CMR INSTITUTE OF TECHNOLOGY

Affiliated to VTU, Approved by AICTE, Accredited by NBA and NAAC with "A++" Grade
ITPL MAIN ROAD, BROOKFIELD, BENGALURU-560037, KARNATAKA, INDIA

Department of Computer Science Engineering

Answer Scheme & Model Solution- IAT3

Sub: System Software and Compilers		**Sub Code: 18CS61**	**Sem/Branch: VI / CSE**	**Sections: A,B,C**

			MARKS	**CO**	**RBT**
Question	1	i)Differentiate between syntax tree, parse tree & annotated parse tree ii)Compare various forms of 3-address code instruction representations. Translate the arithmetic expression b * -c + b * -c to its equivalent 3-address code & its corresponding quadruple, triple & indirect triple form. Consider -c as unary minus	[3+7]	CO2	L3
Scheme		i) Min 2 Difference points ii) Comparison of various forms of 3- address code - min 2 points ; Translation of the given arithmetic expression in 3 forms & 3 address code.	*3* *3+4*		
Solution		i) Parse tree is a hierarchical structure that defines the derivation of the grammar to yield input strings. In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol. It is the graphical description of symbols that can be terminals or non-terminals. Parse tree follows the precedence of operators. A syntax tree is a tree that displays the syntactic structure of a program while ignoring inappropriate analysis present in a parse tree. Thus, the syntax tree is nothing more than a condensed form of the parse tree. The operator and keyword nodes of a parse tree are shifted to their parent and a group of individual production is replaced by an individual link. An Annotated Parse Tree is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called annotating or decorating the parse tree. An annotated parse tree is one in which various facts			

about the program have been attached to parse tree nodes.

ii) 1. Quadruple – It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

It is easy to rearrange code for global optimization.

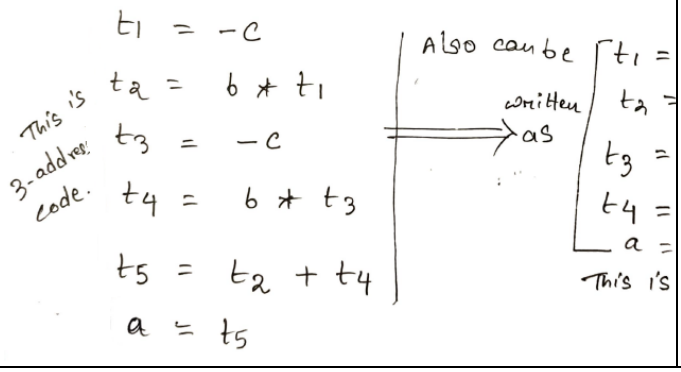
One can quickly access value of temporary variables using symbol table.

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.

Temporaries are implicit and difficult to rearrange code.

It is difficult to optimize because optimization involves moving intermediate code.

3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.



Now converting to Quadruple :-

Add	OP	arg1	arg2	Result
(0)	−	c		t1
(1)	*	b	t1	t2
(2)	−	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

(2) Triple :-

It contains 3 fields → operator, argument 1, argument 2

Add	OP	arg1	arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

(3) Indirect-Triple :-

Here also we use triple only but we need to have an extra table. The table contains pointer to the triple. This table called instruction array to list the pointers to in desired order.

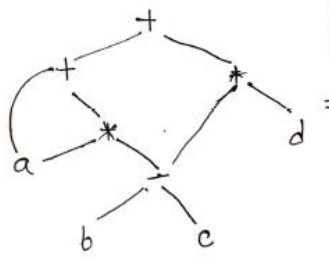
add	OP	arg1	arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

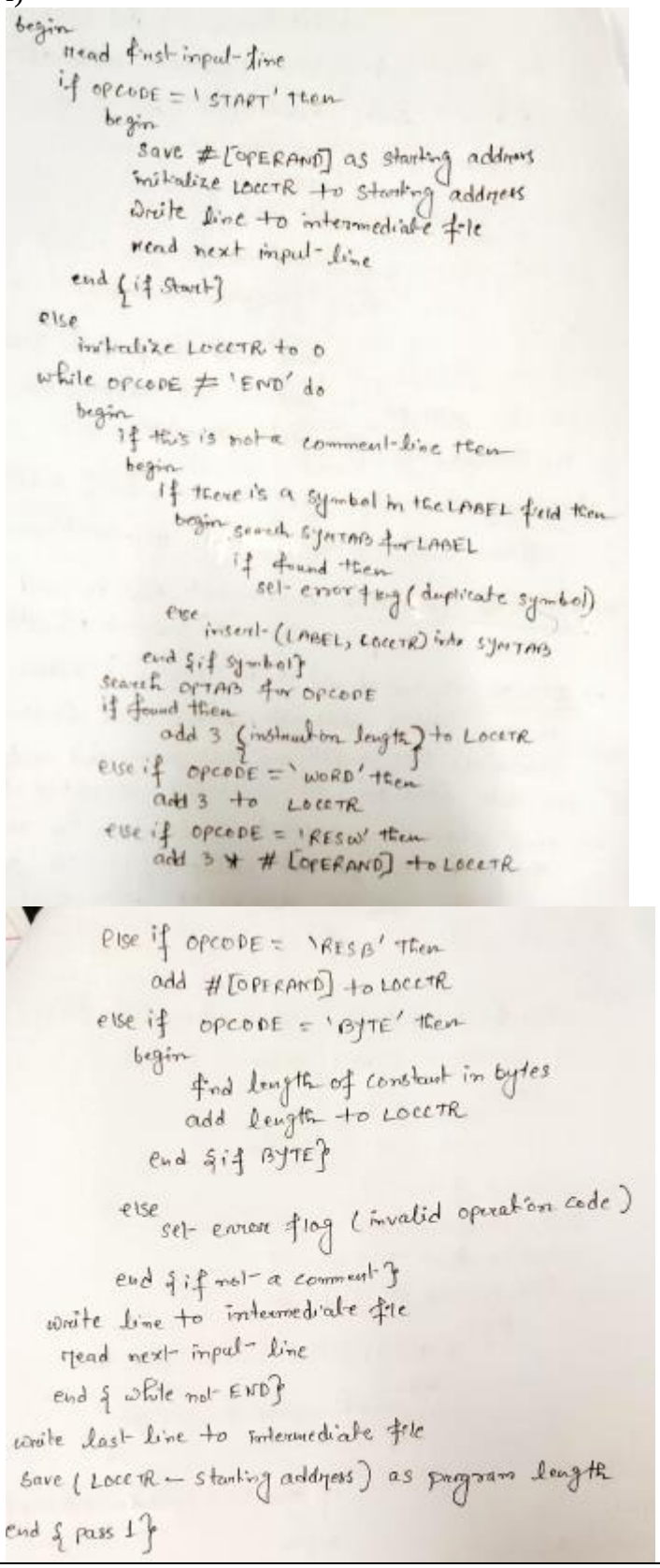
pointers

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)
105	(5)

Instruction a

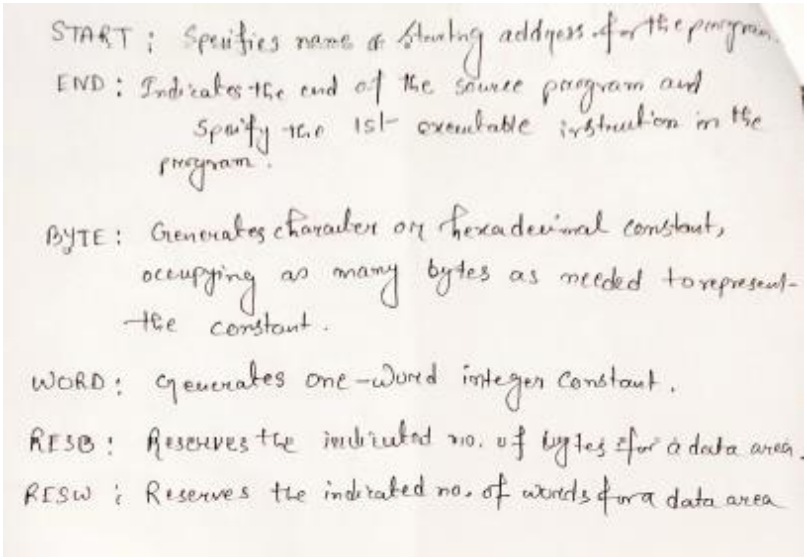
Question	2	Briefly explain the issues in the design of a code generator	10	CO2	L2
Scheme		All the 6 issues	*10*		

Solution		(1) Input to code generator (2) Target Program (3) Memory Management (4) Instruction selection (5) Register allocation (6) Evaluation order. The most important criterion for a code generator is that it produce correct code. These are the 6 issues or criteria that a code generator should consider while producing correct code.			
Question	3	i)Demonstrate the SDD to produce a DAG for arithmetic expression grammar. ii)Draw a DAG and write the corresponding 3-address code for the expression a+ a * (b-c) + (b-c) *d	[5+5]	CO2	L3
Scheme		Writing the SDD Drawing the DAG & writing the 3-address code	*5* *5*		
Solution		i) Grammar Production — Semantic Rules 1) $E \rightarrow E_1 + T$ — E.node = new Node('+', E_1.node, T.node) 2) $E \rightarrow E_1 - T$ — E.node = new Node('-', E_1.node, T.node) 3) $E \rightarrow T$ — E.node = T.node 4) $T \rightarrow (E)$ — T.node = E.node 5) $T \rightarrow id$ — T.node = new leaf (id, id.entry) 6) $T \rightarrow num$ — T.node = new leaf (num, num.val) ii) DAG for the expression a + a * (b-c) + (b-c) *  3-address code $t_1 = b - c$ $t_2 = a * t_1$ $t_3 = a + t_2$ $t_4 = t_1 * d$ $t_5 = t_3 + t_4$			
Question	4	Explain in detail SIC/XE Machine Architecture	10	CO1	L2
Scheme		All the 7 dimensions of SIC/XE architecture	*10*		
Solution		i. Memory ii. Registers iii. Data Format iv. Instruction Format v. Addressing Mode vi. Instruction set			

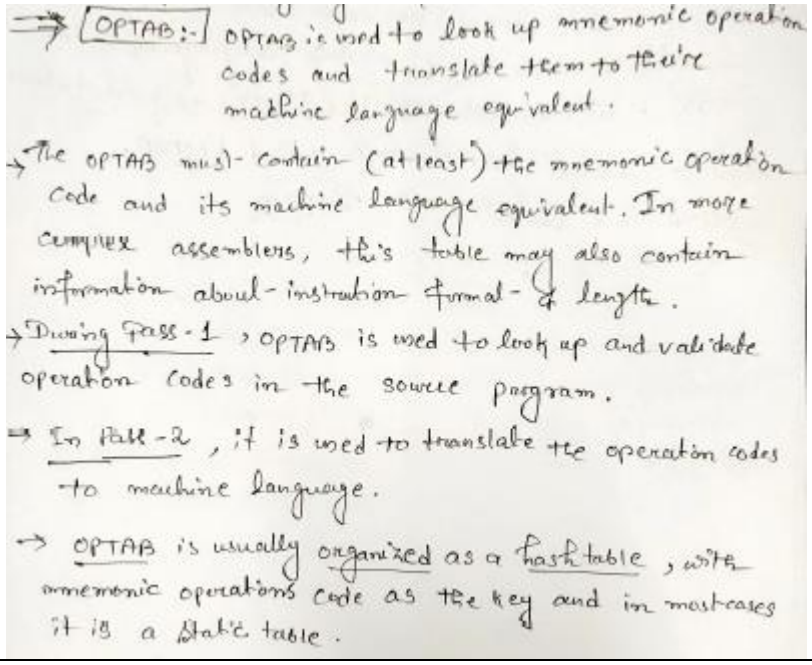
		vii. Input & Output			
Question	5	i)Write the algorithm for pass-1 of a two-pass assembler. ii)Explain the Assembler directives and data structures used in Assembler.	[6+4]	CO1	L2
Scheme		Complete Algorithm Listing & Explaining Assembler directive & Data Structure	*6* *4*		
Solution		i) 			

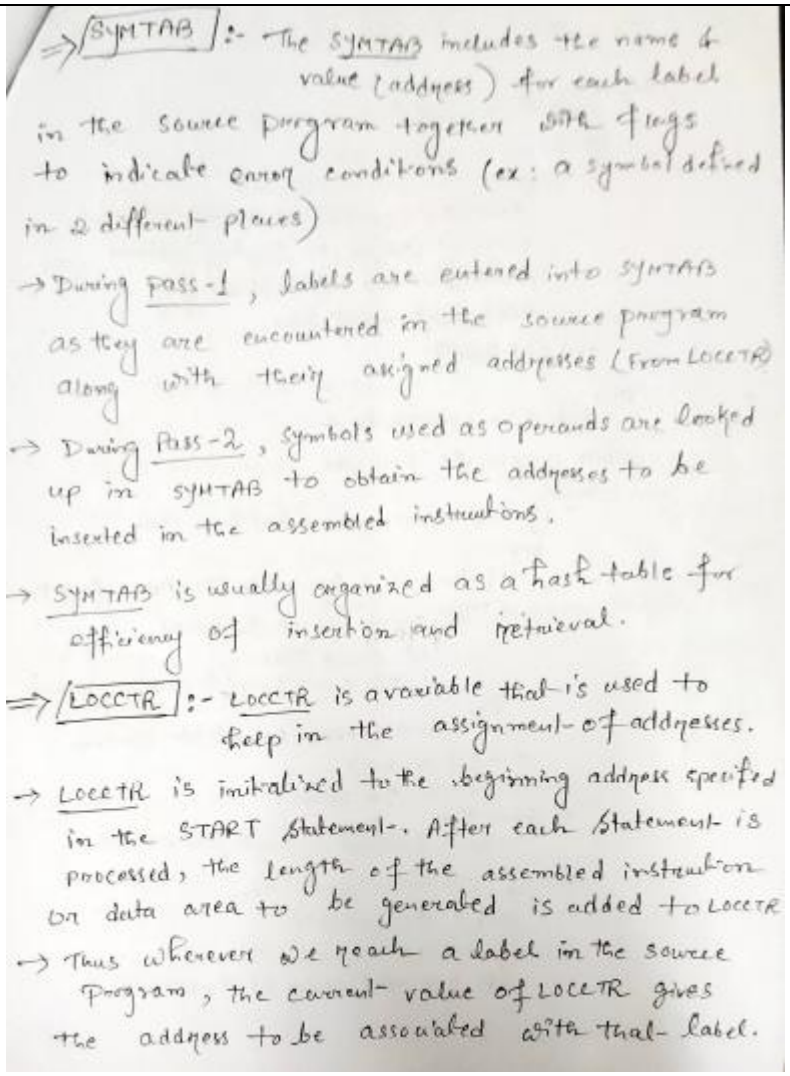
```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR to starting address
      write line to intermediate file
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
  while OPCODE ≠ 'END' do
    begin
      if this is not a comment line then
        begin
          if there is a symbol in the LABEL field then
            begin search SYMTAB for LABEL
              if found then
                set error flag (duplicate symbol)
              else
                insert (LABEL, LOCCTR) into SYMTAB
            end {if symbol}
          search OPTAB for OPCODE
          if found then
            add 3 {instruction length} to LOCCTR
          else if OPCODE = 'WORD' then
            add 3 to LOCCTR
          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 length of constant in bytes
              add length to LOCCTR
            end {if BYTE}
          else
            set error flag (invalid operation code)
          end {if not a comment}
        write line to intermediate file
        read next input line
        end {while not END}
      write last line to intermediate file
      save (LOCCTR - starting address) as program length
    end {pass 1}
```

ii) Assembler Directives


START : Specifies name & starting address of the program.
END : Indicates the end of the source program and specify the 1st executable instruction in the program.
BYTE : Generates character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
WORD : Generates one-word integer constant.
RESB : Reserves the indicated no. of bytes for a data area.
RESW : Reserves the indicated no. of words for a data area

Assembler Data Structures


OPTAB :- OPTAB is used to look up mnemonic operation codes and translate them to their machine language equivalent.
The OPTAB must contain (at least) the mnemonic operation code and its machine language equivalent. In more complex assemblers, this table may also contain information about instruction format & length.
During Pass-1, OPTAB is used to look up and validate operation codes in the source program.
In Pass-2, it is used to translate the operation codes to machine language.
OPTAB is usually organized as a hash table, with mnemonic operations code as the key and in most cases it is a static table.



⇒ SYMTAB :- The SYMTAB includes the name & value (address) for each label in the source program together with flags to indicate error conditions (ex: a symbol defined in 2 different places)

→ During pass-1, labels are entered into SYMTAB as they are encountered in the source program along with their assigned addresses (from LOCCTR)

→ During pass-2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions.

→ SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.

⇒ LOCCTR :- LOCCTR is a variable that is used to help in the assignment of addresses.

→ LOCCTR is initialized to the beginning address specified in the START statement. After each statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR

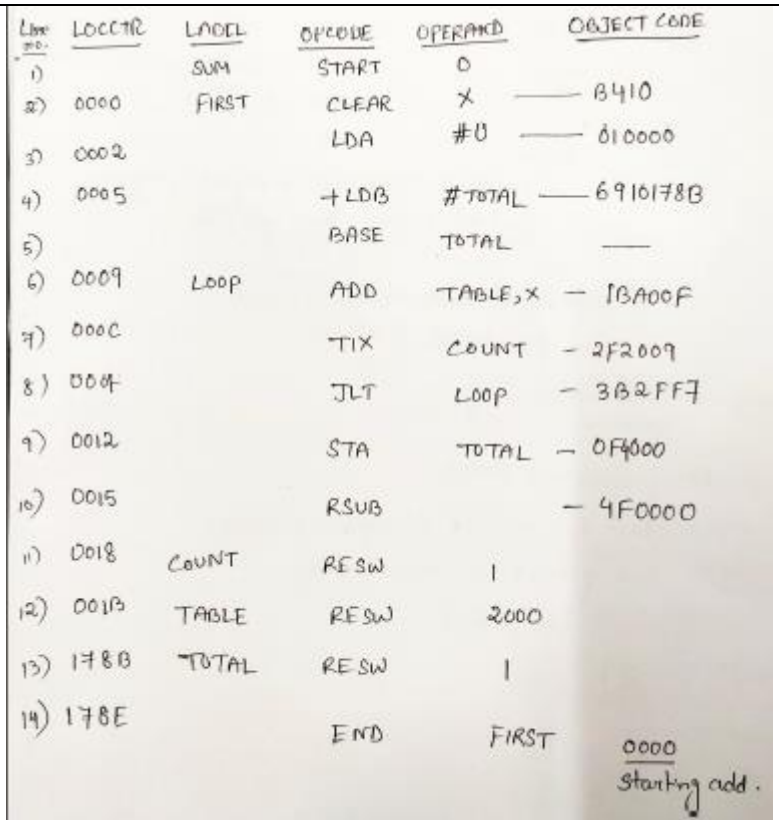
→ Thus whenever we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label.

Question	6	Convert the following SIC/XE source program into an object program (show computation step for each statement). **Given,** CLEAR=B4, LDA=00, LDB=68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA=0C **Source Program:**			10	CO1	L3

SUM	START	0
FIRST	CLEAR	X
	LDA	#0
	+LDB	#TOTAL
	BASE	TOTAL
LOOP	ADD	TABLE, X
	TIX	COUNT
	JLT	LOOP
	STA	TOTAL
	RSUB	
COUNT	RESW	1
TABLE	RESW	2000
TOTAL	RESW	1
	END	FIRST

Scheme	Writing object code correctly for each statement Writing the Object program	*7* *3*		

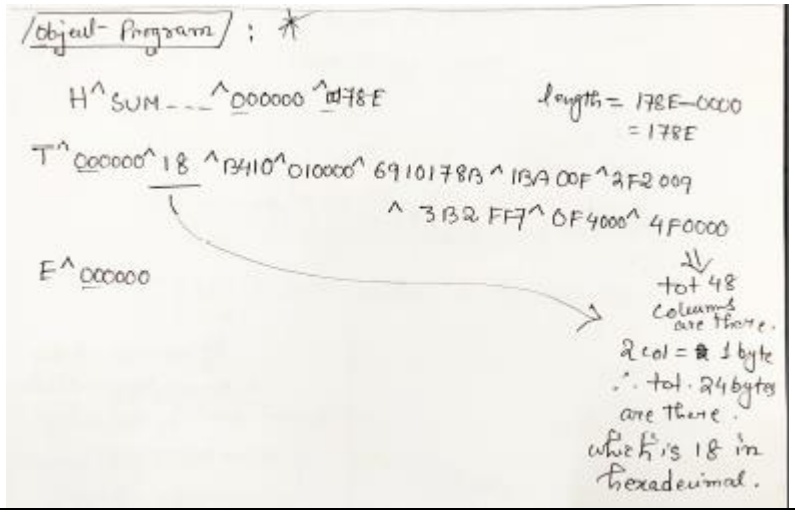
Solution



Line no.	LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
1)		SUM	START	0	
2)	0000	FIRST	CLEAR	X	— B410
3)	0002		LDA	#0	— 010000
4)	0005		+LDB	#TOTAL	— 6910178B
5)			BASE	TOTAL	—
6)	0009	LOOP	ADD	TABLE,X	— 1BA00F
7)	000C		TIX	COUNT	— 2F2009
8)	000F		JLT	LOOP	— 3B2FF7
9)	0012		STA	TOTAL	— 0F4000
10)	0015		RSUB		— 4F0000
11)	0018	COUNT	RESW	1	
12)	001B	TABLE	RESW	2000	
13)	1788	TOTAL	RESW	1	
14)	178E		END	FIRST	

0000
Starting add.

Object Program



Object Program :

H^SUM---^000000^00178E

length = 178E−0000
= 178E

T^000000^18^B410^010000^6910178B^1BA00F^2F2009
^3B2FF7^0F4000^4F0000

E^000000

tot 48 columns are there.
2 col = 1 byte
∴ tot. 24 bytes are there.
where 18 is in hexadecimal.