CMR INSTITUTE OF TECHNOLOGY	USN	INSTITUTE OF CMR TECHNOLOGY

Internal Assesment Test - III

Subject : System Software				Code : 16MCA25
Date : 2/5/2017	Duration : 90 mins	Max Marks : 50	Sem : II	Branch : MCA

	Answer Any FIVE FULL Questions	Marks	OBE CO	OBE RBT
1(a)	**Differentiate between application software and system software. Give examples for each.**	[2]	CO1	L2

System Software	Application Software
Intended to support the operation and use of the computer	An application program is primarily concerned with the solution of some problem, using the computer as tool
Focus is on the Computer system and not on the application	The focus is on the application not on the computing system.
It depends on the structure of the machine on which it is executed.	It does not depend on the structure of the machine it works
Ex. Operating system, Loader, Linkers, assembler, compiler, text editors etc.	Ex. Banking system, Inventory system.

		Marks	CO	RBT
(b)	**Describe SIC/XE Architecture with suitable Examples.**	[8]	CO1	L1

1) Memory

- Memory consists of 8-bit bytes.
- 3 consecutive bytes form a word (24 bits).
- Maximum memory available on a SIC/XE system is 1 megabyte (2^{20} bytes).
-

2) Registers

Five registers of SIC machine remains same in SIC/XE. The additional registers provided by SIC/XE are as follows.

Mnemonic	Number	Use
B	3	Base register; used for addressing.
S	4	General working register – no special use.
T	5	General working register – no special use.
F	6	Floating-point accumulator (48 bits).

3) Data Formats

- SIC/XE provides the same data formats as the standard version.
- In addition there is a 48 bit floating point data type with following format.

1	11	36
s	exponent	fraction

4) Instruction Formats

1. Format 1 (1 byte)

8
op

2. Format 2 (2 bytes)

8	4	4
op	r1	r2

3. Format 3 (3 bytes)

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

4. Format 4 (4 bytes)

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	address

5) Addressing Modes

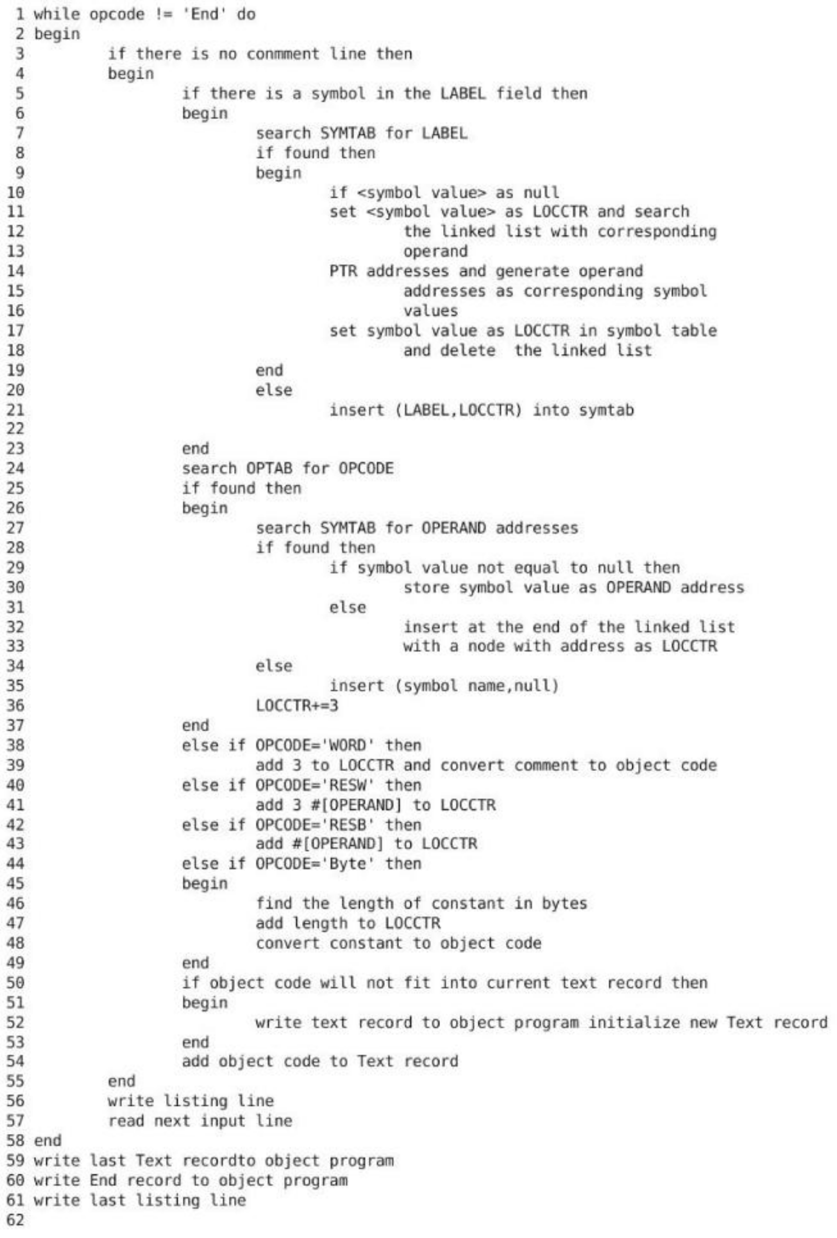
There are two addressing modes, indicated by the setting of the x bit in the instruction.

Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (x)

6) Instruction Set

SIC provides a basic set of instructions that are sufficient for most simple task.

i) Data transfer instruction: This include instructions that load and

	store registers . Eg. LDA, LDX, STA, STX. ii) Arithmetic operation instruction: Basic arithmetic operations that involves register A Eg. ADD, SUB, MUL, DIV, COMP. iii) Conditional Branching: Conditional jump instructions test the settings of conditional code and jump accordingly. Eg. JLT, JGT, JEQ. iv) Subroutine call Instructions: Perform subroutine linkage. Eg. JSUB, RSUB. Return address is stored in linkage(L) register. 7) Input and Output • Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator). • Each device is assigned a unique 8bit code. • There are 3 I/O instructions.			
2	**Write an algorithm for one pass Assembler.** 	[10]	CO2	L2

```
while opcode != 'End' do
begin
    if there is no conmment line then
    begin
        if there is a symbol in the LABEL field then
        begin
            search SYMTAB for LABEL
            if found then
            begin
                if <symbol value> as null
                set <symbol value> as LOCCTR and search
                    the linked list with corresponding
                    operand
                PTR addresses and generate operand
                    addresses as corresponding symbol
                    values
                set symbol value as LOCCTR in symbol table
                    and delete  the linked list
            end
            else
                insert (LABEL,LOCCTR) into symtab

        end
        search OPTAB for OPCODE
        if found then
        begin
            search SYMTAB for OPERAND addresses
            if found then
                if symbol value not equal to null then
                    store symbol value as OPERAND address
                else
                    insert at the end of the linked list
                    with a node with address as LOCCTR
            else
                insert (symbol name,null)
            LOCCTR+=3
        end
        else if OPCODE='WORD' then
            add 3 to LOCCTR and convert comment to object code
        else if OPCODE='RESW' then
            add 3 #[OPERAND] to LOCCTR
        else if OPCODE='RESB' then
            add #[OPERAND] to LOCCTR
        else if OPCODE='Byte' then
        begin
            find the length of constant in bytes
            add length to LOCCTR
            convert constant to object code
        end
        if object code will not fit into current text record then
        begin
            write text record to object program initialize new Text record
        end
        add object code to Text record
    end
    write listing line
    read next input line
end
write last Text recordto object program
write End record to object program
write last listing line
```

3(a) **Define Program Relocation? How relocation is achieved using Modification Record?** [6] CO2 L1

It is often desirable to have more than one program at a time sharing the memory and other resources of the machine.

In such a situation the actual starting address of the program is not known until the load time.

Program in which the address is mentioned during assembling itself. This is called *Absolute Assembly or Absolute Program.*

Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used by the program. However, the assembler identifies for the loader those parts of the program which need modification.

An object program that has the information necessary to perform this kind of modification is called the relocatable program.

This can be accomplished with a Modification record having following format:

Modification record

Col. 1	M
Col. 2-7	Starting location of the address field to be modified, relative to the beginning of the program (Hex)
Col. 8-9	Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified The length is stored in half-bytes. The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

(b) **Write a assembly language program in SIC/XE to add 2 arrays of 200 integers** [4] CO1 L3

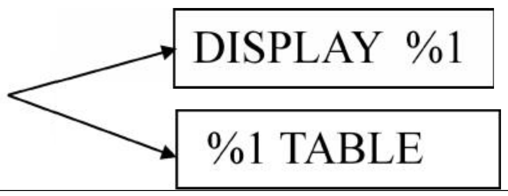
```
        LDS     #3
        LDT     #600
        LDX     #0
ADDLP   LDA     ALPHA,X
        ADS     BETA,X
        STA     GAMMA,X
        ADDR    S,X
        COMPR   X,T
        JLT     ADDLP

ALPHA  RESW  200
```

	BETA RESW 200 GAMMA RESW 200			

4(a)	**Explain MS-DOS Linker.** Complier/Assembler: – Source Program →Object module (.obj) – MS-DOS object module (Figure 3.15) • LEDATA similar to Text record, LIDATA: repeated records. • FIXUP similar to Modification record. Linker (Linkage editor): – Object codes → executable (.exe). – Pass 1 of Two passes • computing starting address of each segment, • segments of same name and same class from different modules are combined • segments of same name but different classes from different modules are concatenated. • Constructing a symbol table associating address with each segment and external symbol • Searching library for any unsolved undefined symbol, if possible. – Pass 2 of the two pass linkage editor • Extracting the translated instructions and data from object modules and building an image of the executable program in memory • The executable is organized by segment, not by the order of the object modules • Memory image allows easy rearrangement caused by combination and concatenation • Temporary disk file may be used if memory is not enough. • LEDATA/LIDATA and corresponding FIXUP are processed (placed into memory in binary format). Repeated data in LIDATA is expanded • relocation within a segment (caused by combination and concatenation) is performed and external reference is resolved. • Relocation related to starting of a segment is added to a table of segment fix up, which is used for relocation when loaded. • Write it to .exe file, containing segment fixups, information about memory requirement, entry points, and the initial contents for registers CS and SP. • When .exe file is typed, OS (a loader in OS) loads the file to memory to execute.	[5]	CO3	L1
(b)	**Explain ELENA Macro processor.** Macro definition header: a sequence of keywords and parameter markers (%) at least one of the first two tokens in a macro header must be a keyword, not a parameter marker body: the character & identifies a local label macro time instruction (.SET, .IF .JUMP, .E) macro time variables or labels (.)	[5]	CO3	

Macro invocation
There is no single token that constitutes the macro “name”
Constructing an index of all macro headers according to the keywords in the first two tokens of the header
Example
DEFINITION:
ADD %1 TO %2
ADD %1 TO THE FIRST ELEMENT OF %2
INVOCATION:

DISPLAY TABLE



5 **Explain the Following Machine independent features of loader :** [10] CO3 L1
i) Relocation
ii) Program Linking

1)Relocation
Loaders that allow for program relocation are called relocating loaders or relative loaders
There are two methods for specifying relocation as part of the object program.
i)Modification record
A Modification record is used to describe each part of the object code that must be when program is relocated. There is one modification record for each value that must be changed during relocation. Each modification record specifies the starting address and length of the field whose value is to be altered. It then describes modification to be performed.
ii) Relocation bit (Bit Mask)
If a machine primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using relocation bit
Each instruction is associated with one relocation bit. It Indicates that the corresponding word should be modified or not.
0: no modification is needed
1: modification is needed
This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

Text record:

col 1: T
col 2-7: starting address
col 8-9: length (byte)
col 10-12: relocation bits
col 13-72: object code

These relocation bits in a Text record are gathered into bit masks.
Twelve-bit mask is used in each Text record (col:10-12 – relocation bits), since each text record contains less than 12 words, unused words are set to 0, and, any value that is to be modified during relocation must coincide with one of these 3-byte segments.

E.g. FFC=111111111100
E00=111000000000

2) Program Linking
The Goal of program linking is to resolve the problems with external references

(EXTREF) and external definitions (EXTDEF) from different control sections.
EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.
ex: EXTDEF BUFFER, BUFFEND, LENGTH EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.
ex: EXTREF RDREC, WRREC EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF
The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record
The format of the Define record (D) along with examples is as shown here.

Col. 1 D
Col. 2-7 Name of external symbol defined in this control section
Col. 8-13 Relative address within this control section (hexadecimal)
Col.14-73 Repeat information in
Col. 2-13 for other external symbols

Example records
D LISTA 000040 ENDA 000054 D LISTB 000060 ENDB 000070

Refer record
The format of the Refer record (R) along with examples is as shown here.

Col. 1 R
Col. 2-7 Name of external symbol referred to in this control section
Col. 8-73 Name of other external reference symbols

Example records
R LISTB ENDB LISTC ENDC R LISTA ENDA LISTC ENDC R LISTA ENDA LISTB ENDB

6	**What are the different macro processor design options? Explain Briefly**	[10]	CO3	L1

1) **Recursive Macro Expansion**

We have seen an example of the definition of one macro instruction by another. But we have not dealt with the invocation of one macro by another. The following example shows the invocation of one macro by another macro.

Problem of Recursive Expansion

Previous macro processor design cannot handle such kind of recursive macro invocation and expansion The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten. The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, i.e., the macro process would forget that it had been in the middle of expanding an "outer" macro.

Solutions

Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.

If you are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses.

The procedure EXPAND would be called when the macro was recognized. The arguments from the macro invocation would be entered into ARGTAB as follows: The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin. The processing would proceed normally until statement invoking RDCHAR is processed. This time, ARGTAB would look like at the expansion, when the end of RDCHAR is recognized, EXPANDING would be set to FALSE.

Thus the macro processor would 'forget' that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the value in ARGTAB was overwritten with the arguments from the invocation of RDCHAR.

2) **General-Purpose Macro Processors**

Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages

Pros

- Programmers do not need to learn many macro languages.
- Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.

Cons

- Large number of details must be dealt with in a real programming language Situations in which normal macro parameter substitution should not occur, e.g., comments.

- Facilities for grouping together terms, expressions, or statements
  Tokens, e.g., identifiers, constants, operators, keywords
- Syntax had better be consistent with the source programming language

3) **Macro Processing within Language Translators**

The macro processors we discussed are called “Preprocessors”.

- Process macro definitions
- Expand macro invocations
- Produce an expanded version of the source program, which is then used as input to an assembler or compiler

You may also combine the macro processing functions with the language translator:

- Line-by-line macro processor
- Integrated macro processor

Line-by-Line Macro Processor

Used as a sort of input routine for the assembler or compiler

- Read source program
- Process macro definitions and expand macro invocations
- Pass output lines to the assembler or compiler

Benefits

- Avoid making an extra pass over the source program.
- Data structures required by the macro processor and the language translator can be combined (e.g., OPTAB and NAMTAB)
- Utility subroutines can be used by both macro processor and the language translator.
  Scanning input lines
  Searching tables
  Data format conversion
- It is easier to give diagnostic messages related to the source statements

Integrated Macro Processor

An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.

Ex (blanks are not significant in FORTRAN)

```
DO 100 I = 1,20

        a DO statement
DO 100 I = 1

        An assignment statement

        DO100I: variable (blanks are not significant in FORTRAN)
```

An integrated macro processor can support macro instructions that depend upon the context in which they occur.

7(a)	**Briefly discuss the machine dependent code optimization Techniques of Compiler**	[8]	CO4	L1

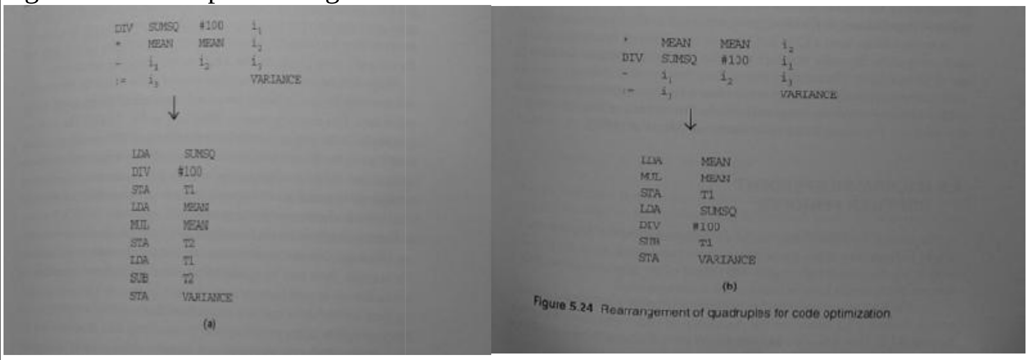
There are several different possibilities for performing machine dependent code optimization.

<u>1)Assignment and use of registers</u>

General purpose register are used for various purpose like storing values or intermediate result or for addressing (base register, index register).
Registers are also used as instruction operands. Machine instructions that use registers as operands are usually faster than the corresponding instruction that refer to location in memory. Therefore it is preferable to store value or intermediate results in registers.
There are rarely as many registers available as we would like to use. The problem then becomes one of selecting which register value to replace when it is necessary to assign a register for some other purpose.
One approach is to scan the program and the value that is not needed for longest time will be replaced. If the register that is being reassigned contains the value of some variable already stored in memory, the can value can be simply discarded. Otherwise this value must be saved using temporary variable
Second approach is to divide the program into basic blocks. A basic block is a sequence of quadruples with one entry point, which is at the beginning of the block, one exit point, which is at the end of the block and no jumps within the block. When control passes from one block to another all the values are stored in temporary variables.

<u>2)Rearranging quadruples before machine code is generated.</u>

Note that the value of the intermediate result i1 is calculated first and stored in temporary variable T1. Then the value of i2 is calculated. The third quadruple in this series calls for subtracting the value of i2 from i1. Since i2 had just been computed, its value is available registers A; however, this does no good, since the first operand for a – operation must be in register. It is necessary to store the value of i1 from T1 into register A before performing the subtraction.



```
DIV   SUMSQ   #100   i1
*     MEAN    MEAN   i2
-     i1      i2     i3
:=    i3             VARIANCE

        ↓

 LDA     SUMSQ
 DIV     #100
 STA     T1
 LDA     MEAN
 MUL     MEAN
 STA     T2
 LDA     T1
 SUB     T2
 STA     VARIANCE

        (a)
```

```
*     MEAN    MEAN   i2
DIV   SUMSQ   #100   i1
-     i1      i2     i3
:=    i3             VARIANCE

        ↓

 LDA     MEAN
 MUL     MEAN
 STA     T1
 LDA     SUMSQ
 DIV     #100
 SUB     T1
 STA     VARIANCE

        (b)
```

Figure 5.24 Rearrangement of quadruples for code optimization

With a little analysis, an optimizing compiler could recognize this situation and rearrange the quadruples so the second operand of the subtraction is computed first. The resulting machine code requires two fewer instructions and uses only one temporary variable instead of two.

<u>3)Taking advantage of specific characteristics and instructions of the target machine</u>

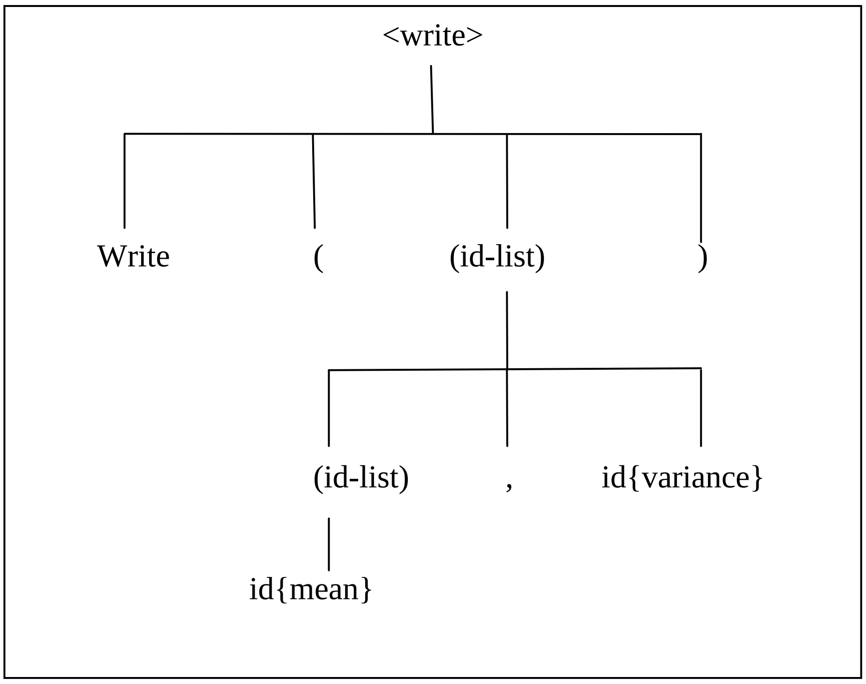
For example there may be special loop-control instructions or addressing modes that can be used to create more efficient object code.
On some computers there are high level machines instructions that can perform complicated functions such as calling procedures and manipulating data structures in single operations.
Use of such feature can greatly improve the efficiency of the object program.
CPU is made of several functional units. On such system machine instruction order can affect speed of execution. Consecutive instructions that require different functional unit can be executed at the same time.

(b) **Write Algorithm for Absolute Loader** [2] CO3 L2

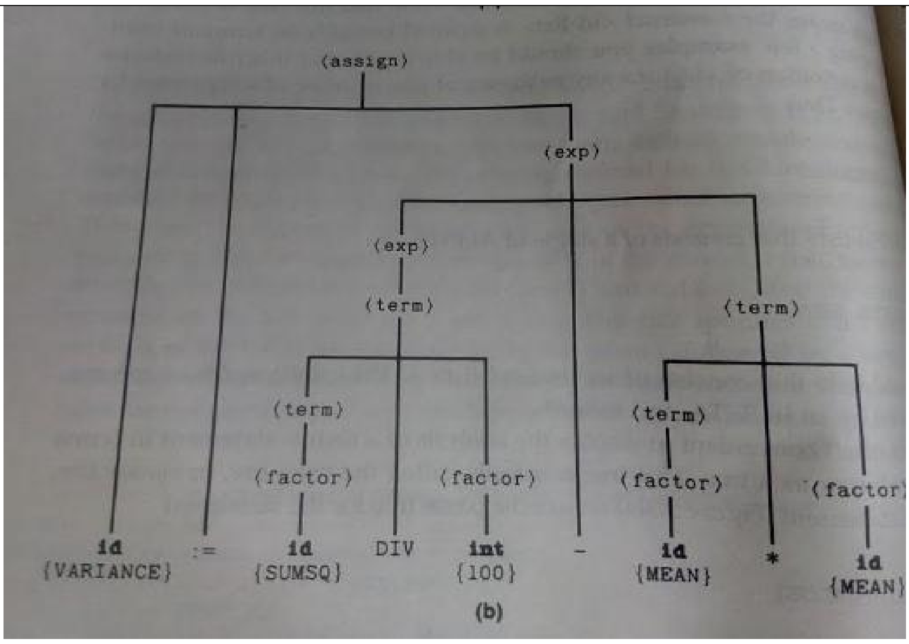
```
Begin
read Header record
verify program name and length
read first Text record
while record type is <> 'E' do
 begin
 {if object code is in character form, convert into internal  representation}
move object code to specified location in memory
read next object program record
end
jump to address specified in End record
end
```

8(a) **Construct Parsing Tree for following PASCAL statement** [8] CO4 L3

1) **WRITE(MEAN,VARIANCE)**



2) **VARIANCE := SUM DIV 100 – MEAN * MEAN**

	 ⟨assign⟩ ⟨exp⟩ ⟨exp⟩ ⟨term⟩ ⟨term⟩ ⟨term⟩ ⟨factor⟩ ⟨factor⟩ ⟨term⟩ ⟨factor⟩ ⟨factor⟩ id {VARIANCE} := id {SUMSQ} DIV int {100} - id {MEAN} * id {MEAN} (b)			
(b)	**What are the Basic Functions of complier?** Basic functions of Compiler are Scanning, parsing, and (object) code generation. 1) Lexical analysis Scan the program to be compiled and recognize the tokens (from string of characters). 2) Syntactic analysis The source statements written by programmers are recognized as language constructs described by the grammar. This process is achieved by building the parse tree for the statements being translated. 3) Code Generation Most compilers create machine-language programs directly instead of producing a symbolic program for later translation by an assembler.	[2]	CO4	L1