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

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 themodel 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.

1. a) Attempt any THREE of the following: **Marks 12**

(i) Define system software and list components of system software. Describe any one.
(Definition - 1 mark; List - 1 mark; Description - 2 marks)

Ans:

System programs 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. The software that controls the operations of a computer system. It is a group of programs rather than one program.

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

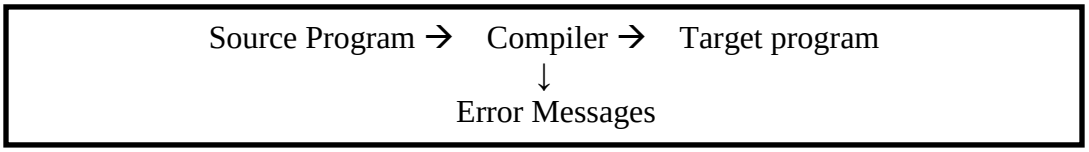
2. Macros:
The assembly language programmer often finds that certain set of instructions get repeated often in the code. Instead of repeating the set of instructions the programmer can take the advantage of macro facility where macro is defined to be as "Single line abbreviation for a group of instructions".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



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

(ii) Explain the use of following pseudo – ops –
USING, START, DC, DS
(1 mark for each pseudo-ops)
Ans:

a) USING
It indicates to the assembler which general register use as a base and what are its contents will be. This is necessary as no special register are sent aside for addressing thus the programmer

must inform the assembler which register(s) to use and how to use them. Since address are relative he can indicate to the assembler the address contained in the base register. The assembler is thus able to produce the machine code with the correct base register and offset.

b) START

It is pseudo op that tells the assembler where the beginning of the program is allows the user to give a name to the program. This statement indicates the start of assembly language program.

Eg**:** PRG1 START

PRG1 START 0

PRG1 START 100

c) DC

It is a pseudo op that is used to define a constant, which places constant value as full words in the memory.

DC<value>: Declare Constant (DC) constructs a memory word containing constant.

d) DS: it is used to define storage of specified memory location

DS: [Label] DS<constant>: Declare Storage (DS) reserves memory area and associates name with the memory area.

(iii) State and explain the tasks of macroprocessor.

(1 mark for each task; 1 mark if only tasks are listed)

Ans:

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

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

1) Recognize the Macro definitions:- A microprocessor must recognize macro definitions identified by the MACRO and MEND pseudo-ops. When MACROS and MENDS are nested, the macro-processor must recognize the nesting and correctly match the last or outer MEND with the first MACRO.

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

3) Recognize the Macro calls: - The processor must recognize macro call that appear as operation mnemonics. This suggests that macro names be handled as a type of opcode.

4) Perform Macro Expansion:- The processor must substitute for macro definition arguments the corresponding arguments from a macro call, the resulting symbolic text is then substituted for the macro call.

(iv) State the difference between macropreprocessor and macroassembler.
(1 mark for each point of distinguish; any other point shall be considered)
Ans:

MACRO PROCESSOR	MACRO ASSEMBLER
It processes macro call and macro definition.	It processes source program
Usage of macro processor is optional. i.e. it is only used when macro is used in source program	Every source program will be accessed by assembler as it has to be process machine language.
Macro processor triggers only when it encounter MACRO pseudo-op.	Macro assembler triggers when program is executed
It has data structure like MDT, MNT,ALA	It has data structure like POT,MOT,LT,ST etc.

b) Attempt any ONE of the following: Marks 6

(i) Explain evolution of system software.
(Any relevant description shall be considered - 2 marks, loader - 2 marks, linkers - 2 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. Loader is a program that place program into memory and prepare them for execution.
5. Assembler could place the object program but leaving assembler in memory waste the memory and programmer would have to translate program with each execution thus wasting translation time.
6. Loader overcomes these problems.
7. The relocating loader allowed the users of the sub-programs to write each sub-program as it starts at location zero.
8. 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.

9. Linkers had to deal with object code generated by high level programming languages such as FORTRAN.
10. Compiler provides complex facilities and complex accessing methods for pointer variables and data structure used in high level language.

(ii) Draw the block diagram of phases of a compiler and state the main function of each phase.

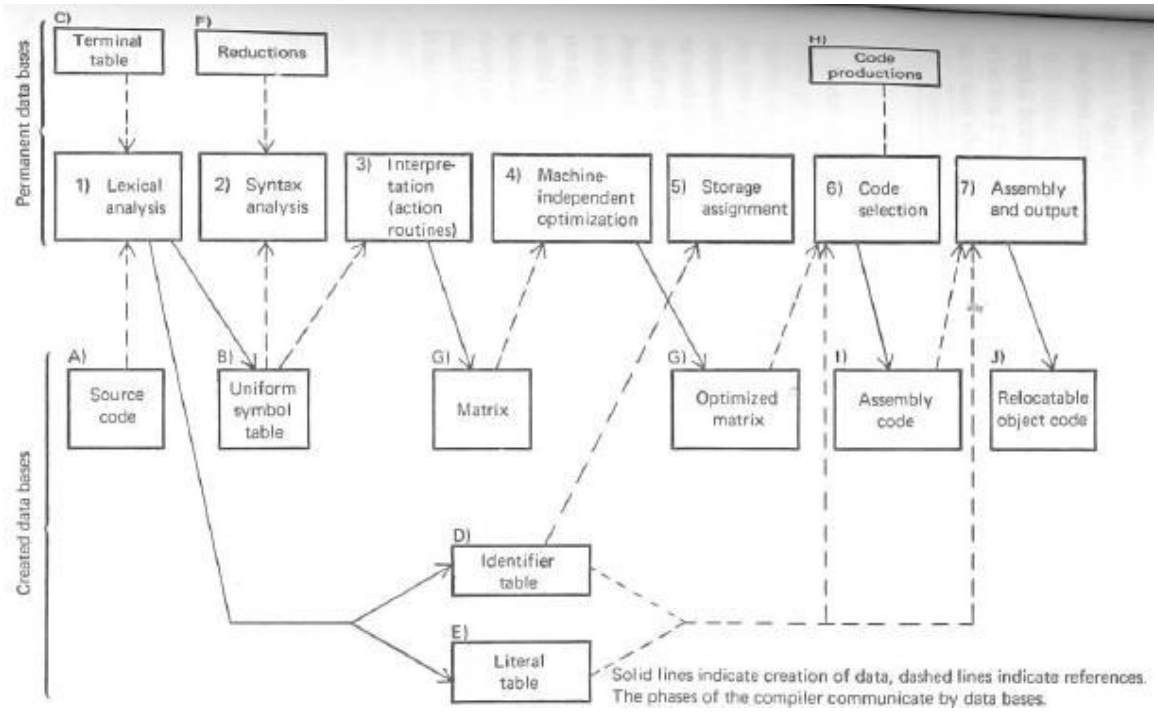
(Diagram - 3 marks; Functions - 3 marks)

Ans:

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

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.

3. Interpretation Phase:-

This phase is typically a routine that are called when a construct is recognized. The purpose of these routines is to on intermediate form of source program and adds information to identifier table.

4. Code optimization Phase:-

Two types of optimization is performed by compiler, machine dependent and machine independent. Machine dependent optimization is so intimately related to instruction that the generated. It was incorporated into the code generation phase. Where Machine independent optimization is was done in a separate optimization phase.

5. Storage Assignment:- The purpose of this phase is as follows:-
Assign storage to all variables referenced in source program.
Assign storage to all temporary locations that are necessary for intermediate results.
Assign storage to literals.
Ensure that storage is allocated and appropriate locations are initialized.

6. Code generation:-

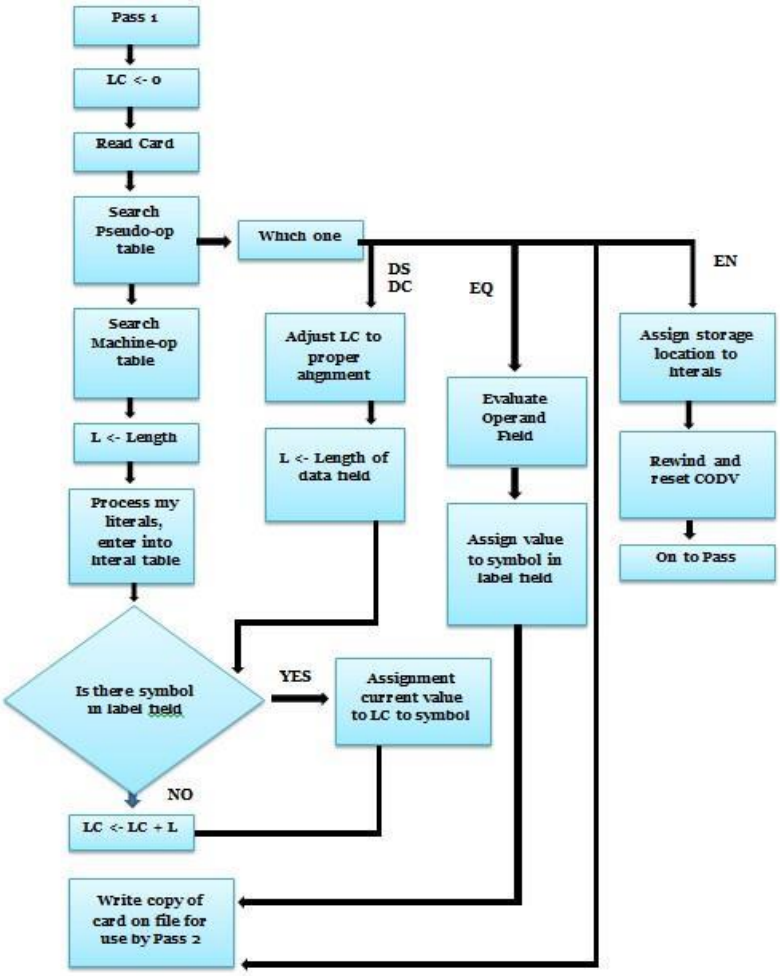
This phase produce a program which can be in Assembly or machine language. This phase has matrix as input. It uses the code production in the matrix to produce code. It also references the identifier table in order to generate address & code conversion.

7. Assembly phase:-The compiler has to generate the machine language code for computer to understand. The task to be done is as follows:-

- Generating code
- Resolving all references.
- Defining all labels.
- Resolving literals and symbolic table.

2. Attempt any TWO of the following : Marks 16

a) Draw the flowchart for pass-I of assembler
(Correct flow chart - 8 marks; flow shall be considered)
Ans:



b) Draw and explain the format of MDT, MNT and ALA with any suitable example.
(MDT - 3 marks, MNT-3 marks, ALA- 2 marks)

Ans:

MDT: - The MDT is a table of text line; if input is from 80 column cards, the MDT can be a table with 80 byte string as entries. Every line of each macro definition except the MACRO line is stored in the MDT. The MEND is kept to indicate end of definition. The macro name line is retained to facilitate keyword argument replacement. Thus for example INCR macro will be stored as

Macro Definition Table

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

MNT: The Macro Name Table serves a function very similar to that of the assembler's Machine-Op Table (MOT) and Pseudo-op Table (POT). Each MNT entry consists of a character string and a pointer to the entry in the MDT that corresponds to the beginning of the macro definition. The MNT entry for the INCR could be.

	8 Bytes	4 Bytes
Index	Name	MDT Index
.	.	.
.	.	.
.	.	.
3	"INCRbbbb"	15

. . .
. . .
. . .

ALA: The argument list array is used during both pass 1 and pass 2 but for somewhat reverse functions. During pass 1, in order to simplify later argument replacement during macro expansion, dummy arguments in the macro definition are replaced with positional indicators hen the definition is stored. The i^{th} dummy argument on the macro name card is represented in the body of the macro by the index marker symbol.

ARGUMENT LIST ARRAY

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

(b gives blank string)

c) Explain the design of direct linking loader with neat diagram.
(Description - 4 marks; diagram - 4 marks; any pass shall be considered)
Ans:
It is a general relocatable loader, and is perhaps the most popular loading scheme presently used.
There are four sections of the object deck for a direct linking loader

- External symbol dictionary card (ESD)
- Text (TXT)
- Relocation and linkage Directory card (RLD)
- End card (END)

The ESD card contains 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 programs are used only by the assembler.

	Type	ID	ADDR	Length
Program name (Segment Definition)	SD	01	Zero	Length of program

Entry (local Definition)	LD	--	Relative address	--
External Reference	ER	Unique number	--	--

(The unique ID numbers are usually assigned sequentially)

ESD Card Format

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, nonrelated data or initial values of address constants.

Column	Contents
1	Hexadecimal byte X'02'
2-4	Characters TXT
5	Blank
6-8	Relative address of first data byte (ADDR)
9-10	Blank
11-12	Byte Count BC = number of bytes of information in cc. 17-22
13-16	Blank
17-72	From 1 to 56 data bytes (instructions and "data" look the same)
73-80	Card Sequence Name

TXT Card Format

The RLD cards contain the following information
The location and length of each address constant that needs to be changed for relocation or linking. The external symbol by which the address constant should be modified.
The operation to be performed.

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters RLD
5-18	Blank
19-20	ID corresponding to a number assigned to SD or ER on ESD card.
21	Flag byte
22-24	Relative address of first byte of address constant. (ADDR)
25-72	Blank
73-80	Card Sequence Name

RLD Card Format

The END card specifies the end of the object deck.

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	Blank
73-80	Card Sequence Name

END Card Format

Advantages of DLL:

The DLL has the advantage of allowing the programmer multiple procedure segments and multiple data segment and of giving him complete freedom in referring data or instruction contained in other segment. This provides flexible intersegment referencing and accessing ability, while at the same time allowing independent translation of programs.

3. Attempt any FOUR of the following: Marks 16

a) Explain four purposes of storage Assignment phase of compiler

(1 mark for each function)

Ans:

The purpose of these phases is to:

1. Assign storage to all variables referenced in the source program
2. Assign storage to all temporary locations that are necessary for intermediate result, e.g. the result of matrix lines. These storage references were reserved by the interpretation phase and dis not appear in the source code.
3. Assign storage to literals.
4. Ensure that the storage is allocated and appropriate locations are initialized (literals and any variables with the initial attribute).

b) Explain conditional macro with an example.
(2 marks for explanation, 2 marks for example)
Ans:

Two important macroprocessor pseudo-ops AIF and AGO permit conditional reordering of the sequence of macro expansion. This allows conditional selection of the machine instructions that appear in expansions of Macro call. Consider the following program. .

```
Loop 1    A1, DATA 1
          A2, DATA 2
          A3, DATA 3
.
.
Loop 2    A1, DATA 3
          A2, DATA 2
.
.
Loop 3    A1, DATA1
.
.
 DATA 1 D C F '5'
 DATA 2 D C F '10'
 DATA 3 D C F '15'
```

In the below example, the operands, labels and the number of instructions generated change in each sequence. The program can written as follows:-

```
MACRO
& ARGO VARY & COUNT, & ARG1, &ARG2, &ARG3
& ARGO   A    1, &ARG1
         AIF (& COUNT EQ1).FINI
         A 2 & ARG2
         AIF (& COUNT EQ2).FINI
         A 3 & ARG3 .FINI
MEND . . .
LOOP1    VARY      3, DATA1, DATA2, DATA3        loop 1    A1, DATA1
 A2, DATA2
 A3, DATA3

LOOP2    VARY      1, DATA1                      loop 2    A1, DATA3
                                                           A2, DATA2
 DATA 1 D C F '5'
DATA 2 D C F '10'
DATA 3 D C F '15'
```

Labels starting with a period (.) such as .FINI are macro labels and do not appear in the output of the macro processor. The statement AIF (& COUNT EQ1) .FINI direct the macro processor to skip to the statement. Labeled .FINI if the parameter corresponding to & COUNT is a1; otherwise the macro processor is to continue with the statement following the AIF pseudo-ops. AIF is conditional branch pseudo ops it performs an arithmetic test and branches only if the tested condition is true. AGO is an unconditional branch pseudo-ops or 'Go to' statement. It specifies label appearing on some other statement. AIF & AGO controls the sequence in which the macro processor expands the statements in macro instructions.

c) Sort the following numbers by applying Radix Exchange Sort 10, 15, 05, 08, 02, 03.

(2 marks for each pass)

Ans:

Pass-1

0	1	2	3	4	5	6	7	8	9
10		02	03		15 05			08	

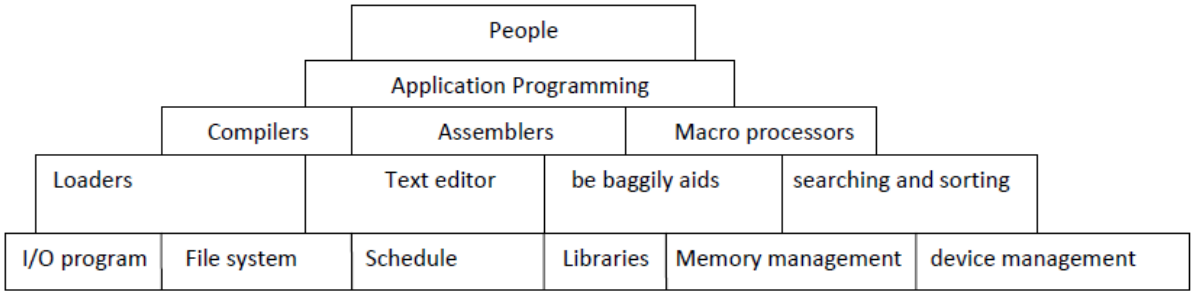
Pass-2

0	1	2	3	4	5	6	7	8	9
08 05 03 02	15 10								

d) Explain foundation of system programming with diagram.

(2 marks – Diagram, 2 marks Explanation)

Ans:



System programs eg. Compiler, loaders, macro processor, operating systems were developed to make computer better adapted to the needs of their users. Compiler is system program that

accept people life languages and translate them into machine language. Loaders are system programs that prepare machine language programs for execution. Macro processors allow programmers to use abbreviations. Operating system and file system allows flexible storing and retrieval of information. The productivity of each computer is heavily dependent upon the effectiveness, efficiency and sophistication of the system programs.

(e) List and give the syntax of database tables used in lexical analysis phase of compiler.
(1 mark for each database table)

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

4. a) Attempt any THREE of the following: Marks 12

(i) Describe top down parsing with technique.

(Recursive Descent Parsing - 3 marks; Backtracking - 3 marks; example - 1 mark each)

Ans:

Top-down Parsing: When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

Recursive descent parsing: It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking. Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Predictive parsers are much faster than backtracking ones Predictive parsers operate in linear time. Backtracking parsers operate in exponential time.

Back tracking: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd

X → oa | ea

Z → ai

For an input string: read, a top-down parser will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

Now the parser matches all the input letters in an ordered manner. The string is accepted.

(ii) Explain Machine dependent optimization of compiler with example.
(Description- 2 marks, example – 2 marks)

Ans:

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 sub expression (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 the first pass and second pass steps of assembler while stating the problem.
(2 marks for pass 1 steps, 2 marks for pass 2 steps)

Ans:

Pass1: Purpose-define symbols and literals

1. Determine length of machine instructions (MOTGETI)
2. Keep track of Location Counter (LC)
3. Remember values of symbols until pass 2 (STSTO)
4. Process some pseudo ops,e.g.,EQU,DS(POTGET1)

5. Remember literals (LITSTO)

Pass2: Purpose –generate object program

1. Look up value of symbols(STGET)
2. Generate instructions(MOTGET2)
3. Generate data (for DS,DC,and,literals)
4. Process pseudo ops(POTGET2)

(iv) Demonstrate the use of databases by assembler passes.
(2 marks for pass 1 databases, 2 marks for pass 2 databases)

Ans:

Databases with their use for pass 1 assembler:

1. **Location counters (LC):-** to keep track of location of each instruction.
2. **Machine Operation Table (MOT):-** that consists symbolic mnemonic for each instruction along with its length.
3. **Pseudo Operation Table (POT):-** that indicates symbolic mnemonic and action to be taken for each pseudo op in pass1
4. **Symbol Table (ST):-** that is used to store each label and its corresponding value.
5. **Literal Table (LT)**:- that is used to store each literal and its corresponding assigned location.

Databases with their use for pass 2 assembler:

1. **Copy file:-** it is prepared by pass 1 to be used by pass 2.
2. **Location counter:-** It is used to assign address to instruction and addressed to symbol defined in the program.
3. **Machine operation Table (MOT) or Mnemonic Opcode Table (MOT):-**

It is used to indicate for each instruction.

a) Symbolic mnemonic

b) Instruction length

c) Binary machine Opcode.

d) Format

4. **Symbol Table (ST):-** It is used to generate the address of the symbol address in the program.
5. **Base table (BT):-** It indicates which registers are currently specified as base register by USING Pseudo-ops.
6. **INST workspace: -** It is used for holding each instruction and its various parts are being getting assembled.

7. PUNCH CARD workspace:- It is used for punching (outputting) the assembled instruction on to cards.
8. PRINT LINE workspace:- It is used for generating a printed assembly listing for programmers reference.
9. Object card:- This card contain the object program in a format required by the loader.

b) Attempt any ONE of the following: Marks 06

(i) Give standard code definition for +, *, -, = and generate the code for the following expression: C = R * (S – F) + 2 * R * (S – F – 100)
(Correct 4 generation – 6 marks; marks shall be awarded for right steps)
Ans:

Standard code definition for -, +,*, =

```
-   L     1, & OPERAND 1
    S     1, & OPERAND 2
    ST    1, M&N
*   L     1, & OPERAND 1
    M     0, & OPERAND 2
    ST    1, M&N
+   L     1, & OPERAND 1
    A     1, & OPERAND 2
    ST    1, M&N
=   L     1, & OPERAND 2
    ST    1, & OPERAND 1
```

By using above standard definition for +,-,*, =, the code will be generated for respective arithmetic statement.

Matrix				Generated Code
1	-	S	F	L 1, S
				S 1, F
				ST 1, M1
2	*	R	M1	L 1, R
				M 0, M1
				ST 1, M2

3	*	2	R	L 1, =F'2' M 0, R ST 1, M3
4	-	S	F	L 1, S S 1, F ST 1, M4
5	-	M4	100	L 1, M4 S 1, =F'100' ST 1, M5
6	*	M3	M5	L 1,M3 M 0, M5 ST 1, M6
7	+	M2	M6	L 1, M2 A 1, M6 1, M7
8	=	M7	C	L 1, M7 ST 1, C

(ii) Define parser. Draw the parse tree for the string 'abccd' using top-down parser.

(Parser - 2 marks; parse tree - 4 marks)

Ans:

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

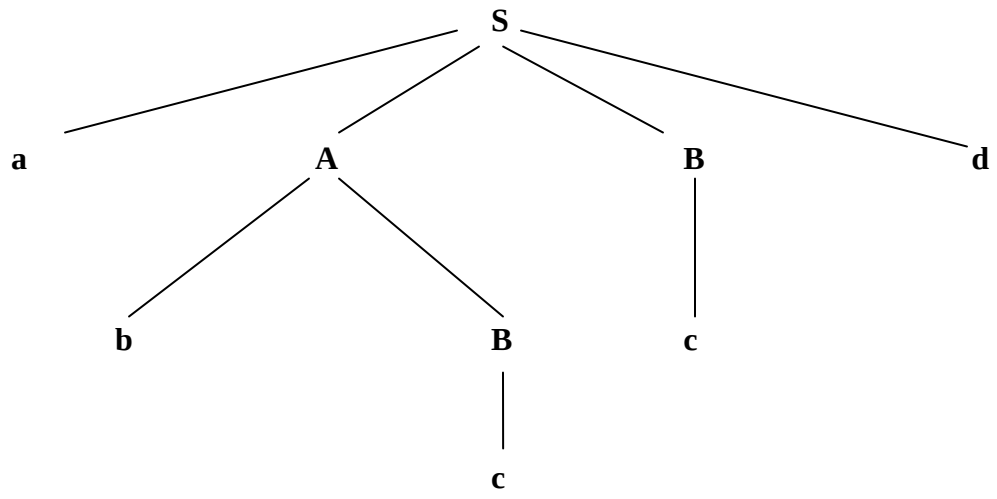
The parse tree for the string 'abccd'

Assume following grammar for the given string "**abccd**"

S → aABd

A → b | Bc

B → c



5. Attempt any TWO of the following: Marks 16

a) Describe the term overlay structure and dynamic binder.
(Overlay structure - 4 marks; dynamic binder - 4 marks)

Ans:

Overlays and Dynamic Binders: Sometimes a program may require more storage space than the available one Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. In such situations only those segments are resident in the memory that area actually needed at the time of execution.

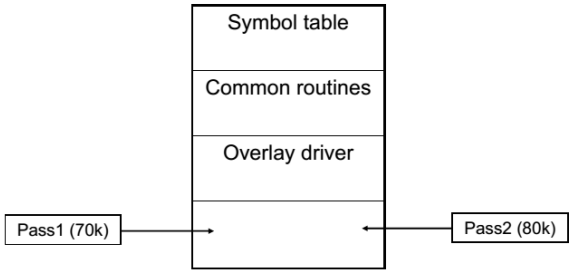
Dynamic Linking:

1. Linking postponed until execution time.
2. Small piece of code, stub, used to locate the appropriate memory-resident library routine.
3. Stub replaces itself with the address of the routine, and executes the routine.
4. Operating system needed to check if routine is in processes, for memory address.
5. Dynamic linking is particularly useful for libraries.

Overlays:

1. Keep in memory only those instructions and data that are needed at any given time.
2. Needed when process is larger than amount of memory allocated to it.
3. Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.
4. The entire program and data of a process must be in the physical memory for the process to execute.
5. The size of a process is limited to the size of physical memory.
6. Overlays are implemented by user, no special support needed from operating system, programming design of overlay structure is complex.

Overlay A needs 130k and Overlay B needs 140k.



b) Describe the syntax analysis phase of compiler and outline the algorithm for syntax analysis phase.

(Syntax analysis phase - 4 marks; Algorithm - 4 marks)

Ans:

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.

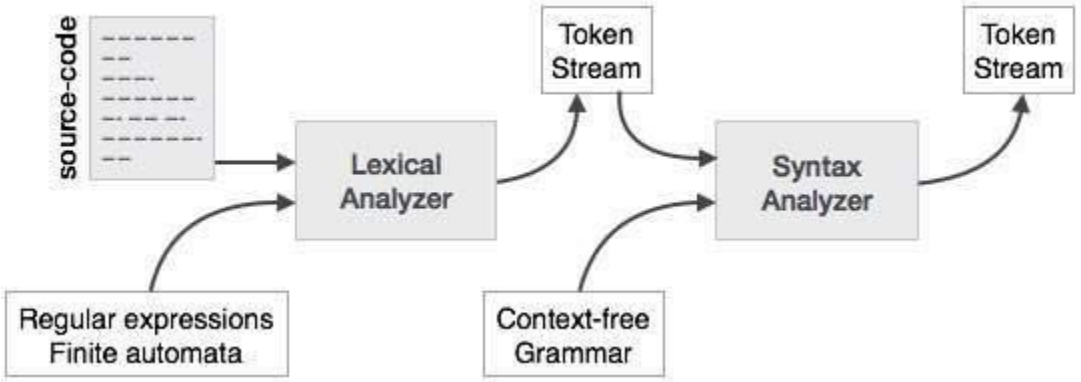
Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the

limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.

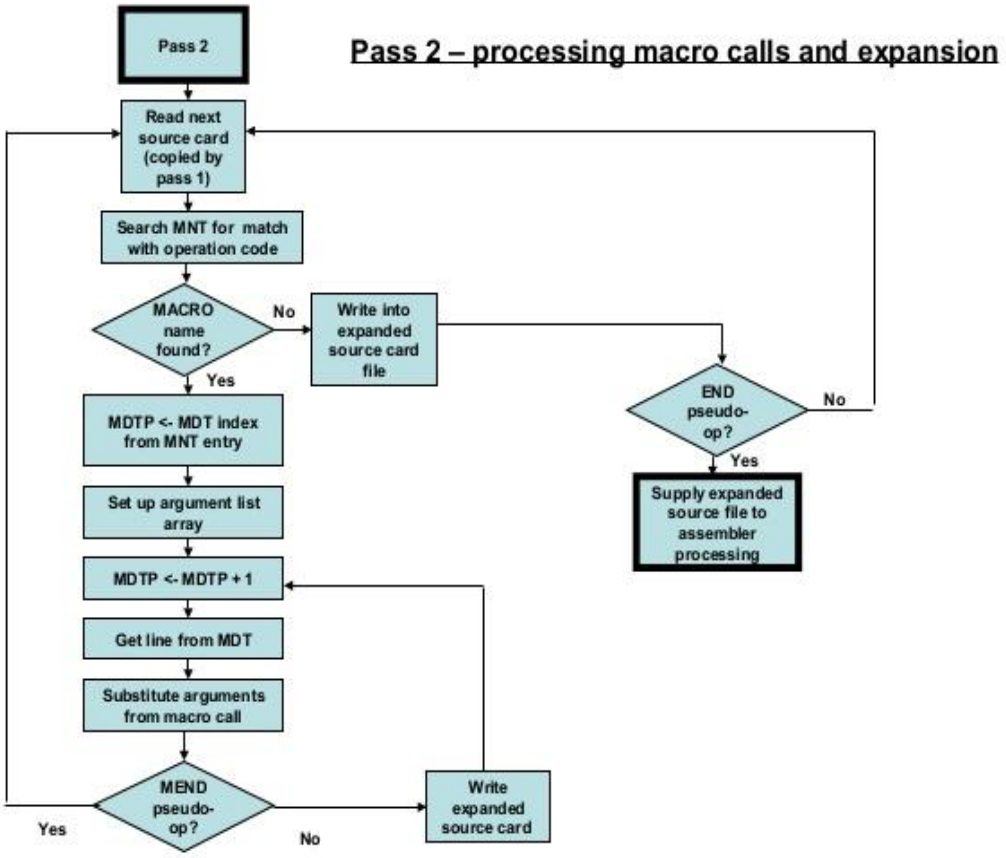


This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

1. Syntax analysis imposes a hierarchical structure on the token stream. This hierarchical structure is called syntax tree.
2. A syntax tree has an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children.
3. A leaf is a record with two or more fields, one to identify the token at the leaf, and the other to record information about the token.

c) Draw the flowchart for processing macro calls and expansion in two passes macro - processor.

(Correct flow chart - 8 marks; flow shall be considered)



6. Attempt any FOUR of the following: Marks 16

a) Describe elimination of common sub expression technique of Optimization phase of compiler with suitable example.

(Definition - 2 marks, Example - 2 marks)

Ans:

Common subexpression elimination (CSE) is a compiler optimization technique of finding redundant expression evaluations, and replacing them with a single computation. This saves the time overhead resulted by evaluating the expression for more than once. Example

```
main(){
int i, j, k, r;
```

```
scanf("%d%d", &i, &j);
k = i + j + 10;
r = i + j + 30;
printf("%d %d\n", k, r);
}
```

The above ststement can be eliminated by writing x = i+ j and in the statements replaceing as

```
main(){
int i, j, k, r, x;
scanf("%d%d", &i, &j);
x = i +j;

k = x + 10;
r = x + 30;
printf("%d %d\n", k, r);
}
```

b) Describe "compile – and – go" loader with neat diagram.

(Description - 2 marks; Diagram - 2 marks)

Ans:

"Compile and go" loader: in this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter.

Advantages:

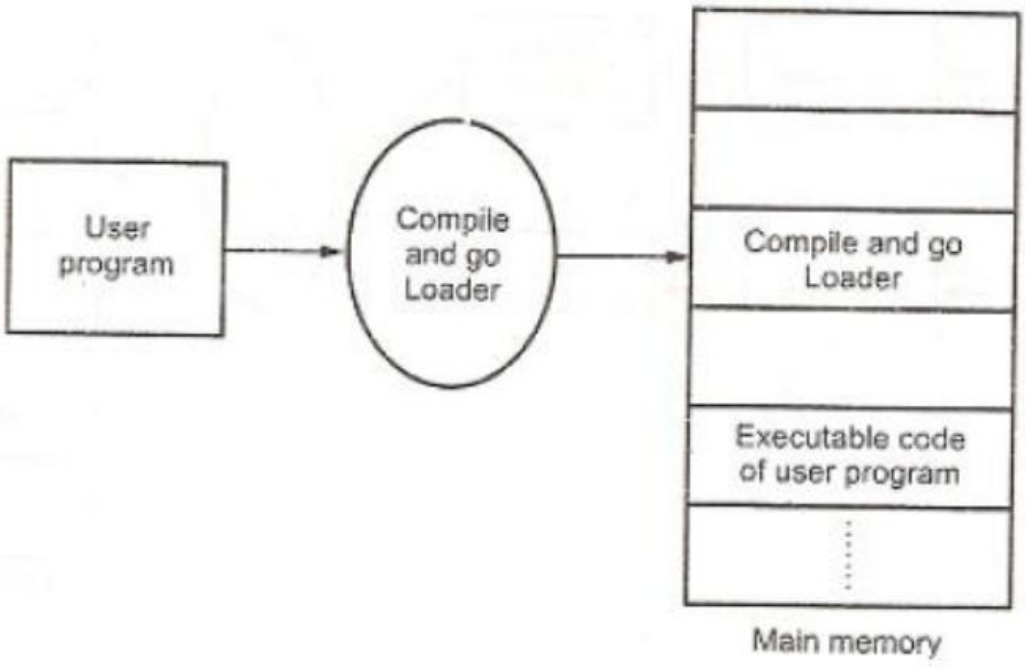
• This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.

Disadvantages:

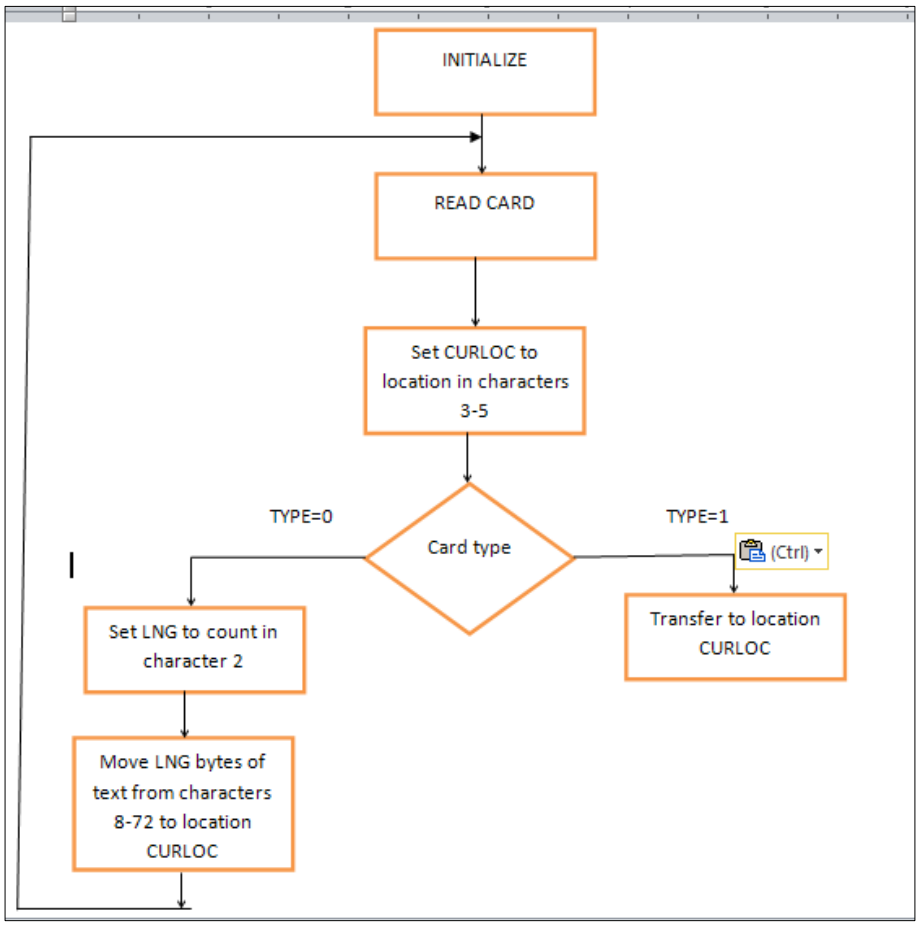
• In this scheme some portion of memory is occupied by assembler which is simply a wastage of memory. As this scheme is combination of assembler and loader activities, this combination program occupies large block of memory.

There is no production of .obj file; the source code is directly converted to executable form. Hence even though there is no modification in the source program it needs to be assembled

and executed each time, which then becomes a time consuming activity.
• It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.
• For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the "compile and go" loader cannot handle such programs.
• The execution time will be more in this scheme as every time program is assembled and then executed.



c) Draw the flowchart of Absolute loader.
(Any Diagram - 4 marks)
Ans:



d) Consider the following assembly language program. Shows pass l and also show the entries of symbol table.

```
John:  START 0
USING *, 15
L  1,  FIVE
A  1,  FOUR
ST  1,  TEMP
FOUR  DC  F  '4'
FIVE   DC  F  '5'
TEMP  DS  1F
END
```

(First pass - 2 marks, entries in symbol table - 2 marks)

Ans:

Source program			First pass	
			Relative address	Mnemonic instruction
JOHN	START	0		
	USING	*,15		
	L	1,FIVE	0	L 1,-(0,15)
	A	1,FOUR	4	A 1,-(0,15)
	ST	1,TEMP	8	ST 1,-(0,15)
FOUR	DC	F'4'	12	4
FIVE	DC	F'5'	16	5
TEMP	DS	1F	20	-
	END			

← 14-bytes per entry →

Symbol (8bytes)(characters)	Value (4bytes)(hexadecimal)	Length (1-byte)(hexadecimal)	Relocation (1byte)(character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	0000	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMbbbb"	0014	04	"R"

Symbol Table(ST)

e) Explain the use of ESD, TXT, RLD and END card used in loader by giving example. ***(1 mark each)***

Ans:

There are four sections of the object deck for a direct linking loader 1. External symbol

• **The ESD card** contains the information necessary to build the external symbol. The external symbols are symbols that can be referred beyond the subroutine level. The normal labels in the source program are used only by the assembler. **He ESD card** contains the information necessary to build the external symbol. The external symbols are symbols that can be referred beyond the subroutine level. The normal labels 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