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

SUMMER-15 EXAMINATION
Model Answer

Subject Code: 17517 **Subject Name: System Programming**

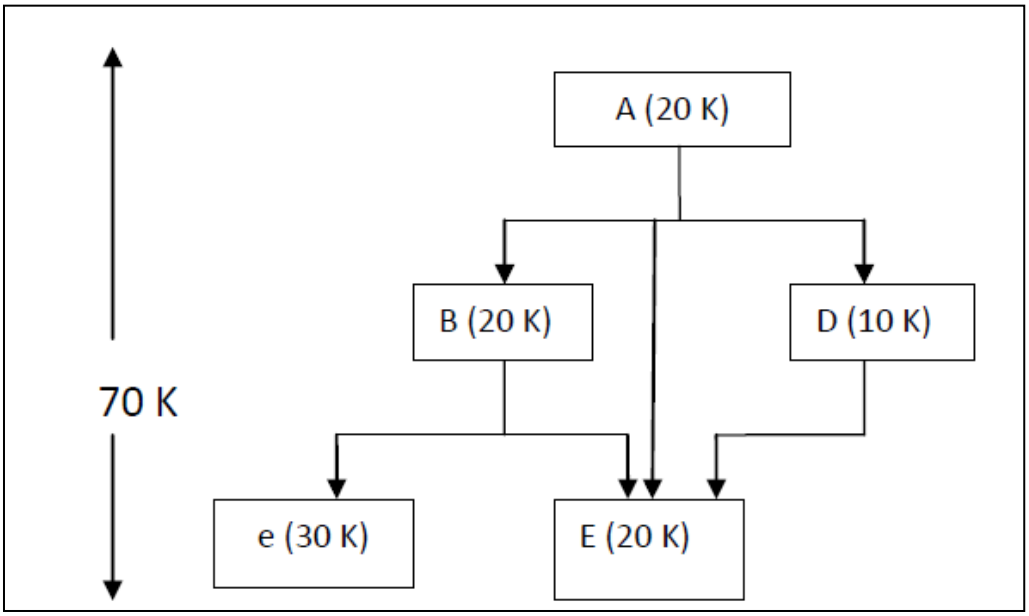
Important Instructions to examiners:
1) The answers should be examined by key words and not as word-to-word as given in themodel answer scheme.
2) The model answer and the answer written by candidate may vary but the examiner may try to assess the understanding level of the candidate.
3) The language 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: **MARKS**

1) Explain overlay structure used in dynamic loading scheme. **(4x3=12)**
(Diagram- 1Mark, Explanation- 3Marks)

Ans:

The subroutines of a program are needed at different times. For e.g. Pass 1 and pass 2 of an assembler are call other subroutine it is possible to produce an overlay structure that identifiers mutually exclusive subroutines. In order for the overlay structure to work it is necessary for the module loader to the various procedures as they are needed. The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is collect the overlay supervision or simply the upper.



Above program consisting of five subprogram (A, B, C, D & E) that require look bytes of core. The arrow indicate that subprogram A only calls B, D and E; subprogram B only calls C and E; subprogram D only calls E; and subprogram C and E do not call any other routine procedures B and D are never in use at the same time; neither are C and E. If are load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure. This happens to be 70k. Overlay reduces the memory requirement of a program. It also makes it possible to execute program where size exceeds the amount of memory which cane ne allocated to them. For the execution of overlay structured program, the root is loaded in memory and given control for the execution. Other overlays are loaded as and when headed. Loading of an overlay overwrite a previously loaded overlay with the same load origin.

2) List components of system software and explain any two of them.
(List-2Marks, Explanation- 1Mark each)

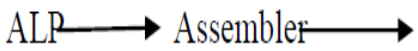
Ans:

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

Components of System software:

1. Assembler:

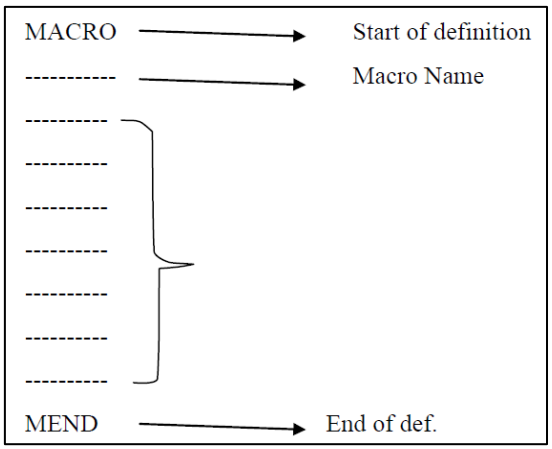
Assembler is a language translator that takes as input assembly language program (ALP) and generates its machine language equivalent along with information required by the loader.



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.



3. Loader: It is responsible for loading program into the memory, prepare them for execution and then execute them.

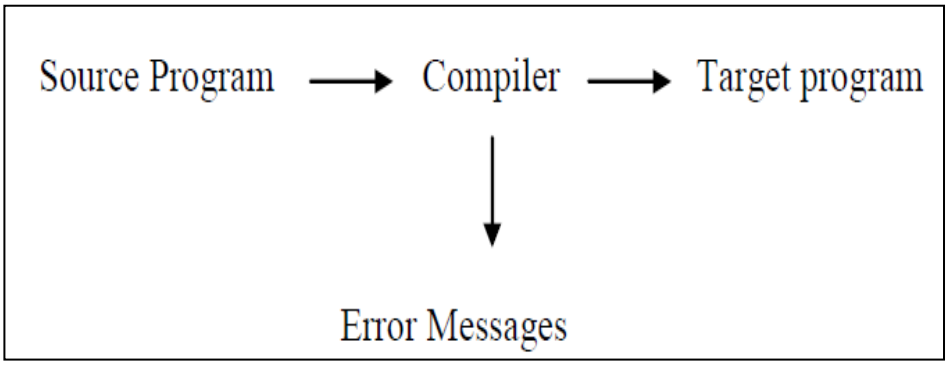
OR Loader is a system program which is responsible for preparing the object programs for execution and start the execution. Functions of loader

a. Allocation
b. Linking
c. Relocation
d. Loading

4. 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.

5. 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.

6. 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. (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

3) Explain the working of Bucket sort.
(Explanation-2Marks, Example- 2Marks)

Ans:

The sort involves examine the least significant digit of the keyword first, and the item is then assigned to a bucket uniquely depend on the value of the digit. After all items have been distributed the buckets items are merged in order and then the process is repeated until no more digits are left. A number system of base P requires P buckets. There are serious disadvantages to using it internally on a digital compiler

1) It takes two separate processes, a separation and a merge
2) It requires a lot of extra Storage for the buckets.

The average time required for the sort is (* N *log P(K)) where N is the table size, K is the maximum key size & P is the radix of the radix sort. The extra storage required is N *P.

Original table	First Distribution	Merge	First Distribution	Final merge
19	01		01	
13	0)	31	0) 01, 02, 05, 09	02
05	1) 01, 31, 11, 21	11	1) 11, 13, 16, 19	05
27	2) 02	21	2) 21, 26, 27	09
01	3) 13	02	3) 31	11
26	4)	13	4)	13
31	5) 05	05	5)	16
16	6) 26, 16	26	6)	19
02	7) 27	16	7)	21
09	8)	27	8)	26
11	9) 19, 09	19	9)	27
21	09		31	
↑		↑		
Separate, based on last digit		Separate, based on first digit		

4) List and give syntax of database tables used in lexical analysis phase of compiler.
(List- 1Mark, Syntax- 3Marks)

Ans:

Lexical Phase:-

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

b) Attempt any one: **MARKS**

1) Explain evolution of system software. **(6x1=6)**
(Explanation- 6Marks)

Ans:

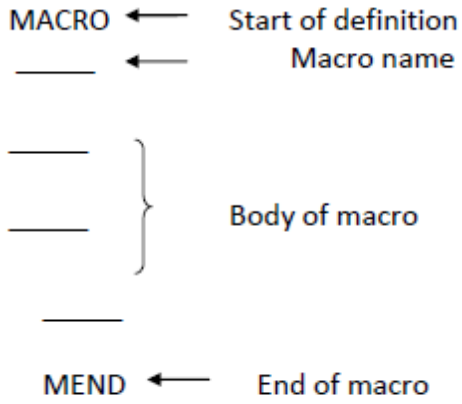
Evolution of System Software

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.

2) What is Macro Language? Explain conditional macro with an example.
(Definition- 2Marks, Explanation- 2Marks for AIF; 2 Marks AGO)

Ans:

Macro language is a language which is understand by macroprocessor. Macro requires an overhead of macroprocessor. Macro language is always enclosed in MACRO & MEND pseudo-opcode. Macro is a single line abbreviation for group of instructions. The template for defining a macro is as follows:-



Conditional Macros
Two important microprocessor 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 macroprocessor. The statement AIF (& COUNT EQ1) .FINI direct the macroprocessor to skip to the statement. Labeled .FINI if the parameter corresponding to & COUNT is a1; otherwise the macroprocessor is to continue with the statement following the AIF pseudo-ops. AIF is a 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 control the sequence in which the macroprocessor expands the statements in macro instructions.

2. Attempt any two: **MARKS**

1) Explain different data structure used by Phase-II assembler. **(8x2=16)**
(Any 8- 1Mark each)

Ans:

The various data structure used is as follows:-

i) Copy file:- It is prepared by pass 1 to be used by pass 2.

ii) Location counter:- It is used to assign address to instruction and addressed to symbol defined in the program.

iii) Machine operation Table (MOT) or Mnemonic Opcode Table (MOT):- It is used to indicate for each instruction.

a) Symbolic mnemonic
b) Instruction length
c) Binary machine opcode.
d) Format.

iv) Pseudo-operation Table (POT):- It indicate for each pseudo-op the symbolic mnemonic and action to be taken in pass 2

Or

It is consulted to process pseudo like DS, DC, Drop & using.

v) Symbol Table (ST):- It is used to generate the address of the symbol address in the program.

vi) Base table (BT):- It indicate which registers are currently specified as base register by USING Pseudo-ops.

vii) INST workspace:- It is used for holding each instruction and its various parts are being getting assembled.

viii) PUNCH CARD workspace:- It is used for punching (outputting) the assembled instruction on to cards

ix) PRINT LINE workspace:- It is used for generating a printed assembly listing for programmers reference.

x) Object card:- This card contain the object program in a format required by the loader.

2) Explain Binary Search with suitable example.
(Explanation- 4Marks, Example- 4Marks)

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.

3) Explain code optimization phase of compiler.
(Explanation 4 Marks; Machine Dependent- 2Marks; Machine Independent 2 Marks)

Ans:

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

Machine- 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					

3. Attempt any four: **MARKS**

1) Explain operating system as "Manager of system". **(4x4=16)**
(Explanation- 4 Marks)

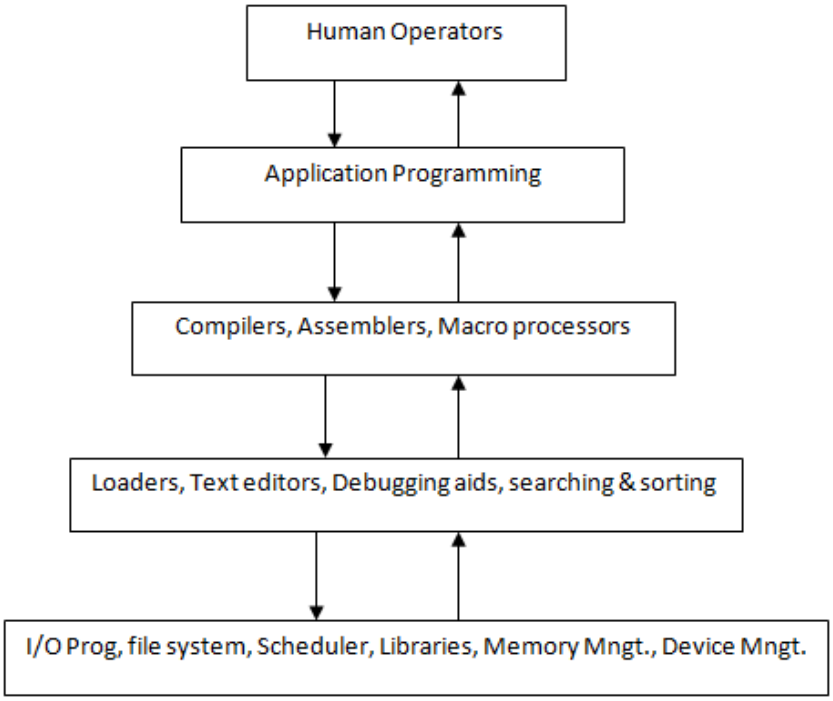
Ans:

Operating system provides an orderly and controlled allocation of the processors, memories and I/O devices. When a computer has multiple users the need for managing and protecting the memory, I/O devices and other devices is greater. Thus the primary task of OS is to keep track of who is using which resource, to grant resource requests, to mediate conflicting requests from different programs etc. Resource management includes multiplexing resources in two ways - "in time" and "in space".

(i)When a resource is time multiplexed different programs or different users gets their turn to use that resource. E.g.: Printer.

(ii)When a resource is space multiplexed instead of taking turns, the resource is shared among them, i.e. each one gets a part of the resource. E.g.: Sharing main memory, hard disk etc.

Following diagram illustrates operating system as system manager.



2) Explain general design of the assembler.
(4 Marks for all steps)

Ans:

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

3) Explain with flowchart of over view of passes of compiler.
(2 Mark for diagram, 2 Marks for brief explanation of each phase)

Ans:

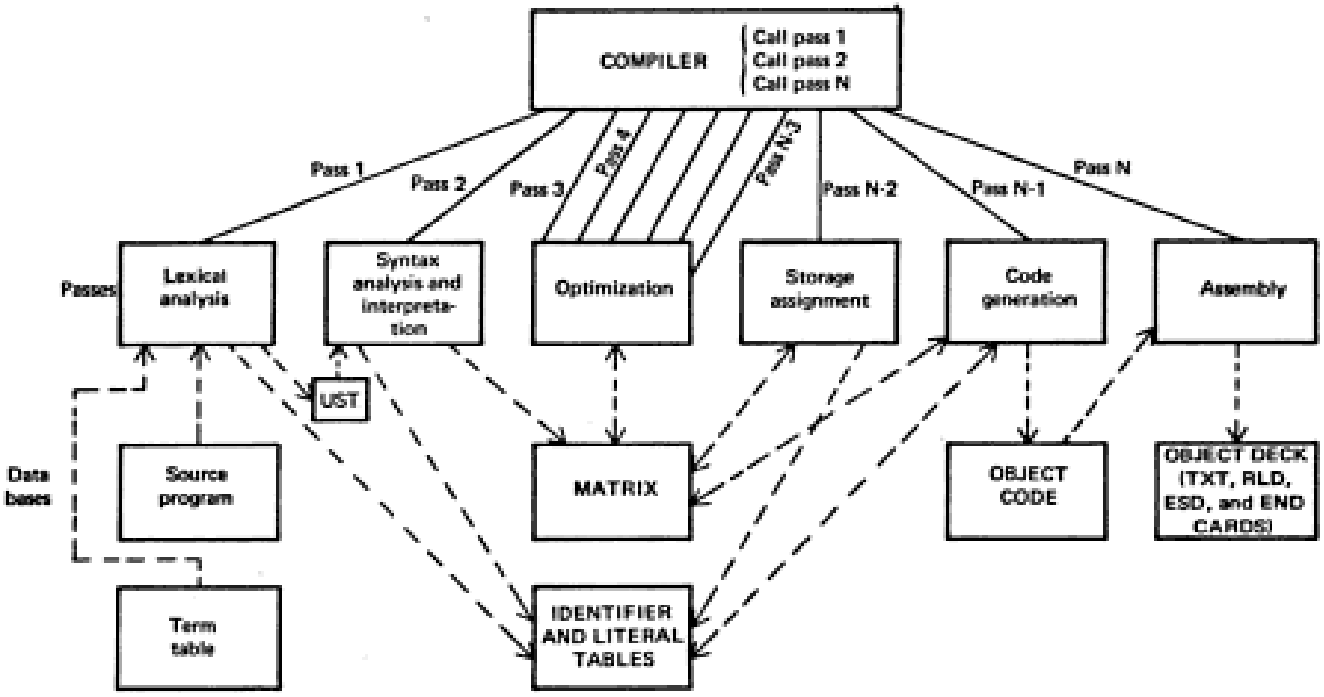


FIGURE 8.29 Passes of a compiler

Passes is an 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 require 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.

4) Explain databases used in direct linking loader system software.
(2 Marks for databases used in pass 1, 2 Marks for pass2)

Ans:

Databases required for Pass 1 and Pass 2 of direct linking loader with their purposes listed below:

Pass 1

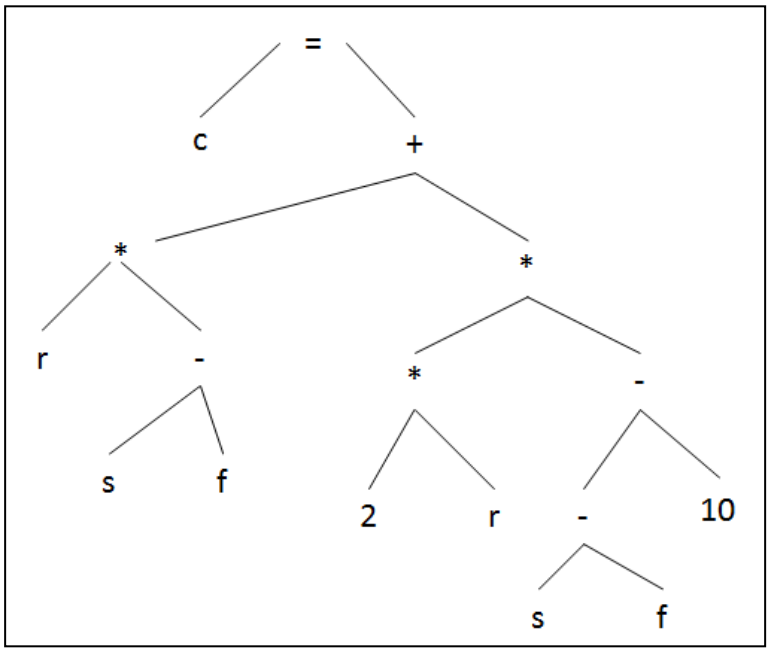
1. Input object decks
2. A parameter, the Initial Program Load Address (IPLA) supplied by the programmer or the operating system that specifies the address to load the first segment.
3. A Program Load Address (PLA) counter, used to keep track of each segments assigned location.
4. A table, the Global External Symbol Table (GEST), that is used to store each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2. This may be stored on an auxiliary storage device, such as magnetic tape, disks or drum, or the original objects deck may be reread by the loader a second time for pass 2.
6. A printed listing, the load map, that specifies each external symbol and its assigned value.

Pass 2

1. Copy of object program inputted to pass1
2. The Initial Program Load Address parameter (IPLA)
3. The Program Load Address Counter (PLA)
4. The Global External Symbol Table (GEST), prepared by pass1, containing each external symbol and its corresponding absolute address value.
5. An array, the Local External Symbol array (LESA), which is used to establish a correspondence between the ESD ID numbers, used on ESD and RLD cards, and the corresponding external symbols' absolute address value.

5) Convert following expression into parse free using top down parsing
c= r*(s-f) +2 *(s-f-10).
(4 Marks for correct diagram)

Ans:



4. a) Attempt any three: **MARKS**

1) Explain binders in detail. **(3x4=12)**
(2 Marks for static binders, 2 Marks for dynamic binders)

Ans :

Static binders:

- In static binders a specific core allocation of a program is performed at the time that the subroutines are bound together.
- Since this kind of module looks like an actual 'snapshot' or 'image' of a section of core, it is also called as 'core image module' and the corresponding binder is also called a core image builder.
- It does not keep track of relocation information
- The module loader performs allocation and loading.
- Relatively simple and fast

Dynamic binders:

- The dynamic binder is 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.

In both cases, a program that is to be used repeatedly need only be bound once and then can be loaded whenever required.

2) Explain in detail machine dependent optimization.
(2 Marks explanation, 2 Marks example)

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 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					

3) Explain compile time compute optimization with example.
(2 Marks for explanation, 2 Marks for example)

Ans:

Compile time compute means shifting of computations from run time to compilation time. Doing computation that includes constants at compile time save both space and execution time for the object program.

The algorithm for this optimization is as follows:-

i) Scan the matrix.

ii) Look for operators, both of whose operands were literals.

iii) When it found such an operation it would evaluate it, create new literal, delete old line

iv) Replace all references to it with the uniform symbol for the new literal.

v) Continue scanning the matrix for more possible computation.

For e.g.- A = 2 * 276 / 92 * B

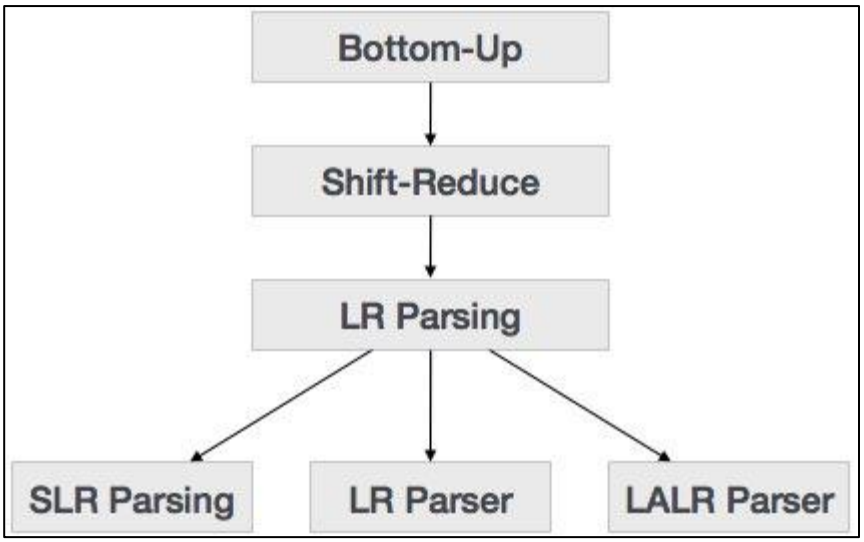
The compile time computation would be

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

4) Explain the concept of bottom up parser.
(2 Marks for basic and one mark each for types of bottom up parser)

Ans:

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing
Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step** : When the parser finds a complete grammar rule RHS and replaces it to LHS, it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR1 – Simple LR Parser:
    - Works on smallest class of grammar
    - Few number of states, hence very small table
    - Simple and fast construction
- LR1 – LR Parser:
    - Works on complete set of LR1 Grammar
    - Generates large table and large number of states
    - Slow construction
- LALR1 – Look-Ahead LR Parser:
    - Works on intermediate size of grammar
    - Number of states are same as in SLR1

b) Attempt **any one:** **MARKS**

1) State and explain four basic task of macro processor. **(6x1=6)**
(List -2 Marks, explanation of each-1 Mark)

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	"INCRbbbb"	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	"bbbbbbbb" (all Blank)
1	"Data3bbb"
2	"Data2bbb"
3	"Data1bbb"

4. **Replacing macro definitions with macro calls:** The pre-processor 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.

2) Explain the difference between top down and bottom up parser.
(6 Marks for any 6 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

5. Attempt any two: **MARKS**

1) Explain Dynamic Binders. **(8x2=16)**
(Explanation-4Marks, Advantages-2 Marks, disadvantages-2 Marks)

Ans:

In dynamic linking, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external ·reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is resumed. Thus dynamic linking both the loading and linking is done dynamically.

Advantages

1. The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.
2. The system can be dynamically reconfigured.

Disadvantages

The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory

2) Explain four purposes of storage assignment phase of compiler.
(Each purpose - 2 Marks)

Ans:

The purpose of this phase is to:

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

The storage allocation phase first scans through the identifier table, assigning locations to each entry with a storage class of static. It uses a location counter, initialized at zero, to keep track of how much storage it has assigned. Whenever it finds a static variable in the scan, the storage allocation phase does the following four steps:

1. Updates the location counter with any necessary boundary alignment.
2. Assigns the current value of the location counter to the address field of the variable.
3. Calculate the length of the storage needed by the variable (by examining its attributes).
4. Updates the location counter by adding this length to it.

Once it has assigned relative address to all identifiers requiring STATIC storage locations, this phase creates a matrix entry:

STATIC	Size	

This allows code generation to generate the proper amount of storage. For each variable that requires initialization, the storage allocation phase generates a matrix entry:

Initialize	Variable	

This tells code generation to put into the proper storage location the initial values that the action routines saved in the identifier table.
A similar scan of the identifier table is made for automatic storage and controlled storage. The scan enters relative location for each entry. An “automatic” entry and a “controlled” entry are also made in the matrix. Code generation use the relative location entry to generate the address part of instructions. No storage is generated at compile time for automatic or controlled. However, the matrix entry automatic does cause code to be generated that allocates this storage at execution time, i.e., when the generated code is executed, it allocates automatic storage.

LIT	Size	

The literal table is similarly scanned and location are assigned to each literal, and a matrix entry is made. Code generation generates storage for all literals in the static area and initializes the storage with the values of the literals.
Temporary storage is handled differently since each source statement may reuse the temporary storage (intermediate matrix result area) of the previous source statement. A computation is made of the temporary storage that is required for each source statement. The statement required the greatest amount of temporary storage determines the amount that will be required for the entire program. A matrix entry is made of the form
This enables the code

AUTOMATIC	Size	

generation phase to generate code to create the proper amount of storage. Temporary storage is automatic since it is only referenced by the source program and only needed while the source program is active.

3) Explain Radix Sort with example.
(Explanation- 4 Marks , for example -4 Marks)

Ans:

A considerable better distributive sort tis the radix exchange sort which is applicable when the keys are expressed or are expressible in binary. Sorting is accomplished by considering groups with the same (M) first bits and ordering that group with respect to the (M+1) st bit. The ordering of a group on a given bit is accomplished by scanning down from the top of the group for a one bit and up from the bottom for a zero bit: these two are exchanged and the sort continues. This algorithm required the program to keep up with a large number of groups and, coded in a bad form, could require an additional table N long. However, with optimal coding it is possible to keep track of the groups by simply monitoring the top of the table and a list of break points, one for each it of the keyword. (Thus with 32 bit words a table of only 33 entries required.) an example of the radix exchange sort is shown. It is a rather complicated example and somewhat difficult to understand- qualities characteristic of most distributive sorts. If the sort algorithm is programmed to quit sorting when a group contains only one item, then the time required for the radix exchange sort is proportional to N*log(N) as compared to $N*\log_p(K)$ for the bucket sort (here K is the maximum key size and p is the radix). Note that the radix exchange sort does not require extra table space for "buckets."

The sort involves examine the least significant digit of the keyword first, and the item is then assigned to a bucket uniquely depend on the value of the digit. After all items have been is distributed the buckets items are merged in order and then the process is repeated until no more digits are left. A number system of base P requires P buckets. There are serious disadvantages to using it internally on a digital compiler 1) It takes two separate processes, a separation and a merge 2) It requires a lot of extra Storage for the buckets. The average time required for the sort is (*N*log P(K)) where N is the table size, K is the maximum key size & P is the radix of the radix sort. The extra storage required is N * P.

Original table	First Distribution	Merge	First Distribution	Final merge
19	01		01	
13	0)	31	0) 01, 02, 05, 09	02
05	1) 01, 31, 11, 21	11	1) 11, 13, 16, 19	05
27	2) 02	21	2) 21, 26, 27	09
01	3) 13	02	3) 31	11
26	4)	13	4)	13
31	5) 05	05	5)	16
16	6) 26, 16	26	6)	19
02	7) 27	16	7)	21
09	8)	27	8)	26
11	9) 19, 09	19	9)	27
21	09		31	
↑		↑		
Separate, based on last digit		Separate, based on first digit		

6. Attempt any four: **MARKS**

1) State and explain task of macro processor. **(4x4=16)**
(Each task-1 Mark)

Ans:

1. **Recognize Macro Definitions.**
   A Macro instruction processor must recognize macro definitions identified by the MACRO and MEND pseudo ops. This task can be complicate when macro definitions appear within macros. 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. All of the intervening text, including nested MACROs and MENDs defines a single macro instruction/
2. **Save the Macro Definitions**
   The processor must store the macro instruction definitions, which it will need for expanding macro calls.
3. **Recognize macro calls**
   The processor must recognize macro calls that appear as operations mnemonics. His suggest that macro names be handled as a type op-code.
4. **Expand Macro calls and substitute arguments.**

The processor must substitute for dummy or macro definitions arguments the corresponding arguments from a macro call; the resulting symbolic text is then substitute for the macro call. This text of course, may contains additional macro definitions or calls.

2) Explain use of following instructions:
1) USING 2) START
3) DC 4) DS
(Use of each instruction- 1 Mark)

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 tell the assembler where the beginning of the program is an allows the user to give a name to the program .

c. DC
It is a pseudo op that is used to define a constant, which places constant value as fullwords in the memory.

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

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

Ans:

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

4) Describe uniform symbol table and explain process of tokenising with example.
(2 Marks for uniform symbol table, 2 Marks for tokenising)

Ans:

Uniform symbol table is created by lexical analysis phase of compiler. It contains the source program in the form of uniform symbols. It is used by syntax and interpretation phases as the source of input to the stack. Each symbol from the uniform symbol table enters the stack only once.

TABLE	INDEX

UST ENTRY

Process of tokenising
A **token** is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called **tokenization**. Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input
Example:

Consider this expression in the C programming language:

sum = 3 + 2;

Tokenized and represented by the following table:

Lexeme	Token type
Sum	Identifier
=	Assignment operator
3	Integer literal
+	Addition operator
2	Integer literal
;	End of statement

` **5) What is loader? How it works? Explain with diagram.**
(1 Marks for Loader Definitio, 2 Marks for Working of Loader, 1 Mark for Diagram)

Ans:

Loader is a program which accepts the object program decks, prepares these programs for execution by the computer, and initiates the executions.

It allocates space in memory for the program. Then resolve symbolic references between object decks. It adjust all address dependent locations, such as address constants, to correspond to the allocated space. Physically place the machine instructions and data into memory.