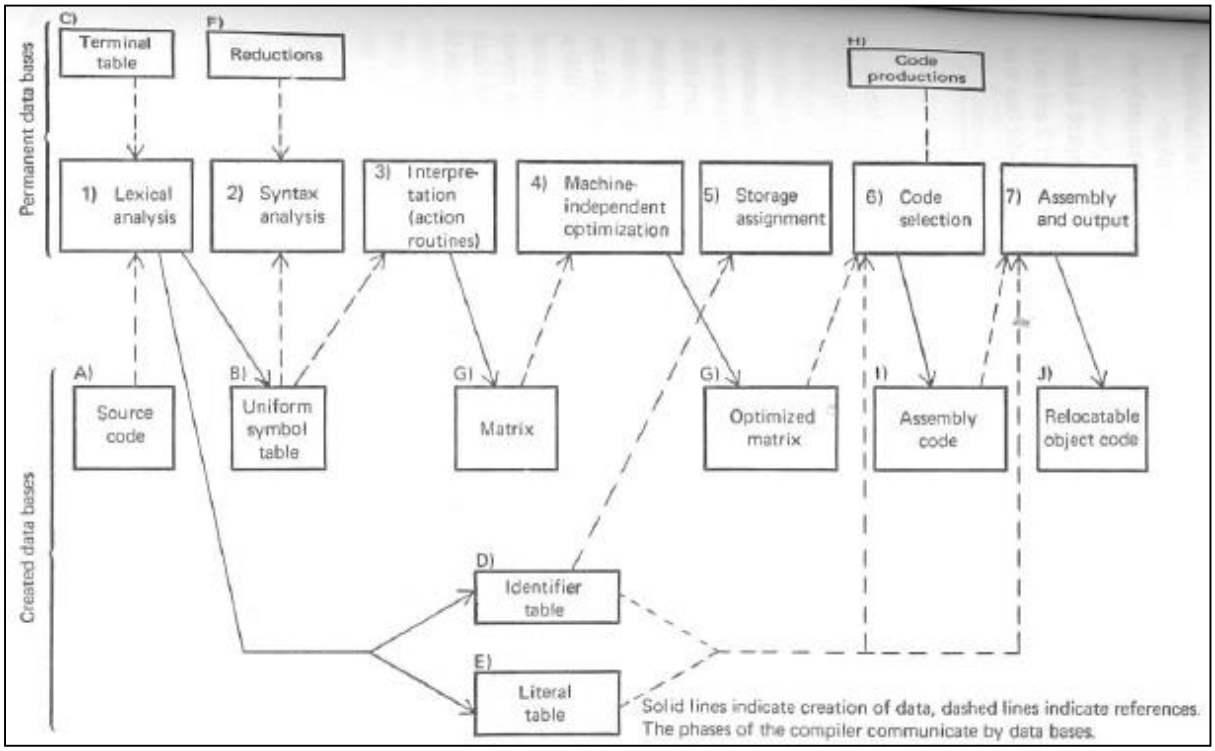
Name	Data attributes	Address

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

(ii) Draw block diagram of the phase of complier and indicate the main function of each.
(Diagram - 2 marks; functions - 2 marks)

Ans:



Lexical Analysis
The first phase of scanner 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>
```

Syntax Analysis
The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis
Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether

identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. 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).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

(iii) Define following terms:

1) **Searching**
2) **Sorting**
3) **Hashing**
4) **Mnemonic**

(1 mark for each definition)

Ans:

1) **Searching:** Searching allows to find data that meets specific criteria. Searching is required to search an assembler's symbol table. Different searching techniques are linear search and binary search.
2) **Sorting:** Sorting allows you to organize the data in ascending or descending order. Sorting is required to arrange symbol table generated by assembler in ordered fashion. Different sorting techniques are interchange sort, bucket sort, address calculation sort.
3) **Hashing:** Hashing is the transformation of a string of characters into usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find shorter hashed key than to find it using the original value.
4) **Mnemonics:** a mnemonic is an abbreviation for an operation. It is entered in the operation code (OPCODE) field of each assembler program instruction. For eg, in assembly language programming INC (increase by one) is a mnemonic.

(iv) Show the result of each pass for following using radix sort 00100, 10001, 01011, 0001, 00101, 00000, 01001, 10101 etc.
(1 mark for each pass)

Ans:

Pass 1:00000

0	**00100,00000**
1	**10001, 01011, 00001, 00101, 01001, 10101**

Sorted array: 00100, 00000, 10001, 01011, 00001, 00101, 01001, 10101

Pass 2:00000

0	**00100,00000, 10001, 00001, 00101, 10101**
1	**01011, 10110**

Sorted array: 00100, 00000, 10001, 00001, 00101, 10101, 01011, 10110

Pass 3: 00000

0	**00000, 10001, 00001, 01011**
1	**00100, 00101, 10101, 10110**

Sorted array: 00000, 10001, 00001, 01011, 00100, 00101, 10101, 10110

Pass 4: 00000

0	**00000, 10001, 00001, 00100, 00101, 10101, 10110**
1	**01011**

Sorted array: 00000, 10001, 00001, 00100, 00101, 10101, 10110, 01011

Pass 5: 00000

0	**00000, 00001, 00100, 00101, 01011**
1	**10001, 10101, 10110**

Sorted array: 00000, 00001, 00100, 00101, 01011, 10001, 10101, 10110

b) Answer any ONE of the following: **6**

(i) Define parser. Draw the parse tree for the string 'abccd' using top down parser
(Definition - 2 marks, parse tree - 4 marks)

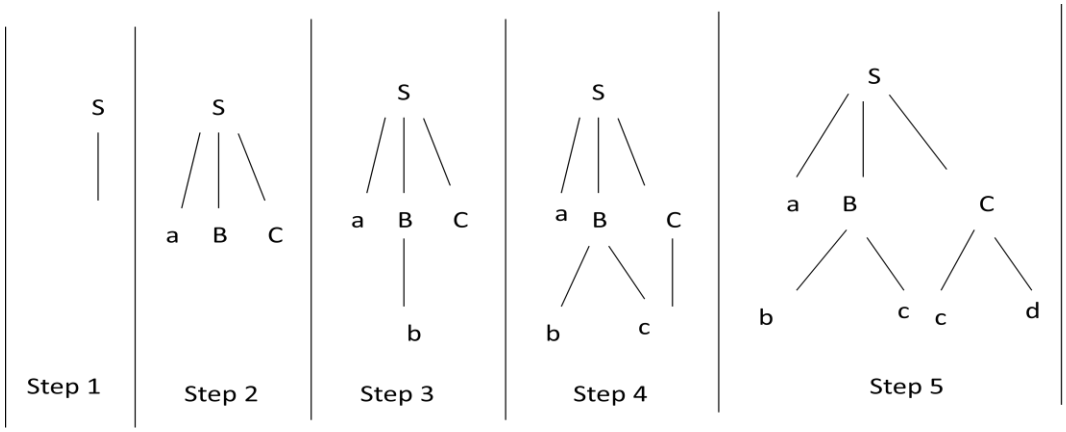
Ans:

Parser: A parser is a program that receives input in the form of sequential source program instructions, interactive online commands, markup tags, ets and breaks them up into parts such as mnemonics, symbols, objects, methods, etc that can then be managed by other phases of compiler. Parser is also called as "Syntax analyzer".

Parse tree for the string 'abccd' using top down parser.
String is "abccd"
Assume:
S→ xyz | aBC
B → b | bc
C → dc | cd

Steps

- Assertion 1 : abccd matches S
  - Assertion 2: abccd matches xyz:
  - **Assertion is false. Try another.**
  - Assertion 2 : abccd matches aBC i.e bccd matches BC:
    - Assertion 3 : bccd matches cC i.e ccdd matches C:
      - Assertion 4 : ccd matches dc:
      - False.
      - Assertion 4 : ccd matches dc:
      - False.
    - Assertion 3 is false. Try another.
    - Assertion 3 : ccd matches bcC i.e cd matches C:
      - Assertion 4 : cd matches dc:
      - False.
      - Assertion 4 : cd matches cd:
      - **Assertion 4 is true.**
    - **Assertion 3 is true.**
  - **Assertion 2 is true.**
- **Assertion 1 is true.**



(ii) Describe token with respect to lexical analysis with suitable example.
(Explanation - 4 marks, suitable example - 2 marks)

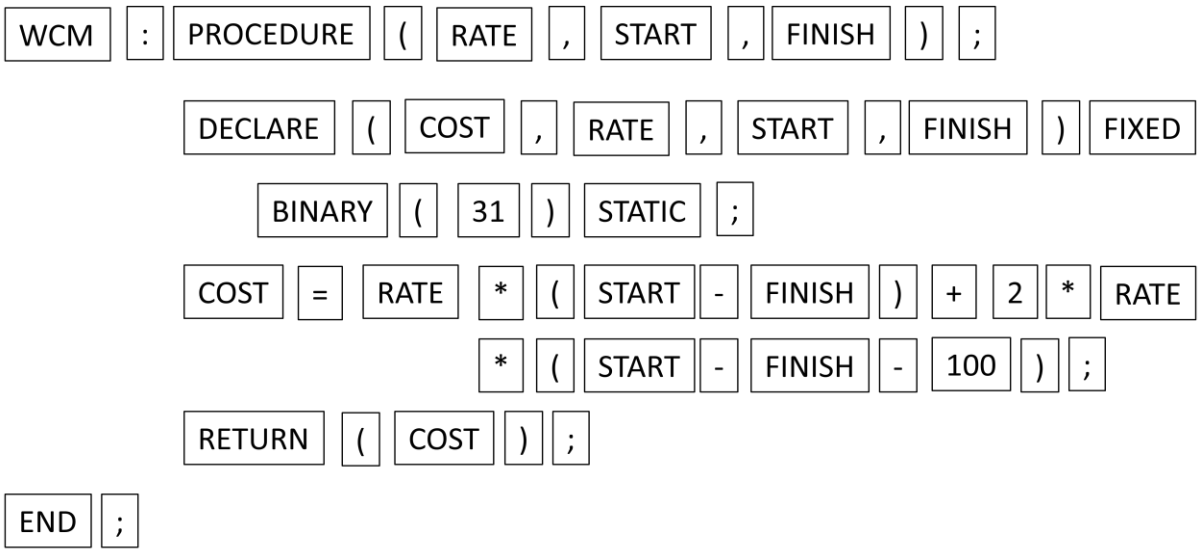
Ans:

- The first tasks of the lexical analysis algorithm are to the input character string into token.
- A token is a substring of the input string that represents a basic element of the language. It may contain only simple characters and may not include another token. To the rest of the compiler, the token is the smallest unit of currency. Only lexical analysis and the output processor of the assembly phase concern themselves with such elements as characters. Uniform symbols are the terminal symbols for syntax analysis.
- Lexical analysis recognizes three types of token: terminal symbols, possible identifiers, and literals.
- It checks all tokens by first comparing them with the entries in the terminal table. Once a match is found, the token is classified as a terminal symbol and lexical analysis creates a uniform symbol of type "TRM", and inserts it in the uniform symbol table.
- If a token is not a terminal symbol, lexical analysis proceeds to classify it as a possible identifier or literal. Those tokens that satisfy the lexical rules for forming identifiers are classified as "possible identifiers".

Example:
Consider following program

```
WCM: PROCEDURE(RATE,START,FINISH);
        DECLARE (COST,RATE,START,FINISH) FIXED BINARY
                (31)STATIC;
        COST = RATE * (START-FINISH) + 2*RATE*(START-
                FINISH-100);
        RETURN (COST);
END;
```

Lexical analysis – tokens of example program



5. Answer any TWO of the following: **16**

a) At what point of time do each of the following loading schemes perform binding

→Direct linking loader **→BSS loader**

→Absolute loader **→Dynamic linking loader**

(explanation -2 marks each)

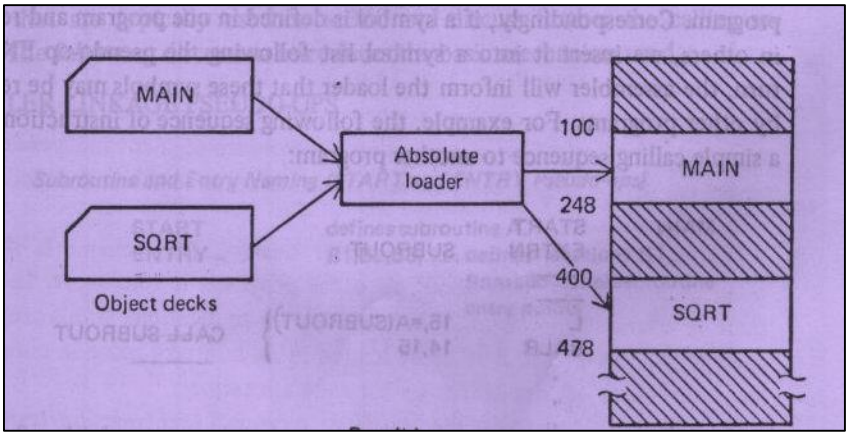
Ans:

Absolute loader

The simplest type of loader scheme, which fits the general model of Figure 5.3, is called an absoluteloader. In this scheme the assembler outputs the machine language translation of the source program in almost the same form as in the "assemble-and-go" scheme,except that the data is punched on cards (object deck) instead of being placed directly in memory .The loader in turn simply accepts the machine language text and place it into core at the location prescribed by the assembler .This scheme makes more core available to the user since the assembler is not in memory at load time.

Absolute loaders are simple to implement but they do have several disadvantages .First, the programmer must specify to the assembler the address in core where the program is to be loaded furthermore ,if there are multiple sub-routines ,the programmer explicitly in his other subroutines to perform subroutine linkage.

Figure 5.4 illustrate the operation of an absolute assembler and an absolute loader. The programmer must be careful not to assign two subroutines to the same or overlapping locations.



Direct linking loader

Adirect –linking loader is a general relocatable loader,and is perhaps the most popular loading scheme presently used.we will assume that the system employs such a loader .
The direct – linking loader has the advantage of allowing the programmer multiple procedure segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments .This provides flexible intersegment referencing and accessing ability, while at the same time allowing independent translations of programs.
In this section we present a general format for the assembler output with such a loading scheme, patterned after those used in the IBM 370.While the formats themselves are somewhat arbitrary, the information that the assembler must give to the loader is not .The assembler (translator) must give the loader the following information with each procedure or data segment:

1. The length of segment
2. A list of all the symbols in the segment that may be referenced by other segments and their relative location within the segment
3. A list of all symbols not defined in the segment but referenced in the segment
4. Information as to where address constants are located in the segment and a description of how to revise their values
5. The machine code translation of the source program and the relative addresses assigned

Dynamic linking loader

A major disadvantage of all the previous loading schemes is that if a subroutine is referenced but never executed (e.g, if the programmer had placed a call statement in his program but this statement was never executed because of a condition that branced around it), the loader would still incur the overhead of linking the subroutine.

Further more, all of these schemes require the programmer to explicitly name all procedures that might be called, it is not possible to write programs as follows:

LOADERS

:
.
READ SUBNAME, ARGUMENT
ANSWER = SUBNAME (ARGUMENT)
PRINT ANSWER
.
:

Where the name of the subroutine (e.g., SQRT, SINE, etc.) is an input parameter, SUBNAME, just like the other data.

BSS loader

The changes that are necessary for use of a relocatable loader have implications for the assembly language which produces the input for the loader. The MIXAL language which is used on the MIX computer is an absolute assembly language; it is used in connection with an assembler which produces loader input for an absolute loader. It has no provisions for generating relocatable loader code. A relocatable MIXAL would vary in several respects. One variation would be in the ORIG statement; it would be allowed in only limited ways, if at all. A programmer would no longer be able to ORIG her code to an arbitrary absolute address. The only uses which would be permitted would be relative ORIGs, like ORIG *+10, ORIG N*2+*, and so forth. Many assemblers thus replace the ORIG statement with a BSS statement. The BSS statement stands for "Block Storage Save" and takes one expression in its operand field. A BSS 10 is identical to an ORIG 10+*, a BSS N*2 to an ORIG N*2+*, and so on.

b) With neat diagram describe the analysis and synthesis phase of general model of compiler.
(Diagram - 2 marks; analysis phase - 3 marks; synthesis phase - 3 marks)

Ans:

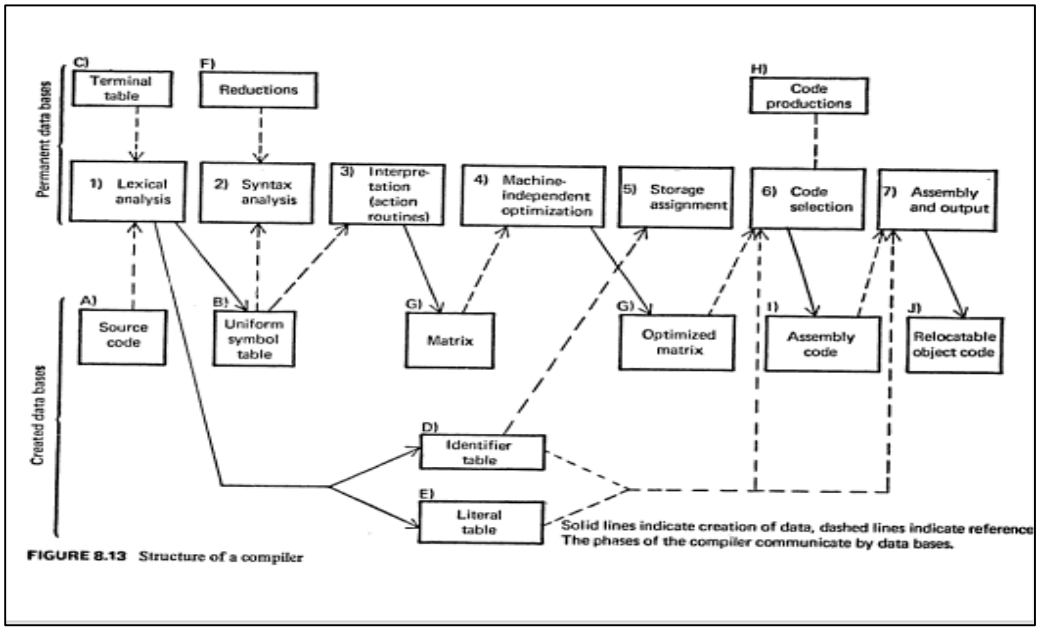


FIGURE 8.13 Structure of a compiler

1) Lexical Phase:-

Its main task is to read the source program and if the elements of the program are correct it generates as output a sequence of tokens that the parser uses for syntax analysis.

The reading or parsing of source program is called as scanning of the source program.

It recognizes keywords, operators and identifiers, integers, floating point numbers, character strings and other similar items that form the source program.

The lexical analyzer collects information about tokens in to their associated attributes.

2) Syntax Phase:-

In this phase the compiler must recognize the phases (syntactic construction); each phrase is a semantic entry and is a string of tokens that has meaning, and 2nd Interpret the meaning of the constructions.

Syntactic analysis also notes syntax errors and assure some sort of recovery. Once the syntax of statement is correct, the second step is to interpret the meaning (semantic). There are many ways of recognizing the basic constructs and interpreting the meaning.

Syntax analysis uses a rule (reductions) which specifies the syntax form of source language.

This reduction defines the basic syntax construction and appropriate compiler routine (action routine) to be executed when a construction is recognized.

The action routine interprets the meaning and generates either code or intermediate form of construction.

c) Write the content of MNT and MDT FOR FOLLOWING CODE

```
MACRO
ONE                 &ARG
        L           1,&ARG
        A           1,=f'1'
        ST          1,&ARG
MEND
MACRO
TWO     &ARG1       &ARG2       &ARG3
        ONE         &ARG1
        ONE         &ARG2
        ONE         &ARG3
MEND
```

(MNT table - 4 marks; MDT table - 4 marks)

Ans:

MNT table

M

Index	8bytes name	4 bytes MDT index
1	ONE	1
2	TWO	6

MDT Table

INDEX	MACRO DEFINITION TABLE(MDT) CARD (80 BYTES/ENTRY)
1	ONE &ARG
2	L 1,#1
3	A 1,=F'1'
4	ST 1,#1
5	MEND
6	TWO &ARG1,&ARG2,&ARG3
7	ONE #1
8	ONE #2

9	ONE #3
10	MEND

6. Answer any FOUR of the following : 16

a) What kind of error that can be detected in a source program during syntatic analysis.
(Any two kinds of errors - 2 marks each)

Ans:

Errors Detected in Syntatic analysis:-

- Syntactic errors include misplaced semicolons or extra or missing braces; that is, "{" or " } . " As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error.
- Unbalanced parenthesis in expressions is handled.
- A missing delimiter ';' cannot be inserted in order to parse the rest of the list. This could lead to an infinite loop in the parser.
- In Syntatic analysis , it checks the code with predefine grammar and find all Grammatical errors like arithmetic, logical, syntactical so on; and display the message as per error.

b) What is the purpose of ID number on ESD card? Why it is not needed for locally define symbol.
(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.

c) List the specification of data structures in direct linking loader.
(Any four data structure - 4 marks)

Ans:

Direct linking loader

The ESD card contain the information necessary to build the external symbol. The external symbol 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 contains the blocks of data and the relative address at which data is to be placed.

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

TXT card format:

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

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.

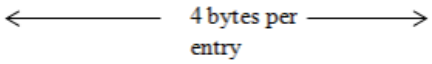
RLD card format:

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:

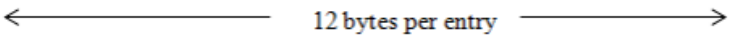
Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters END
5	Blank
6-8	Start of execution entry (ADDR), if other than beginning of program
9-72	Blanks
73-80	Card sequence number

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



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

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



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

d) Explain binary search algorithm with example.
(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] up to list [5]
Pass 1
Low=0
High = 5
Mid= (0+5)/2 = 2
So list[2] = 3 is less than 7
Pass 2
Low= (Mid+1)/2 i.e (2+1)/2 = 1
High = 5
Mid= (1+5)/2 = 6/2 = 3
So list [3] = 11 and the number if found.

e) Explain the following terms:
(i) Segment
(ii) Card
(iii)Core
(iv)Deck
(1 mark each)

Ans:

- **Segment:-** Parts of memory is called as segment
- **Card: -** card is a piece of stiff paper that contains information represented by the presence of absence of holes in predefined positions. The information might be data for data processing applications
- **Core:** - single computing component with two or more independent actual processing units called "core" or Main Memory (RAM) is called as Core.
- **Deck: -** A sequence of cards that is input to or output from some step in an application's processing is called a card deck or simply deck.