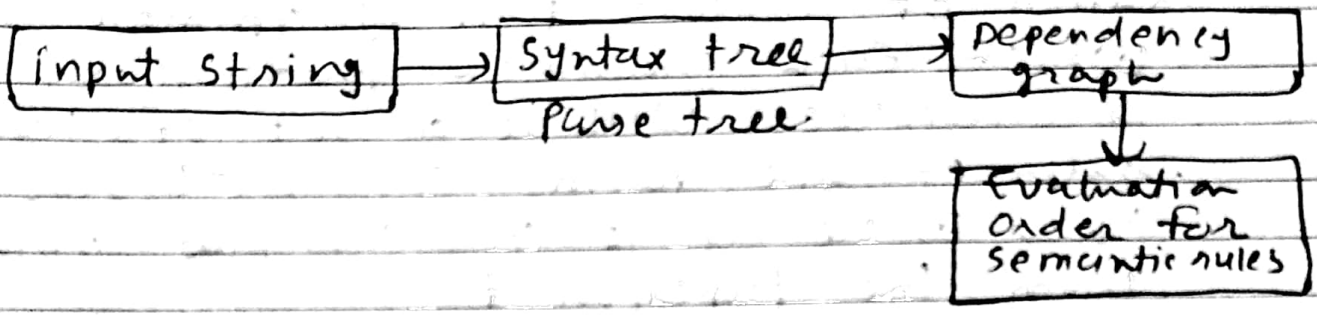
Chapter : 4 Syntax Directed Translation

Page No.

Syntax directed definitions : (SDD).

Conceptual view of Syntax-directed translation :



→ firstly we parse the input token stream and syntax tree is generated. Then the tree is being traversed for evaluating the semantic rules at the parse tree nodes.

→ The Implementation need not have to follow all the steps given in fig. In single pass Implementation semantic rules can be evaluated during parsing without explicitly constructing a parse tree, or dependency graph. In such Semantic evaluation, at the nodes of the Syntax tree, values of the attribute are defined for the given input string. Such a parse tree containing the values of attributes at each node is called an annotated or decorated the parse tree.

SDT : grammar + semantic rules

Syntax directed definition : (SDD)

definition:

Syntax directed definition is a generalization of Context free grammar in which each grammar production $X \to \alpha$ is associated with it a set of semantic rules of the form $a := f(b_1, b_2, \ldots b_k)$ where a is an attribute obtained from the function f.

Attribute : The attribute can be a String, a number, a type, a memory location or anything else. Consider $X \to \alpha$ be a Context free grammar and $a := f(b_1, b_2, \ldots b_k)$ where a is the attribute. (set of semantic rules)

There are two types of attributes :

1 Synthesized attribute :

The attribute 'a' is called Synthesized attribute of X and $b_1, b_2, \ldots b_k$ are attributes belonging to the production symbols.

The value of Synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

2 Inherited attribute : The attribute 'a' is called inherited attribute of one of the grammar symbol on the right side of

attribute : numbers, types, tables reference, string.

Date: / /
Page No.

the production (i.e. α) and $b_1, b_2, \ldots b_k$ are belonging to either x or α

→ The inherited attributes can be computed from the values of the attributes at the siblings and parent of that node.

1 Synthesized attribute :

Example :

Consider the context free grammar is :

$S \rightarrow EN$
$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow f$
$F \rightarrow (E)$
$F \rightarrow$ digit
$N \rightarrow ;$

→ The Syntax directed definition can be written for the above grammar by using Semantic actions for each production :

Production rule	Semantic actions
$S \to EN$	print(E.val)
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to E_1 - T$	$E.val = E_1.val - T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to T_1 / F$	$T.val = T_1.val / F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$
$N \to ;$	Can be ignored by lexical analyzer as ; is terminating symbol.

→ For the non terminals E, T and F the values can be obtained using the attribute "val". Here "val" is a attribute and Semantic rule is computing the value of val.

→ The token digit has Synthesized attribute value can be obtained from lexical analyzer. In the rule $S \to EN$, Symbol S is the start Symbol. This rule is to print the final answer of the expression.

→ In SDD, terminals have Synthesized attributes only.

→ Thus there is no definition of terminal. The Synthesized attributes are quite

often used in Syntax directed definition
The Syntax directed definition that uses only Synthesized attributes is called S-attributed definition.

→ In a parse tree, at each node the semantic rule is evaluated for annotating the S-attributed definition. This processing is in bottom-up fashion.

following steps are followed to Compute S-attributed definition.

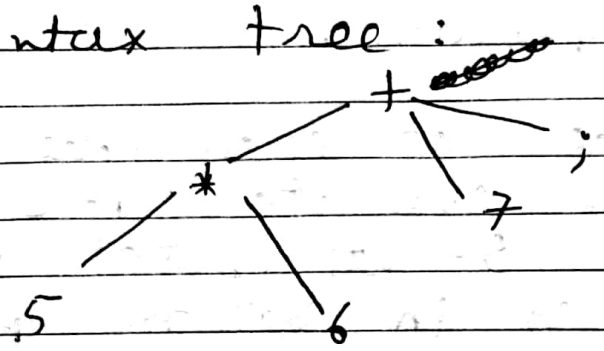
1 Write the Syntax directed definition using appropriate Semantic actions for corresponding production rule of the given grammar.

2 parse tree The annotated parse tree is generated and attribute values are Computed. The Computation is done in bottom up manner.

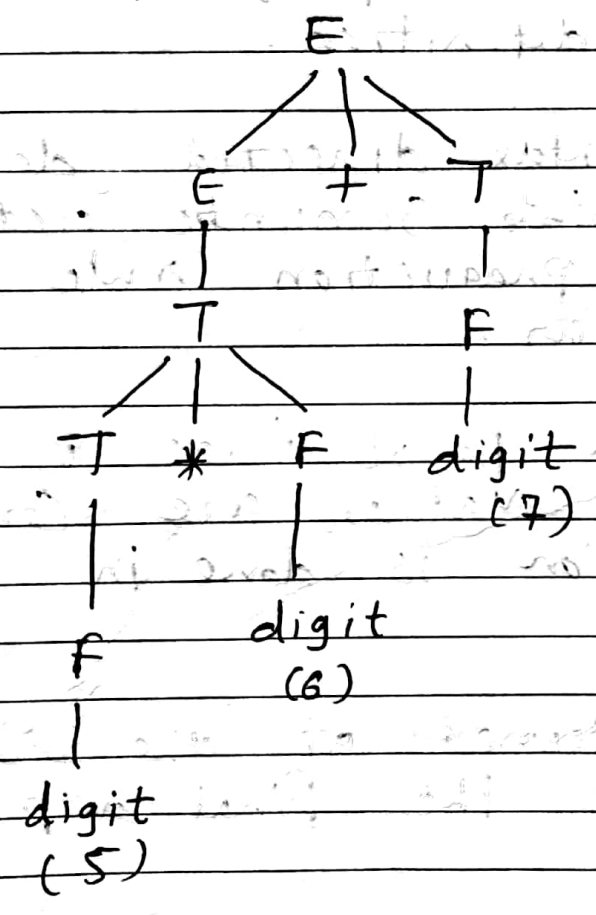
3 The value obtained at the root node is supposed to be the final output.

ex: Construct parse tree, Syntax tree and annotated parse tree

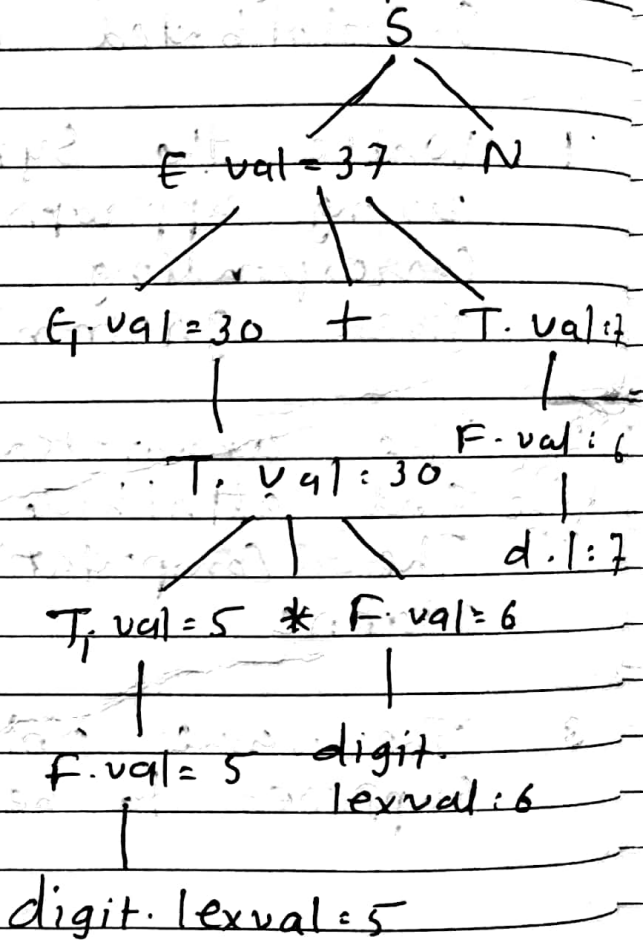
→ Syntax tree :



→ parse tree :



annotated parse tree :



→ for the Computation of attributes we start from left most bottom most node. The rule F → digit is used in order to reduce digit to F. The Semantic action that takes place here is F.val = digit.lexval.

→ The value of digit is obtained from lexical analyzer which becomes the value of F. Hence F.val = 5

→ Then Consider T → F production: Semantic action is T.val → F.val. We can get the T.val = 5. Thus the Computation of S- attributes is done from children.

→ Then Consider: T → T1 * F production: Corresponding Semantic action is:

T.val = T1.val * F.val

T.val = T1.val * F.val

= 5 * 6 = 30

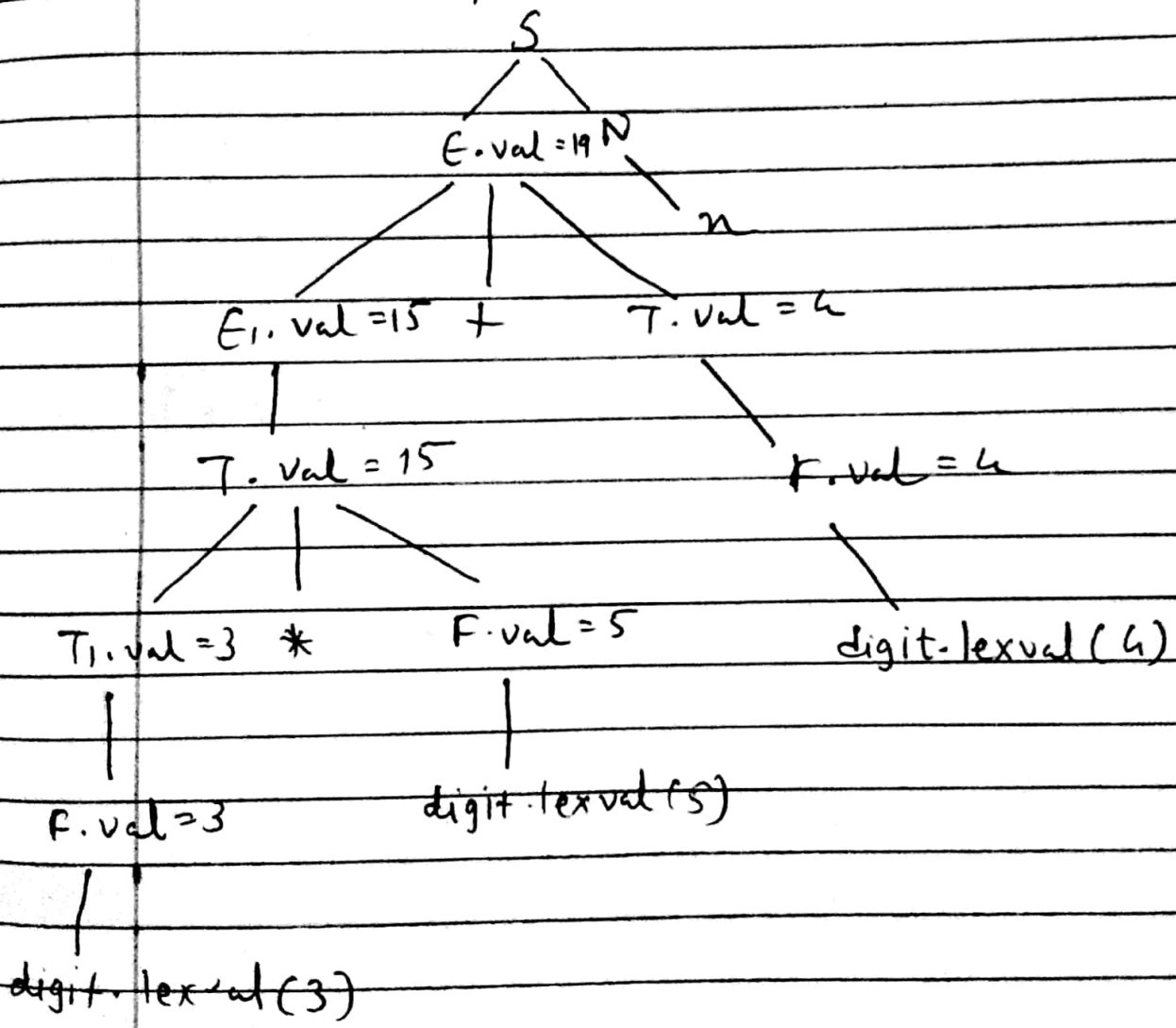
E1.val + T.val becomes the E node

E.val = E1.val + T.val

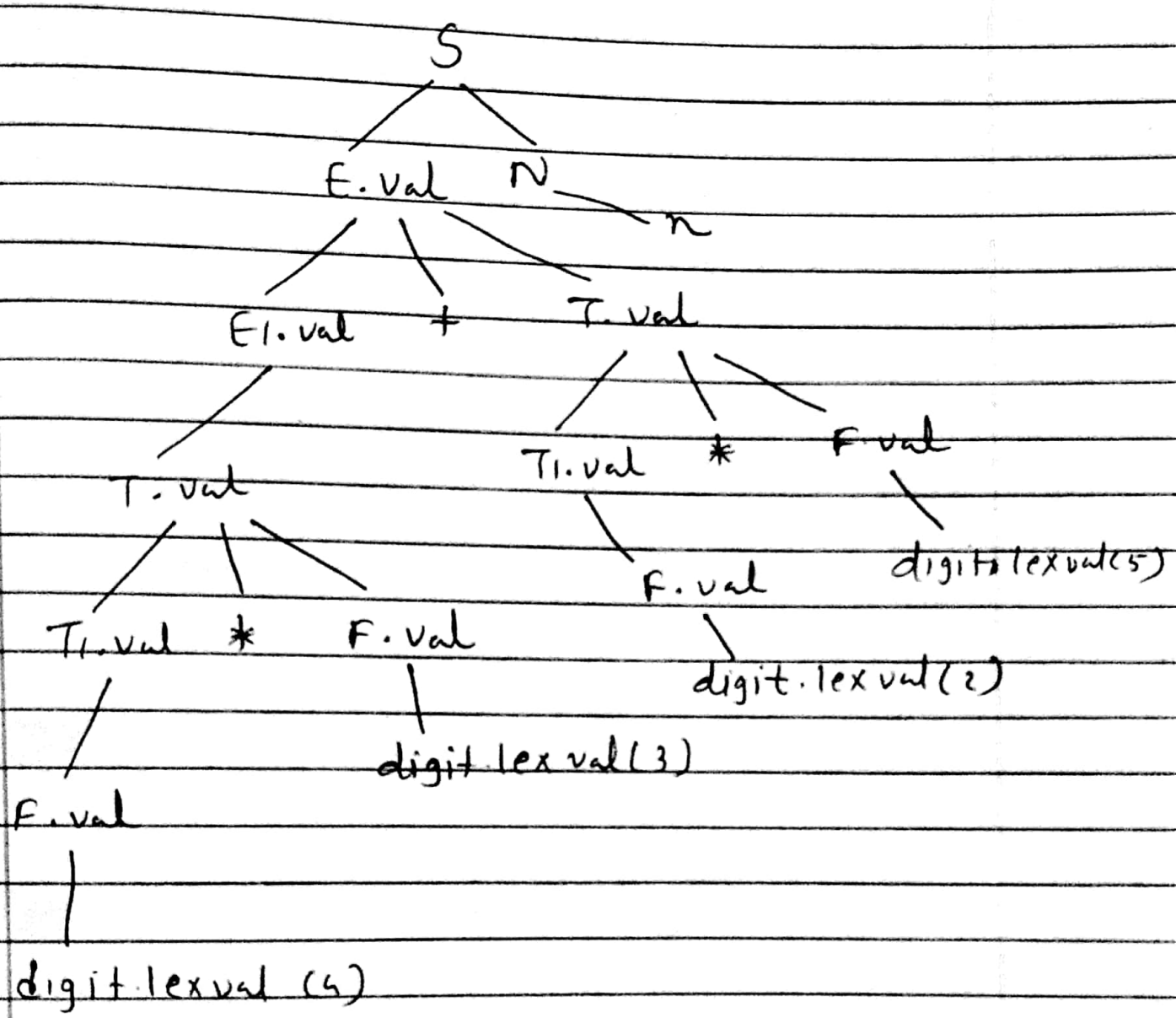
= 30 + 7

E.val = 37

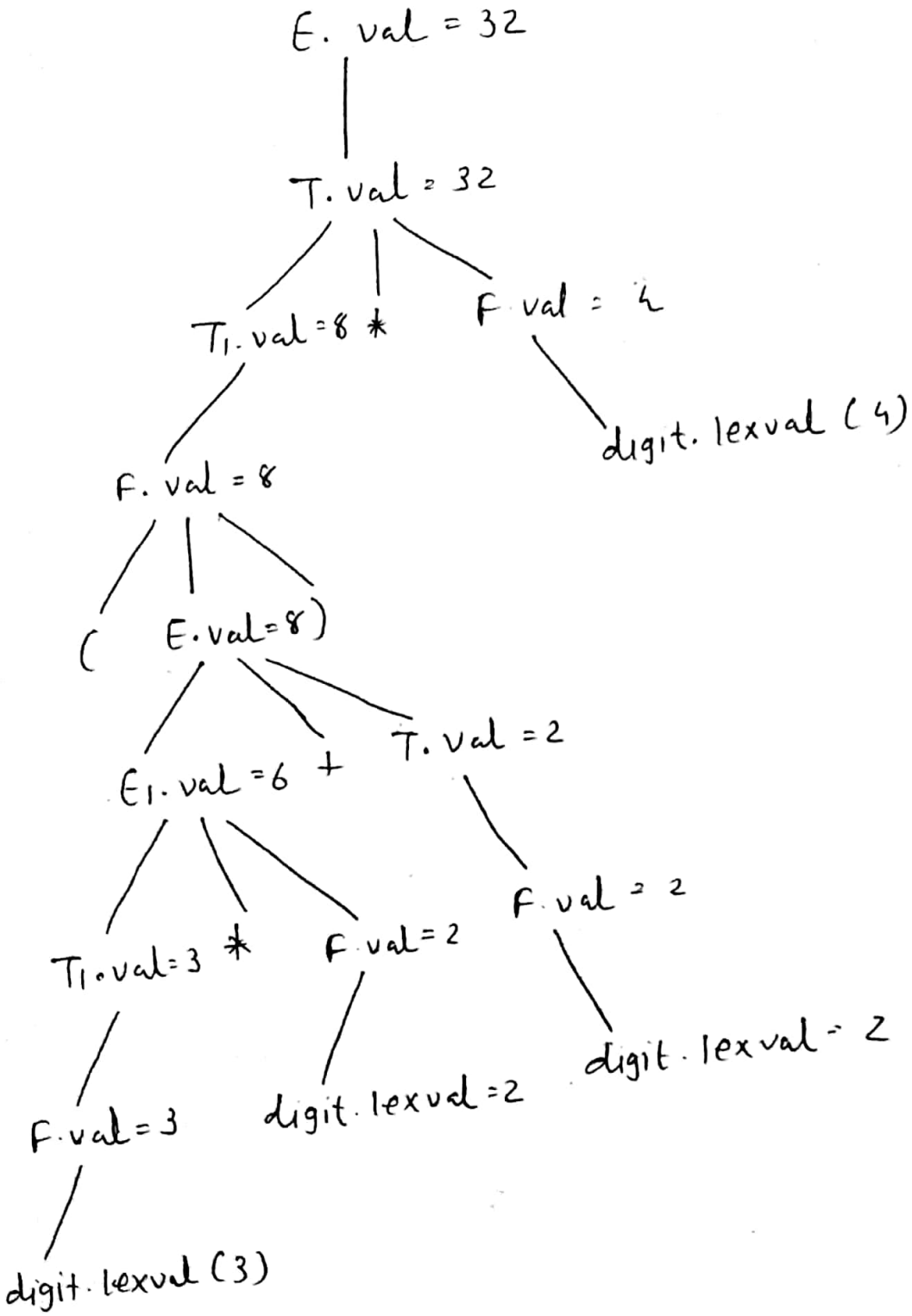
QB-4 Syntax directed definition for desk calculator. Using this definition draw annotated parse tree for 3*5+4n



Q8 Annotated parse tree for 4*3+2*5 n



Annotated parse tree for (3 * 2 + 2) * 4 :



Q-B 2 SDD with Inherited attributes for type declaration for list of identifiers.

Date:
Page No.

2 Inherited attribute :
The value of inherited attribute at a node in parse tree is defined using the attribute values at the parent or siblings.

Annotate the parse tree for the computation of inherited attributes for the given string:
int a, b, c ;

the grammar is as given below:

S → TL
T → int
T → float
T → char
T → double
L → L1, id
L → id

→ for the string int a, b, c we have to distribute data type int to all identifiers. a, b and c ;
Such that a becomes integer
b becomes integer
c becomes integer

following steps are to be followed:
1 Construct the Syntax-directed definition using semantic action.

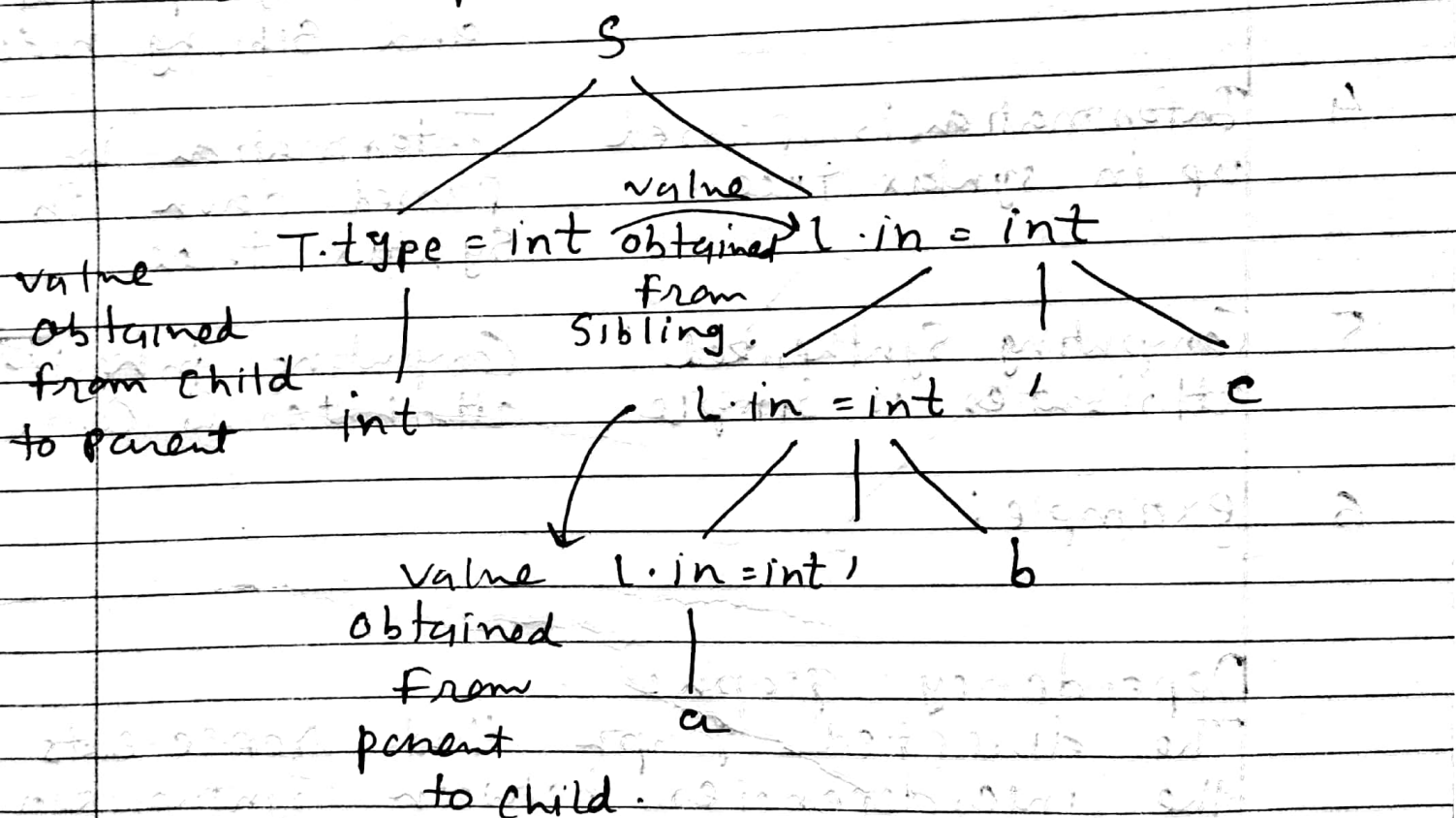
2 Annotate the parse tree with inherited attributes by processing in top down fashion.

Date: / /
Page No.

The Syntax directed definition :

Production Rule	Semantic action
S → TL	L.in := T.type
T → int	T.type := integer
T → float	T.type := float
T → char	T.type := char
T → double	T.type := double
L → L1, id	L1.in := L.in Enter-type (id.entry, L.in)
L → id	Enter-type (id.entry, L.in)

Annotated parse tree :



Q-B: Synthesized attribute vs Inherited attributes

Date
Page No.

3-B marks

	Synthesized translation	Inhesited translation
1	definition.	
2	In Synthesized translation the synthesized attributes are computed.	In Inhesited translation the attributes are computed
3	The Synthesized attributes are computed using values of children	The inhesited attributes are computed using values of parent and sibling nodes.
4	Information is passed up in syntax tree.	Information is passed down in Syntax tree.
5	Computing Synthesized attributes is simple.	Computing inhesited attributes is complex.
6	example:	

Dependency graph:
The directed graph that represents the interdepencies between Synthesized and inhesited attributes at nodes in the parse tree is called dependency graph.

Date: / /
Page No.

for the rule $X \to YZ$ the semantic action is given by $X.x := f(Y.y, Z.z)$ then Synthesized attribute is $X.x$ and $X.x$ depends upon attributes $Y.y$ and $Z.z$.

→

Ex: Dependency graph for the following grammar:

$E \to E_1 + E_2$
$E \to E_1 * E_2$

→ The Semantic rules for the above grammar is as given below:

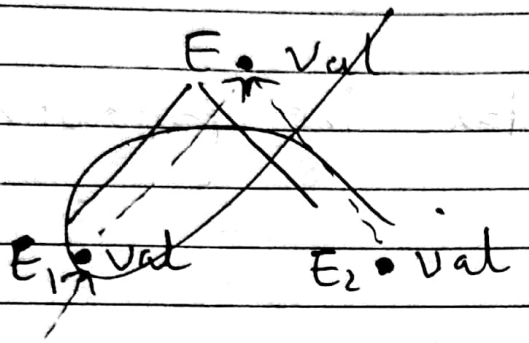
Production rule	Semantic rule.
$E \to E_1 + E_2$	$E.val := E_1.val + E_2.val$
$E \to E_1 * E_2$	$E.val := E_1.val * E_2.val$

The Synthesized attributes can be represented by .val. Hence the synthesized attributes are given by E.val, E1.val and E2.val. The dependencies among the nodes is given by solid arrows. The

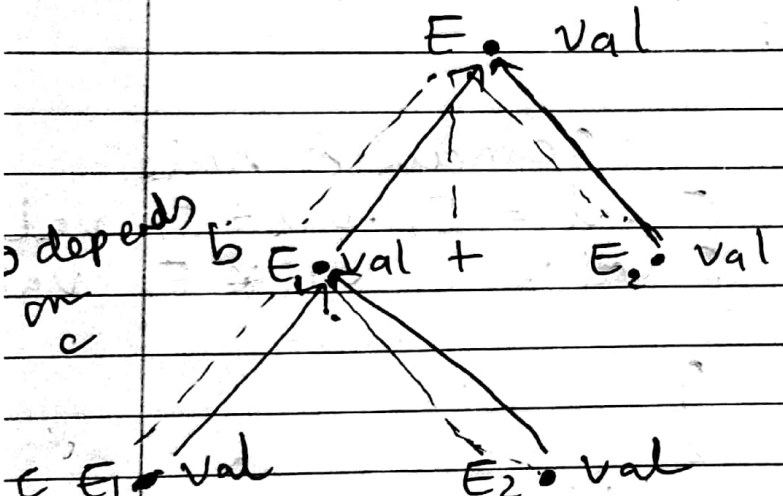
If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).

arrows from E_1 and E_2 show that value of E depends upon E_1 and E_2. We have represented parse tree using dotted lines.

dependency graph:



dependency graph:



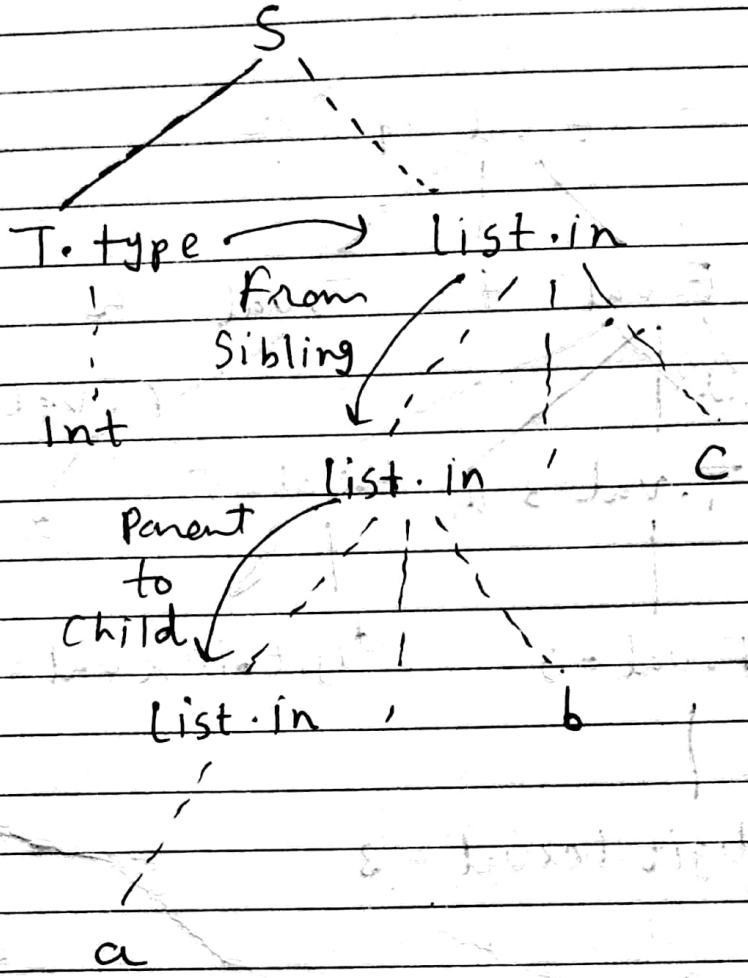
dependency rule: If an attribute depends from an attribute c, then we need to fire Semantic rule for c first and then the Semantic rule for b.

4. Evaluation order:
The topological Sort of the dependency graph decides the evaluation order in a parse tree. For deciding Evaluation Order in a parse tree. In deciding evaluation order the Semantic rules in the Syntax directed definitions are used.

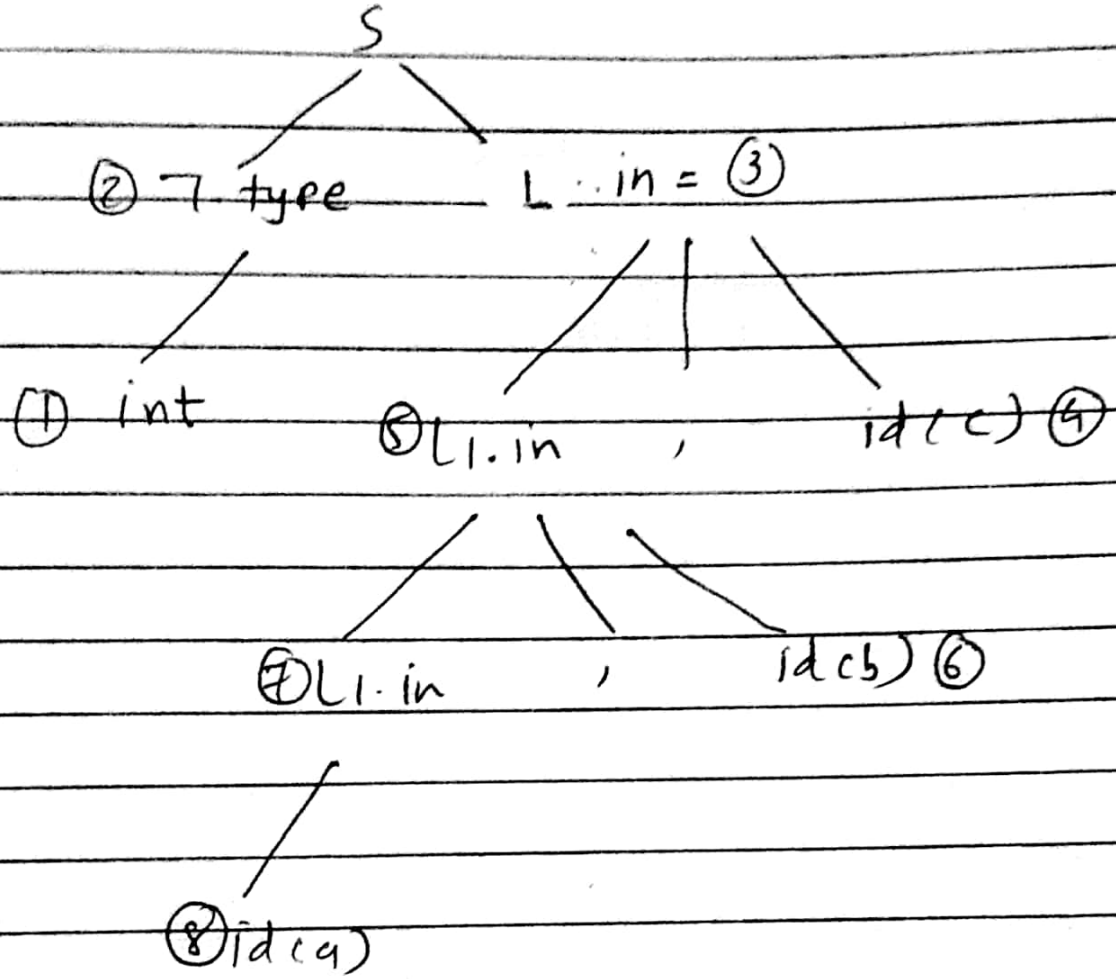
Topological Sort : any ordering $m_1, m_2, \ldots m_k$ such that if $m_i \rightarrow m_j$ is a link in dependency graph then $m_i < m_j$.

Thus the translation is specified by Syntax directed definitions. Therefore Precise definition of Syntax directed definition is required.

Dependency graph :



Evaluation order :



Date: / /
Page No.

L-attributed definition.

→

ex: $A \to PQ$

$P.in := P(A.in)$

$Q.in := q(P.sy)$

$A.sy := f(Q.sy)$

$A \to XY$

$Y.in := y(A.in)$

$X.in := x(Y.sy)$

$A.sy := f(X.sy)$

→ The attributes in and Sy represent the inherited and Synthesized attributes respectively.

→

Production rule	Semantic Action	Class of attribute
$A \to PQ$	$P.in := P(A.in)$	L-attribute
	$Q.in := q(P.sy)$	L-attribute
	$A.sy := f(Q.sy)$	L-attribute
$A \to XY$	$Y.in := y(A.in)$	L-attribute
	$X.in := x(Y.sy)$	Not L-attribute
	$A.sy := f(X.sy)$	L-attribute

Because here value of left Symbol (X) is dependent upon value of right Symbol.

L-attributed definition:

The Syntax directed definition can be defined as the L-attributed for production rule $A \rightarrow X_1 X_2 \cdots X_n$ where the inherited attribute X_k is such that $1 \leq k \leq n$. and it depends upon the attributes of the Symbol $X_1 X_2 \cdots X_{k-1}$ to the left of X_k.

This is because of the definition X.in := X(Y.sy). This semantic action suggests that value of X.in depends upon value of Y.sy. That means value of left symbol is dependent on the value of the right symbol. This violates the 1st rule of the L-attributed definition. Thus X.in : X(Y.sy) is not L-attributed.

→ Differentiate between S-attributed grammar and L-attributed grammar.

→

	S-attributed grammar	L-attributed grammar
1	This is a class of attributed grammar having no inherited attributes but only Synthesized attributes.	This is class of attributed grammar in which inherited attributes can be evaluated.
2	it can be incorporated in top down and bottom up parsing	2 The L-attributed grammar is incorporated in using top-down parsing.
3	YAcc family can be broadly Considered as S-attributed grammar.	3 Many programming languages are L-attributed. Special type of Compilers, the narrow Compilers are based on Some form of L-attributed grammar.

4	S-attributed grammar can be L-attributed grammars.	L-attributed grammar can not be S-attributed.

Page No.

5 The entry in the symbol table for identifier b gets associated with the type int. So b becomes integer type.

6 list.in is assigned the type int from the parent list.in.

7 The entry in the symbol table for id a gets associated with the type int. So a becomes the integer type.

Translation Scheme:
A translation scheme is a Context free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of productions.

→ Translation Scheme generates the output by executing the semantic actions in an ordered manner.

→ This processing is using depth first traversal.

Bottom-up evaluation of Inherited Attributes

Q-B: & Explain with an appropriate example how to perform bottom-up evaluation of inherited attributes.

→ Inherited Attributes can be handled by L-attributed definitions. So we will now discuss the method for bottom-up evaluation of inherited attributes.

→ The bottom up parser reduces the right side of production $X \to ABC$ by removing C, B and A from the parser stack.

→ parser stack is implemented as combination of state and value. The state[i] is for grammar symbol A and value[i] holds the Synthesized attribute A.a.

Ex: Translation Scheme :

$S \to TL$	{ list.in := T.type }
$T \to$ int	{ T.type := int }
$T \to$ float	{ T.type := float }
$L \to L_1$, id	{ L_1.in = L.in id.entry, L.in }
$L \to$ id	{ id.entry, L.in }

Consider the input int a, b, c for bottom-up evaluation of inherited attributes

Input string	State val	production
int a, b, c	-	
a, b, c	int	
a, b, c	T	T → int
, b, c	Ta	
, b, c	TL	L → id
b, c	TL,	
, c	TL, b	
, c	TL	L → L, id
c	TL,	
-	TL, c	
	TL	L → L, id
	S	S → TL

Q-1)

Date: / /
Page No.

Q.B 14. May-2012

Attributed grammar:

An attributed grammar is a context free grammar with attributes. Semantic rules and condition. Let G be an attribute gramm denoted by G = (N, Σ, P, S) where P is a set of production rules in the form $A \rightarrow \alpha$. These rules are associated with set of semantic rules of the form $b := f(a_1, a_2, a_3, \ldots a_n)$ where f denotes functionality.

ex: SDD for translating following grammar for postfix notation.

annotated parse tree for 9-5+2

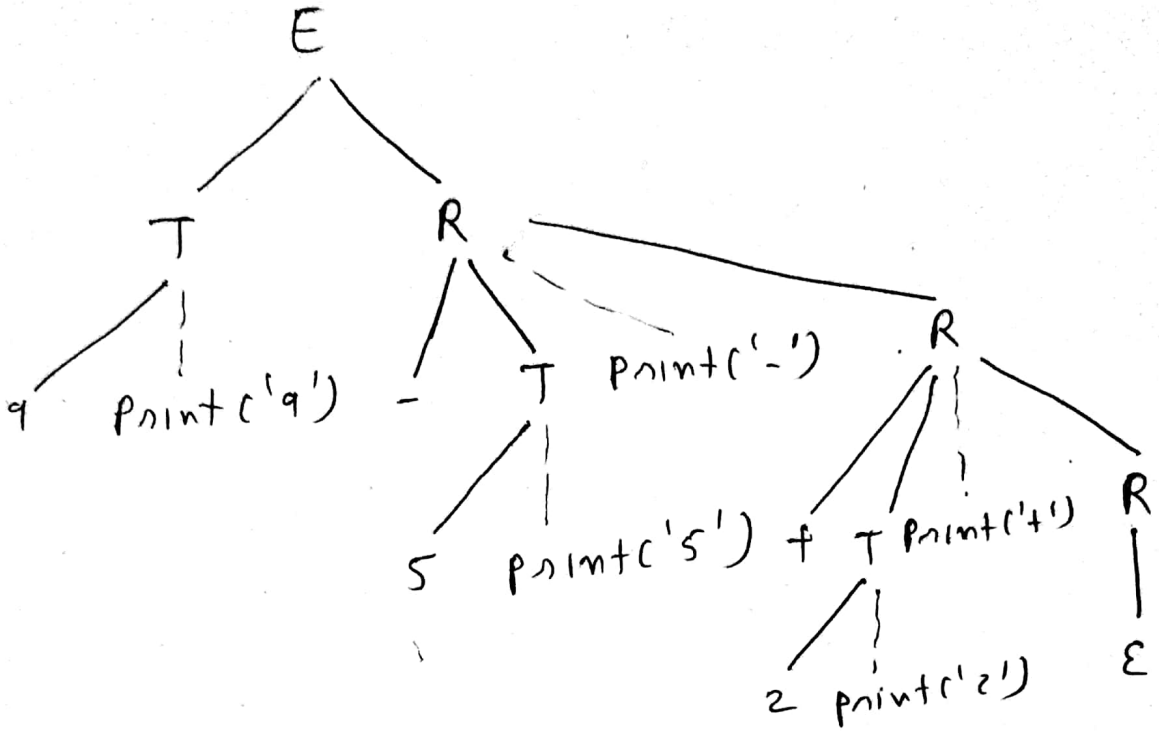
expr → expr + term

expr → expr - term | term

term → 0 | 1 | 2 | ..... | 9

Production Rule	Semantic Rule
E → TR	
T → 0	0 {print('0')}
T → 1	1 {print('1')}
T → 2	2 {print('2')}
T → 3	3 {print('3')}
T → 4	4 {print('4')}
T → 5	5 {print('5')}
T → 6	6 {print('6')}
T → 7	7 {print('7')}
T → 8	8 {print('8')}
T → 9	9 {print('9')}
R → + TR	+ {T print('+') R}
R → - TR	- {T print('-') R}
R → ε	ε

Annotated parse tree: 9-5+2



Translation Scheme for 3 * 4 + 5 * 2

$E \rightarrow E+T$

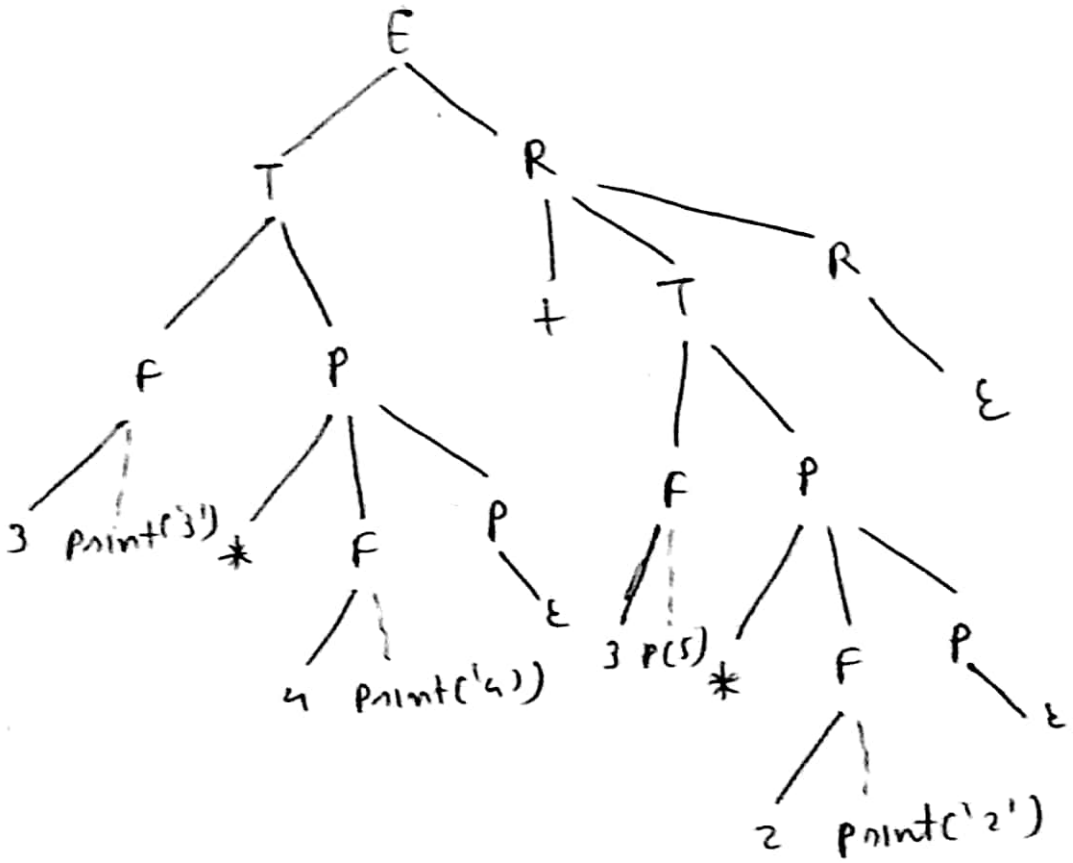
$E \rightarrow T$

$T \rightarrow T*F \mid F$

$F \rightarrow 0 \mid 1 \mid \dots \mid 9$

	Production rules	Semantic rules
1	$E \rightarrow TR$	
2	$R \rightarrow +TR$	+ { T { print('+') } R }
3	$R \rightarrow \varepsilon$ 4) $T \rightarrow FP$	ε
	$T \rightarrow *FP$	* { F print('*') P
	$T \rightarrow \varepsilon$	ε
	$F \rightarrow 0$	0 { print('0') }
	$F \rightarrow 1$	1 { print('1') }
	$F \rightarrow 2$	2 { print('2') }
	$F \rightarrow 3$	3 { print('3') }
	$F \rightarrow 4$	4 { print('4') }
	$F \rightarrow 5$	5 { print('5') }
	$F \rightarrow 6$	6 { print('6') }
	$F \rightarrow 7$	7 { print('7') }
	$F \rightarrow 8$	8 { print('8') }
	$F \rightarrow 9$	9 { print('9') }

Annotated parse tree : 3 * 4 + 5 * 2 .



QB-21 // syntax direction definition to produce three address code:

$S \rightarrow id := E$	id.name := E.place
$E \rightarrow E_1 + T$	E.place := newTemp() append(E.place = E_1.place + T.place)
$E \rightarrow T$	E.place : T.place
$T \rightarrow T_1 * F$	T.place = newTemp() append(T.place = T_1.place * F.place)
$T \rightarrow F$	T.place := F.place
$F \rightarrow id$	F.place := id.name.

Annotated parse tree for : x = a + b * c