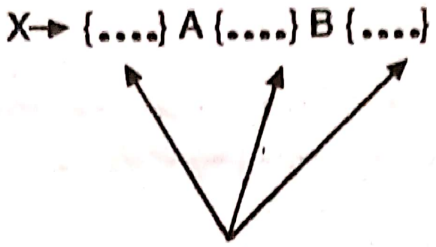
Q. 3(a) Give the translation scheme that converts infix to postfix notation. Generate the annotated parse tree for input string 3 – 5 + 4. (3 Marks)

Ans. :

A translation scheme is a context free grammar(CFG) in which attributes are associated with the grammar symbols. And semantic actions enclosed between braces { } are inserted within the right side of productions.

For example



Semantic actions

Translation scheme generates the output by executing the semantic actions in an ordered manner. It uses depth first traversal. Consider a simple translation scheme that converts infix expressions into postfix expressions.

$E \rightarrow TR$

$R \rightarrow addop\ T\ \{print(addop \cdot lexeme)\}\ R \mid \in$

$T \rightarrow num\ \{print(num \cdot Val)\}$

For example 3 – 5 + 4 as 3 5 – 4 +

Fig. 1-Q. 3(a) shows the annotated parse tree for the input 3 – 5 + 4 with each semantic action attached as the appropriate child of the node corresponding to the left side of their production.

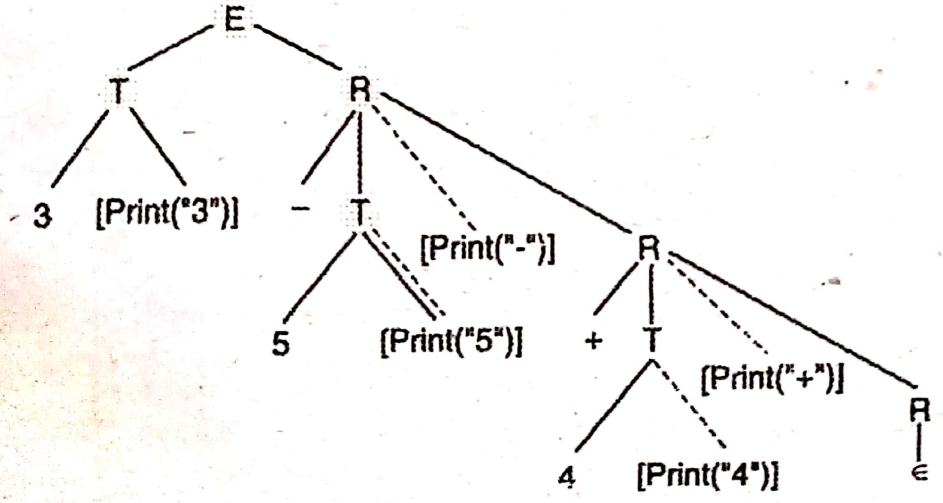


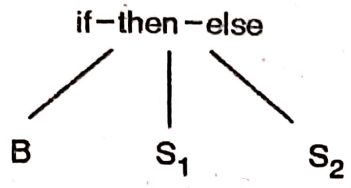
Fig. 1-Q. 3(a) : Annotated Parse tree for 3 – 5 + 4 with action

Q. 3(c)(OR) Define syntax tree. What is s-attributed definition? Explain construction of syntax tree for the expression a – 4 + c using SDD. (7 Marks)

Ans. :

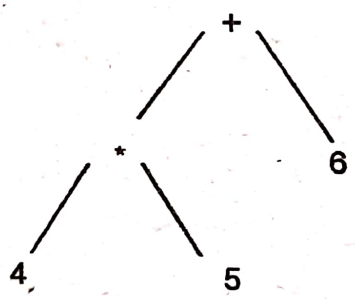
Syntax Tree

An abstract syntax tree is a condensed form of parse tree and it is useful for representing language construct. The production S1 → if B then S1 else S2 appear in a syntax tree as



In a syntax tree, operates and keywords appear as interior node that would be the parent of the leaves in the parse tree. Another simplification in syntax-trees is that chains of single productions may be collapsed. The parse tree becomes the syntax tree as

Syntax directed translation can be based on syntax trees as well as parse trees.



S-attributed definition

A syntax-directed definition that uses synthesized attributes exclusively is said to be an **S-attributed definition.** A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node by bottom up, approach i.e. from the leaves to the root. To compute the S-attributed definition perform the following steps

1. Write syntax directed definition.
2. Generate annotated parse tree.
3. Compute attribute values by following bottom up approach.
4. The value at the root node is the final value of the expression.

Syntax-Directed Definitions

In syntax directed definition, each grammar symbol has an associated set of attributes and each grammar production has an associated set of semantic rules. The value of attribute is computed by a semantic rule associated with the production at that node. An attribute can be a string, a number, a type, a memory location.

The attribute is partitioned into two subsets :

1. Synthesized attribute : In this value is computed from the attribute value of children node.
2. Inherited attribute : In this value is computed from the attribute values of siblings.

Semantic rules set up dependencies between attributes that will be represented by a graph called as the **dependence graph.** From the dependency graph It can derive an evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree for the input string. A semantic rule may also have side effects, e.g. printing a value or updating a global variable so implementation need not explicitly construct dependency graph or parse tree, it just has to produce the same output for each input string. A parse tree showing the values of attributes as each node is called an annotated parse tree. While the process of computing the attribute values at the nodes is called annotating or decorating the parse tree.

Thus carrying out the translation specified by the syntax directed definition involves following steps :

1. For the input string w generate the parse tree.
2. Then generate the dependency graph and using topological sort find out the traversal order of the parse tree.
3. Then evaluate the semantic rules by traversing the parse tree in the proper order.

In a syntax-directed definition each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $x := f(y_1, y_2, \ldots y_k)$ where f is a function, and x is either.

1. A synthesized attribute of A and $y_1, y_2, \ldots y_k$ are attributes belonging to the grammar symbols of the production, or
2. An inherited attribute of one of the grammar symbols on the right side of the production and $y_1, y_2, \ldots y_k$ are attributes belonging to the grammar symbols of the production.

In both the cases the attributes x depends on attributes $y_1, y_2, \ldots y_k$. Semantic rules are written to create side effects in terms of procedure calls or program fragments.

An attribute grammar is a syntax-directed definition in which the functions written as expressions in semantic rules cannot have side effects. Semantic rules define the values of dummy synthesized attributes of the non-terminal on the left side of the associated production. Attribute value of terminals are provided by lexical analyzer and have synthesized attributes only if there is no semantic rules for terminals. Also the start symbol is assumed not to have any inherited attributes, unless otherwise stated.

Example

Consider the following context free grammar for the calculator.

$S \rightarrow EN$
$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow$ alpha-numeric
$N \rightarrow ;$

The syntax directed definition for the above grammar is shown in Table 1-Q. 3(c)(OR).

Table 1-Q. 3(c)(OR) : Syntax-directed definition

Production	Semantic Rules
$S \rightarrow EN$	Print (E. Val)
$E \rightarrow E + T$	E · Val : = E · Val + T · Val
$E \rightarrow E - T$	E · Val : = E · Val - T · Val
$E \rightarrow T$	E · Val : = T.Val
$T \rightarrow$ alphanumeric	T · Val : = alphanumeric. lexval
$N \rightarrow ;$	Terminating symbol

The given definition associates an integer valued synthesized attributes i.e. Val with each of the non-terminals E, T and F. The semantic rule computes the value of attribute Val for each non-terminal on the left side from the values of Val for the nonterminals on the right side. The token digit has a synthesized attribute lexval whose value is assumed to be supplied by the lexical analyzer. The rule associated with the production $S \rightarrow EN$ for the starting non-terminal S is just a procedure that prints as output the value of the arithmetic expression generated by EN. And the production $N \rightarrow ;$ to denote the terminating symbol.

The value of a synthesized attribute at a node is computed from the values of attribute at the children of that node in the parse tree. A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed **definition.** A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node by bottom up, approach i.e. from the leaves to the root.

To compute the S-attributed definition perform the following steps

1. Write syntax directed definition.
2. Generate annotated parse tree.
3. Compute attribute values by following bottom up approach.
4. The value at the root node is the final value of the expression.

Example

The S-attributed definition in Table 1-Q. 3(c)(OR) prints the value of an expression containing digits, parenthesis, the operators - and + followed by a ';' character

For the expression a – 4 + c; definition prints the value.

The Fig. 1-Q. 3(c)(OR) shows the annotated parse tree for the given expression.

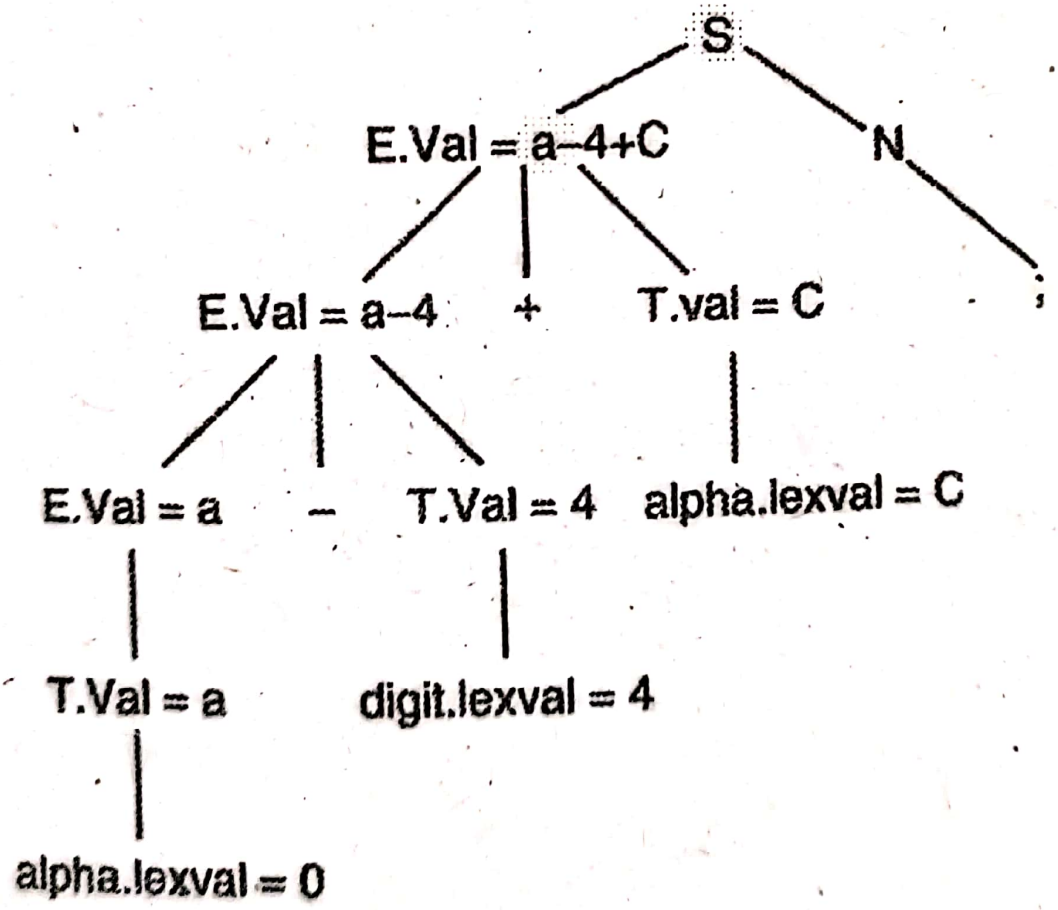


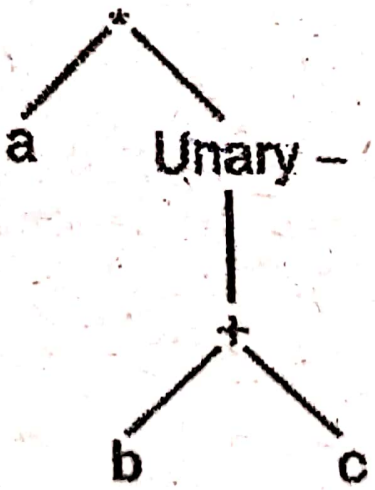
Fig. 1-Q. 3(c)(OR) : Annotated parse tree for expression a – 4 + b

Q. 4(a) **Translate the arithmetic expression a*– (b + c) into**

1. **Syntax tree**
2. **Postfix notation**
3. **Three address code** **(3 Marks)**

Ans. :

1. Syntax tree



2. Postfix notation

a*– (b + c)

a*– (bc +)

a – (bc +)*

3. Three address code

The quadruples for the assignment a*– (b + c) shown in Table 1-Q. 4(a).

$x := y * - z + y * - z$

The three address statement for the above assignment statement is

$t_1 := b + c$

$t_2 := \text{uminus } t_1$

$t_3 := a*t2$

$x := t_3$

Table 1-Q. 4(a) : Quadruple representation of three-address statement

	op	arg1	arg2	result
(0)	+	b	c	t_1
(1)	uminus	t1		t_2
(2)	*	a	t2	t_3
(3)	:=	t3		a

Q. 4(b) Write Syntax Directed Definition to produce three address code for the expression containing the operators := , + , - (unary minus), ( ) and id. (4 Marks)

Ans. :

Consider the following context free grammar for the calculator.

$S \rightarrow EN$

$E \rightarrow E_1 + T$

$E \rightarrow E_1 - T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow -(E)$

$F \rightarrow id$

$N \rightarrow ;$

The syntax directed definition for the above grammar is shown in Table 1-Q. 4(b).

Table 1-Q. 4(b) : Syntax-directed definition

Production	Semantic Rules
$S \rightarrow EN$	Print (E. Val)
$E \rightarrow E_1 + T$	$E \cdot Val := E_1 \cdot Val + T \cdot Val$
$E \rightarrow E_1 - T$	$E \cdot Val := E_1 \cdot Val - T \cdot Val$

Production	Semantic Rules
$E \rightarrow T$	$E \cdot Val := T \cdot Val$
$T \rightarrow T_1 * F$	$T \cdot Val := T_1 \cdot Val \times F \cdot Val$
$T \rightarrow F$	$T \cdot Val := F \cdot Val$
$F \rightarrow (E)$	$F \cdot Val := E \cdot Val$
$F \rightarrow -(E)$	$F \cdot Val := -E \cdot Val$
$F \rightarrow id$	$F \cdot Val := id \cdot lexval$
$N \rightarrow ;$	Terminating symbol

Q. 4(c)(OR) What is importance of intermediate code? Discuss various representations of three address code using the given expression. a = b * – c + b * – c. (7 Marks)

Ans. :

Importance of Intermediate Code

Compiler converts the source program into easy to represent language called intermediate language before translating that directly into machine language program. So It can say that the front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are :

1. A compiler that work on different machine can be created by attaching existing back end to the front end of new machine.
2. A machine independent code optimizer can be applied to the intermediate representation to optimize the code generation.
3. A compiler for different source languages can be created by proving different front ends for corresponding source languages to existing back end.

Fig. 1-Q. 4(c)(OR) shows the role of intermediate code generator. Here It assume that the source program has already been parsed and statically checked.

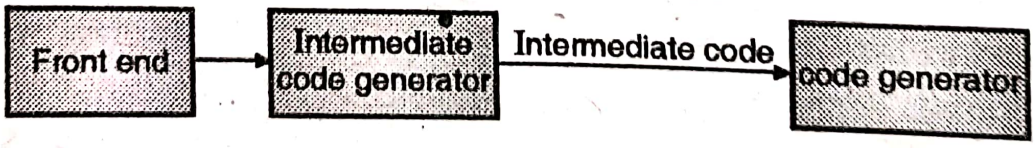


Fig. 1-Q. 4(c)(OR) : Role of intermediate code generator

It know that the three-address is an abstract form of intermediate form. So these three-address statements can be implemented as records with fields for the operator and the operands. There can be several possible implementations for the three-address statements; among them following are the three representations.

Different representations for three-address statement :

a) Quadraple

b) Triples

c) Indirect triples

(a) Quadruple Representation

A quadruple is a structure with at the most four fields as, op arg1, arg2 and result. The op field is used to represent internal node for the operator.

The fields arg1 and arg2 represents the argument or operand of the expression and result field represent the result of the expression i.e. if stores the result of the expression.

The three-address statements with unary operators like a : = – y or

a : = y do not use arg2.

Operator like param does not use arg2 and result, it only use arg1. Conditional and unconditional jumps put the target label in result. The values of the field arg1, arg2 and result are considered as pointers to the symbol-table entries for the names represented by these fields. Similarly whenever the temporary names are created they must be entered into the symbol table.

Example

The quadraples for the assignment a := b * – c + b * – c are shown in Table 1-Q. 4(c)(OR).

The three address statement for the above assignment statement is

t_1 := uminus c

t_2 := b * t_1

t_3 := uminus c

t_4 := b * t_3

t_5 := t_2 + t_4

a := t_5

Table 1-Q. 4(c)(OR) : Quadraple representation of three-address statement

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_5		a

(b) Triple Representation

Triple representation is structure with at the most three field arg1, arg2 and op. The fields arg1 and arg2 are the arguments of the operator op. In triple temporaries are not used instead of that pointers in the symbol table are used directly. Triples corresponds to the representation of a syntax tree or DAG by an array of nodes.

Example 1

Table 2-Q. 4(c)(OR) shows the triples for the assignment statement.

$a := b * - c + b * - c.$

Table 2-Q. 4(c)(OR) : Triple representation of three-address statement

	op	Arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Numbers in the round brackets are used to represent pointers into the triple structure. While symbol-table pointers are represented by the names themselves.

(c) Indirect Triple Representation

In indirect triple representation listing pointers to triples is done instead of listing the triples themselves.

Example

Table 3-Q. 4(c)(OR) : Indirect triples representation of three address statements

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	Op	arg1	arg2
(14)	Uminus	c	
(15)	*	b	(14)
(16)	Uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	Assign	a	(18)

Q. 5(a)(OR) Define following : DAG, Basic Blocks, Flow graph **(3 Marks)**

Ans. :

DAG

DAG is abbreviated for Directed Acyclic Graph. DAG is used to implement transformations on basic blocks.

DAG gives pictorial representation of

(1) How values are computed by each statement in basic block ?

(2) How computed values are subsequently used by further statements in basic block ?

Basic block

Basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at end statement without any halt and any possibility of branching except at the end statement. Thus halting or branching statement may appear as last or ending statement of basic block.

Flow Graph

A program is a set of instructions performing a task. The instructions of the programs are grouped to form basic blocks. Flow graph is a directed graph which is graphical representation of flow of control among all basic blocks constituting program. Flow graph is used to add flow of control information to the set of basic block making up a program.

Q. 4(c)(OR) What is inherited attribute? Write syntax directed definition with inherited attributes for type declaration for list of identifiers. (7 Marks)

Ans. :

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

$S \rightarrow ABC$

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can takes values from S, A, and B.

Expansion :

When a non-terminal is expanded to terminals as per a grammatical rule :

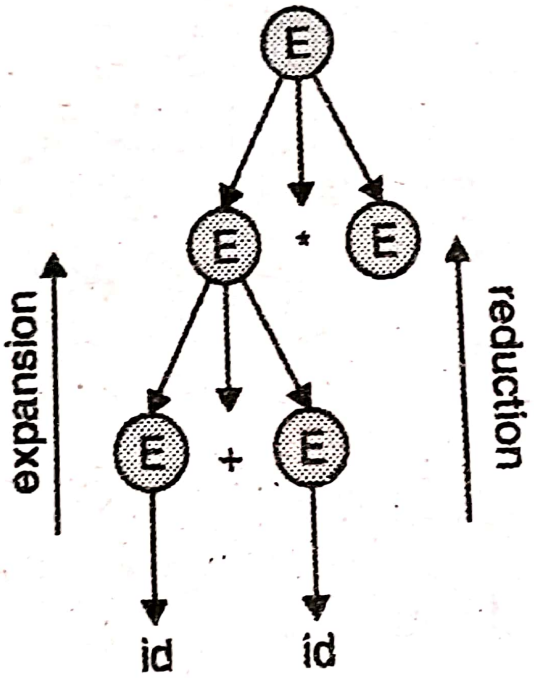


Fig. 1-Q. 4(c)(OR)

Production	Semantic rule
$D \rightarrow TL$	L.in := T.type
$T \rightarrow int$	T.type := integer
$T \rightarrow real$	T.type := real
$L \rightarrow L_1, id$	L_1.in := L.in; addtype(id.entry, L.in)
$L \rightarrow id$	addtype(id.entry,L.in)

Q. 4(a) What is symbol table? For what purpose compiler uses symbol table ? (3 Marks)

Ans. :

Symbol Table is an important data structure created and maintained by the compiler in order to keep tracks of semantics of variable i.e. it stories information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

1. It is built in lexical and syntax analysis phases.
2. The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.
3. It is used by compiler to achieve compile time efficiency.

4. It is used by various phases of compiler as follows :

1. **Lexical Analysis**

Creates new table entries in the table, example likes entries about token.

2. **Syntax Analysis**

Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.

3. **Semantic Analysis**

Uses available information in the table to checks for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.

4. **Intermediate Code generation**

Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

5. **Code Optimization**

Uses information present in symbol table for machine dependent optimization.

6. **Target Code generation**

Generates code by using address information of identifier present in the table.

Q. 4(b)(OR) Explain Quadruples and Triples form of three address code with example.(4 Marks)

Ans. :

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, airg1, and airg2. The results of irrespective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

O p	arg1	arg2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Q. 5(a) **Draw a DAG for expression :**

a + a * (b – c) + (b – c) * d. **(3 Marks)**

Ans. :

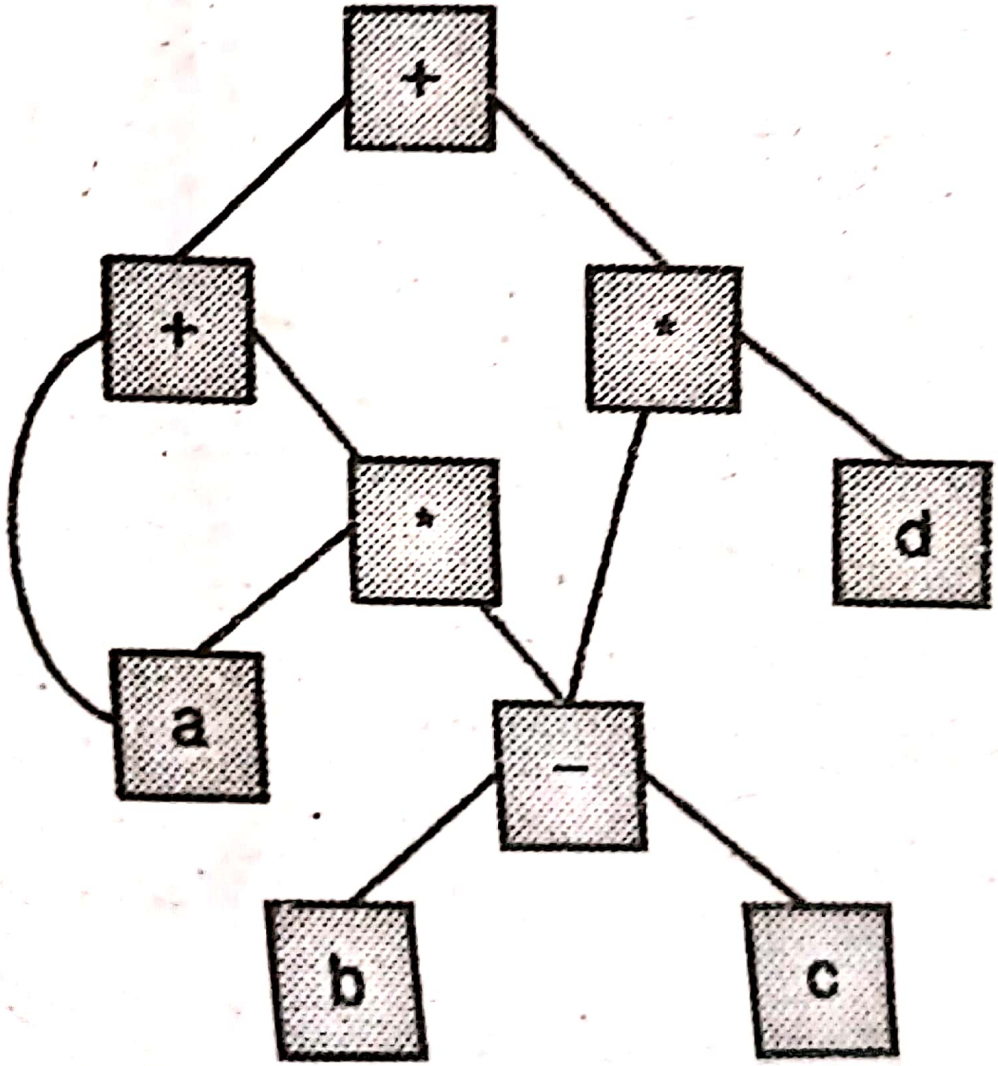


Fig. 1-Q. 5(a)