4 Syntax Directed Translation

4.1 INTRODUCTION

A syntax-directed definition (SDD) is a context-free grammar with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

For example:

Ex. 1 $\underbrace{E \rightarrow E_1 + T}_{\text{Production}}$ $\underbrace{\{E.code = E_1.code \,||\, T.code\}}_{\text{Semantic rule}}$

Ex. 2 $\underbrace{E \rightarrow E_1 + T}_{\text{Production}}$ $\underbrace{\{print\ '+'\}}_{\text{Semantic action}}$

Ex. 3 $E \rightarrow E_1 \underbrace{\{print\ '+'\}}_{\text{Semantic action}} + T$

Note

- Position of the semantic action in the production body determines the order in which the action is executed.
- Value of each attribute at the parse tree node is defined by the semantic action for that node.

4.2 ATTRIBUTE

In grammar, every variable has its own attributes. We shall deal with two kinds of attributes for non-terminals:

a) Synthesised Attributes

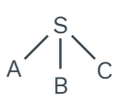
b) Inherited Attributes

Synthesised attributes:

They are those attributes where the value of the attribute depends upon the attribute value of their children.

For example:

$S \rightarrow ABC$ {S.val = A.val + B.val + C.val}

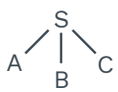


Here we can clearly see that value of attribute 'S' (S.val) depends upon its children attribute values.

Thus, we call 'S' as a synthesised attribute.

Inherited attributes:
They are those attributes where the value of attribute depends on the parent attribute value or siblings attribute values.

For example:
$S \rightarrow ABC$ {B.val = S.val + A.val + B.val}



Here we can clearly see that value of the attribute B depends upon its siblings attribute value, i.e., (A, C) and upon its parent attribute value which is (S).

Comparison between synthesised attributes and inherited attributes:

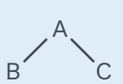
Synthesised Attributes		Inherited Attributes	
1)	Attribute value depends upon its children attribute values.	**1)**	Attribute value depends upon its sibling and parent attributes values.
2)	In the production left-hand, there must be a non-terminal.	**2)**	On the production right-hand side, there must be a non-terminal.
3)	Evaluation is done on the basis of bottom-up traversal of the parse tree.	**3)**	Evaluation is done on the basis of top-down or left to right traversal of the parse tree.
4)	These are used in both S-attributed as well as L-attributed SDT.	**4)**	These are used only in L-attributed SDT.
5)	Example: $F \rightarrow (E)$ {F.val = E.val} F ↓ E Thus parent value depends on children. ∴ F is a synthesised attribute.	**5)**	Example: $A \rightarrow BC$ {B.val = A.val * C.val}  The value of B depends on sibling C and parent A ∴ B is an inherited attribute.

Table 4.1

4.3 CONSTRUCTION OF SYNTAX TREE

Syntax tree:

Definition

A tree in which internal nodes consist of operators and leaf nodes consist of operands is known as a syntax tree.

- In order to construct a syntax tree mainly three functions are used, which are as follows:
  1) Make-node (op, left, right)
  2) Make-leaf (id, entry to symbol table)
  3) Make-leaf (num, value)
- Make-node function is used to create a node with an operator where 'op' represents the operator, left represent the left child address, and right represents the right child address.
- Make-leaf function is used to create a node with an identifier. In this function, 'id' represents identifier, and entry to the symbol table is a pointer that contains the address of the symbol table where this identifier will be stored.
- Make-leaf function is also used to create a node with constant values. In this function, value represent the constant.

SOLVED EXAMPLES

Q.1 **Construct a syntax tree for expression 7*x+4-z.**

Sol: **Functions needed for construction of syntax tree**

Symbol	Function
7	P_1 = make-leaf (num, 7)
x	P_2 = make-leaf (x, pointer to symbol table)
4	P_3 = make-leaf (num, 4)
z	P_4 = make-leaf (z, pointer to symbol table)
*	P_5 = make-node (*, P_1, P_2)
+	P_6 = make-node (+, P_5, P_3)
–	P_7 = make-node (–, P_6, P_4)

Table 4.2

By calling the above functions, syntax tree will be created as follows:

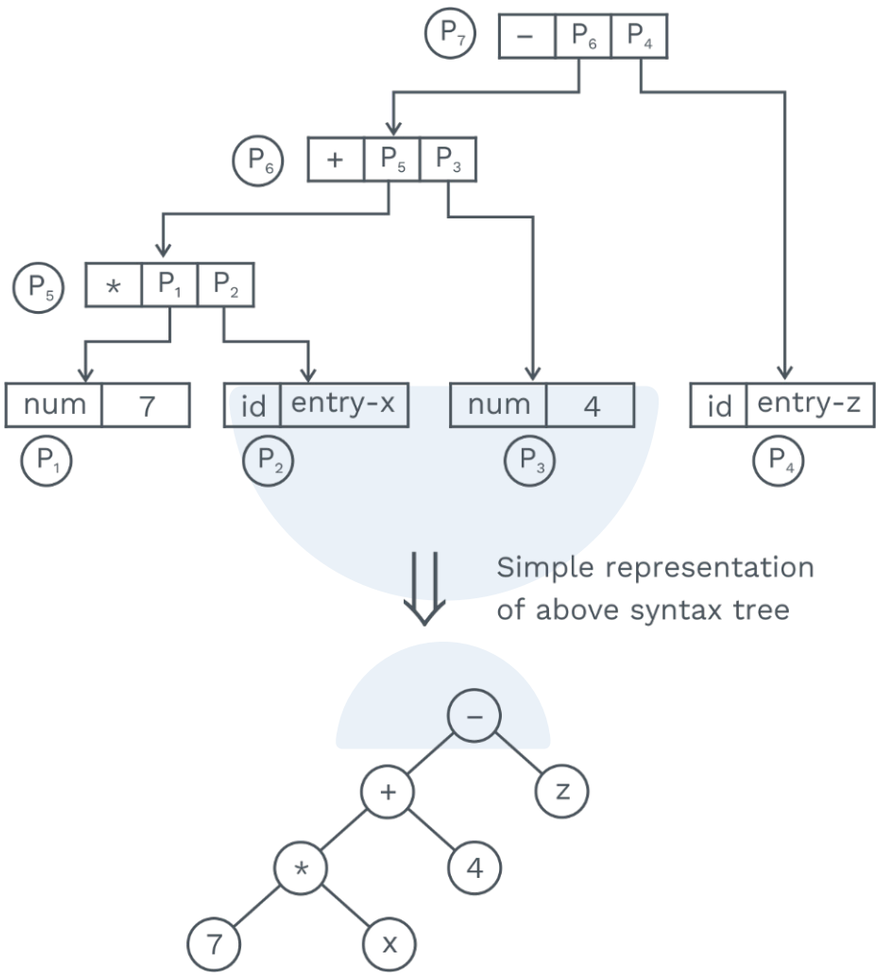


Fig. 4.1

Note

- First of all, all the functions for the construction of leaf node will be called.
- Then, for the construction of internal nodes, the functions with the highest precedence operators are called.

SOLVED EXAMPLES

Q.2 **Construct a syntax tree for the expression x = 4 + y * z.**

Sol: **Functions to be called for the syntax tree construction**

Symbol	Function
x	P_1 = make-leaf (x, pointer to symbol table)
4	P_2 = make-leaf (num, 4)
y	P_3 = make-leaf (y, pointer to symbol table)
z	P_4 = make-leaf (z, pointer to symbol table)
*	P_5 = make-node (*, P_3, P_4)
+	P_6 = make-node (+, P_2, P_5)
=	P_7 = make-node (=, P_1, P_6)

Table 4.3 Functions

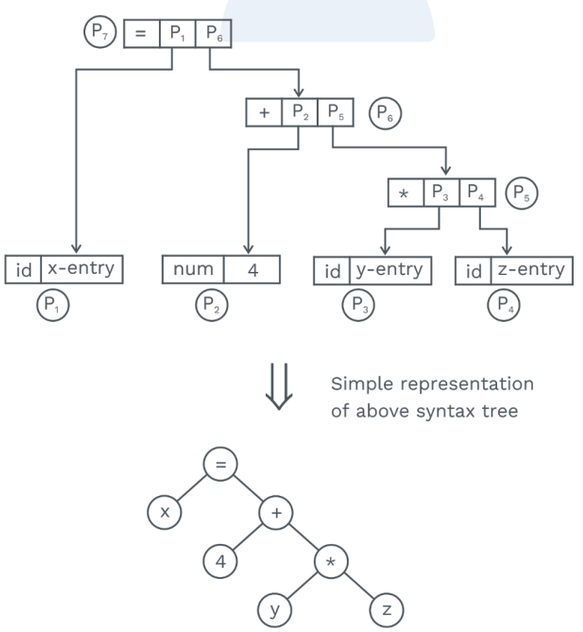


Fig. 4.2 Tree

Q.3 **Construct a syntax tree for expression a – 4 + 6**

Sol: **Functions to be called for the syntax tree construction**

Symbol	Function
a	P_1 = make-leaf (id, entry-a)
4	P_2 = make-leaf (num, 4)
6	P_3 = make-leaf (num, 6)
–	P_4 = make-node (–, P_1, P_2)
+	P_5 = make-node (+, P_3, P_4)

Table 4.4 Functions

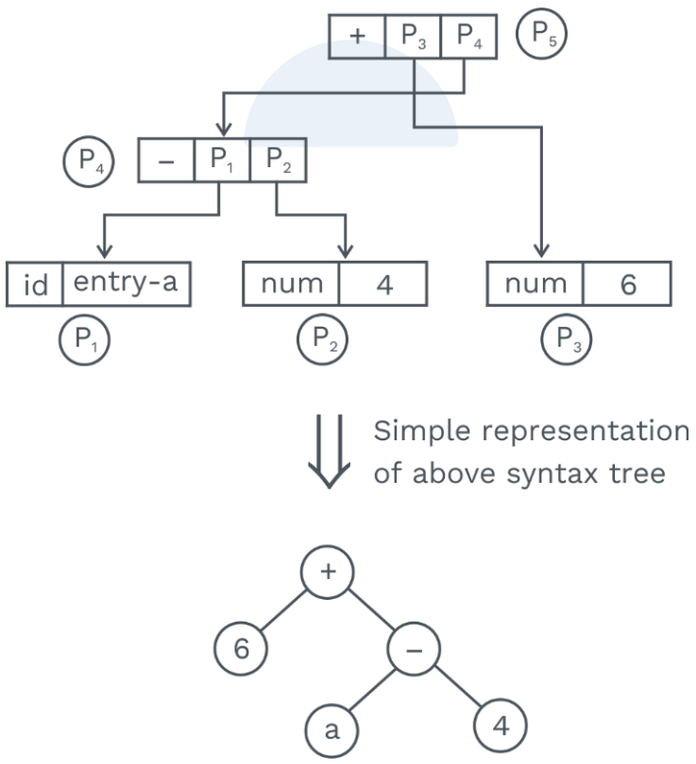


Fig. 4.3 Tree

Syntax directed definition for construction of a syntax tree:
Given below productions and semantic actions for the syntax tree construction of an expression consisting of '*' and '÷' operator. For calling functions make-leaf and make-node productions of grammar are used.

Production	Sematic Function
1) $E \rightarrow E_1 * T$	E.node = make-node (*, E_1.node, T.node)
2) $E \rightarrow E_1 \div T$	E.node = make-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 = make-leaf (id, id-entry)
6) $T \rightarrow num$	T.node = make-leaf (num, value)

Table 4.5 Semantic Functions

- '*' and '÷' are usually at the same precedence level and are jointly left associative.
- All non-terminals are synthesised attribute nodes which represent the node of syntax tree.
- When first production $E \rightarrow E_1 * T$ is used, its action creates a node with '*' for operator and two children, E_1.node and T.node .
- Second production has a similar semantic action.
- For production $E \rightarrow T$ no node is created. Since E.node is the same as T.node.
- Similarly, no node is created for production $E \rightarrow (T)$ value of T.node is same as E.node. Parentheses are for grouping, they influence the structure of the parse tree and syntax tree, but once their job is done there is no need to retain them in the syntax tree.
- Last two T-productions have a single terminal on the right; we use constructor make-leaf to make a suitable node which becomes the value of T.node.
- The syntax tree for input "x ÷ 4 * y" to the above SDD

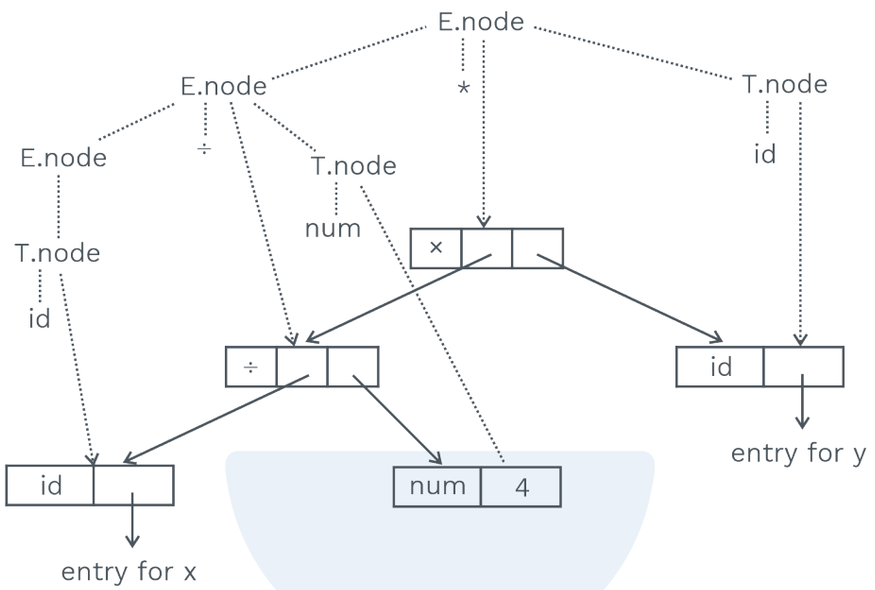


Fig. 4.4 Tree

Syntax tree for x ÷ 4 * y is shown above by solid lines.

Underlying parse tree which need not be constructed as shown with dotted edges.

- Steps in the construction of syntax tree for "x ÷ 4 * y"

1) S_1 = make-leaf (id, entry x)
2) S_2 = make-leaf (num, 4)
3) S_3 = make-node (÷, S_1, S_2)
4) S_4 = make-leaf (id, entry y)
5) S_5 = make-node (*, S_3, S_4)

SOLVED EXAMPLES

Q.4 **Construct syntax tree to execute the given arithmetic expression**
Input: 10 – 5 ÷5
Output: 9

Note

- Steps for creating syntax tree:
- Write unambiguous grammar along with semantic action
- Build tree

Sol:

$X \rightarrow X_1 - T$	**{X.val = X_1.val – T.val}**
$X \rightarrow T$	{X.val = T.val }
$T \rightarrow T_1 \div F$	{T.val = T_1.val ÷ F.val}
$T \rightarrow F$	{T.val = F.val }
$F \rightarrow id$	{F.val = id}

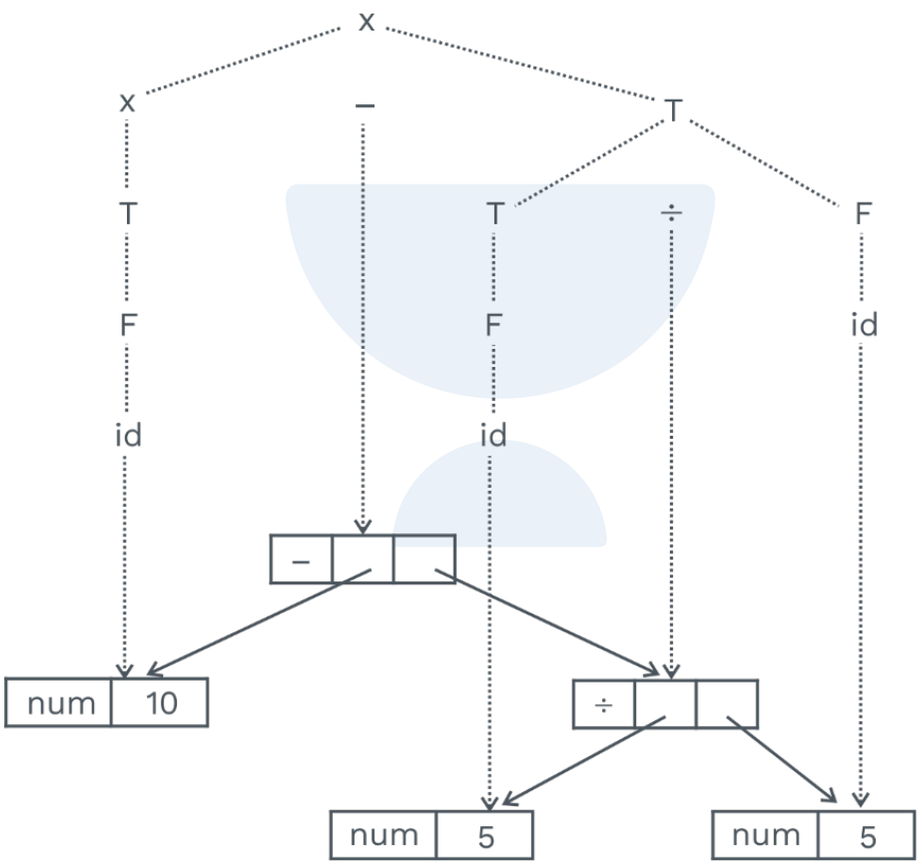


Fig. 4.5 Tree

Note

- In this example we have designed our grammar in such a way that the output should be 9.
- For getting output as 9 we have given higher precedence to '÷' operator and then '–' operator.

Q.5 **Given a SDT:**

X → YX {Print 1}

X → Y {Print 3}

X → ZX {Print 4}

X → Z {Print 9}

Y → aZ {Print 2}

Y → a {Print 5}

Z → bZ {Print 7}

Z → b{Print 8}

If the input is "abba", then what will be the output?

Sol: **Parse tree formed for the given input "abba"**

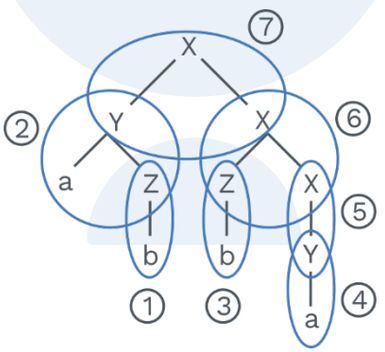


Fig. 4.6 Tree

Evaluation of parse tree will be in post order.
Thus output:

1	2	3	4	5	6	7
↓	↓	↓	↓	↓	↓	↓
8	2	8	5	3	4	1

Therefore, when the input is 'abba', output will be 8285341.

Q.6 **Consider the following SDT:**

A → A$B {A.val = A_1.val + B.val}

A → B {A.val = B.val}

B → B#F {B.val = _____}

B → F {B.val = F.val}

F → num {F.val = num}

If the input is: 7#2$3#5$4 and for the given input, the output is 33. Then what will be the missing semantic action?

a) B.val = B.val + F.val **b) B.val = B.val * F.val**

c) B.val = B.val – F.val **d) None of these**

Sol:

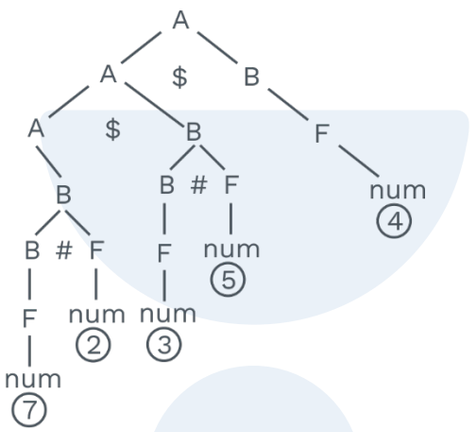


Fig. 4.7 Tree

From the parse tree, it is clear that # has higher precedence than $ because for evaluation, we use post order traversal and thus, first of all, all the '#' operations will be performed.

From semantic actions it is clear that '$' is a plus operation.

Now looking at options and evaluating the output.

Option (a): B.val = B.val + F.val

According to this # = +

$$\therefore 7 + 2 + 3 + 5 + 4 = 21 \neq 33$$

Wrong option

Option (b): B.val = B.val * F.val

$$\therefore \# = *$$

Thus, 7 * 2 + 3 * 5 + 4

As we know for parse tree # has higher precedence

$\therefore$ = 14 + 15 + 4 = 33 = output

$\therefore$ Missing semantic action is B.val = B.val * F.val

Detection of associativity of operator in SDT:

For directly checking the associativity of an operator from the syntax-directed translation, we have to see whether a new variable is coming right side of the operator or left side of the operator in the production.

If a new variable is coming right side of the operator, then its associativity is left to right.

If a new variable is coming left side of the operator, then its associativity is from right to left.

For e.g.

$$A \rightarrow A * B \mid B$$

$$B \rightarrow C + B \mid C$$

$$C \rightarrow id$$

In the given grammar, in production $A \rightarrow A * B$ new variable is coming on right side of the operator thus, '*' is left-associative.

In the production $B \rightarrow C + B$ new variable 'C' is coming on the left side of the production thus associativity of '+' is right to left.

SOLVED EXAMPLES

Q.7 **Given SDT:**

$X \rightarrow X * Z$ {X.val = X_1.val * Z.val}

$X \rightarrow Z$ {X.val = Z.val}

What is the associativity of '*' operator ?

Sol: **In production $X \rightarrow X * Z$ new variable is coming on right-hand side of the operator.**

$\therefore$ Associativity of * is left to right (left associative).

Previous Years' Question

In the following grammar:

$X \rightarrow X \oplus Y \mid Z$

$Y \rightarrow Z \odot Y \mid Z$

$Z \rightarrow id$

Which of the following is true?

a) $\oplus$ is left-associative while $\odot$ is right-associative.

b) Both $\oplus$ and $\odot$ is left-associative.

c) $\oplus$ is right-associative while $\odot$ is left-associative.

d) None of these

Sol: a) **[GATE-CS-1997]**

Checking associativity and precedence of the operator in parse tree:

- During the evaluation of a parse tree, the operator which is evaluated first have higher precedence.
- During the evaluation of parse tree, for a particular operator, if the right-most evaluation is done first, then it is right-associative; if the left-most operation is done first; then it is left-associative.

SOLVED EXAMPLES

Q.8 **Given a syntax tree:**

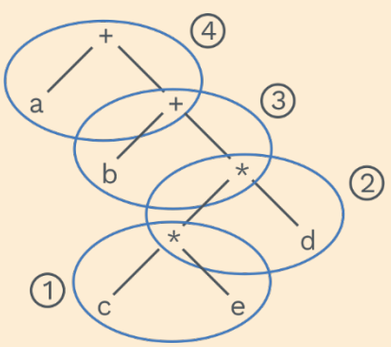


Fig. 4.8 Tree

What will be the associativity and the precedence order of the operators?

Sol:

- Expression derived from syntax tree: "a + b + c * e * d"
- By applying post order evaluation :

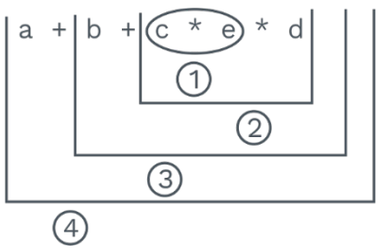


Fig. 4.9 Associativity

c * e is evaluated first, and after that (c * e) * d from this it is clear that '*' operator is having higher precedence then that '+' operator and associativity of * is from left to right because 'c * e' is evaluated first.

The Precedence of '+' operator is less than '*'

Associativity of '+' operator is from right to left because the right side '+' is evaluated first.

Q.9 **Given a syntax tree:**

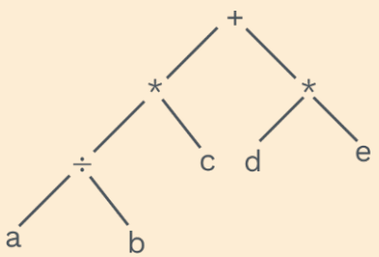


Fig. 4.10 Tree

What will be the associativity and precedence of +, *, ÷ operators?

Sol: **An Expression derived from syntax tree:**

a ÷ b * c + d * e

Now applying post order evaluation on the syntax tree.

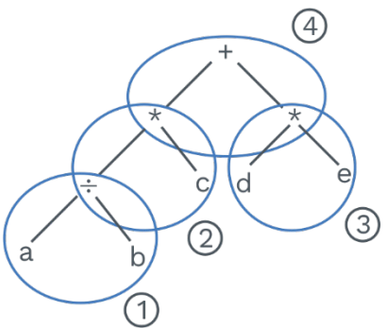


Fig. 4.11 Tree

So, order of evaluation,

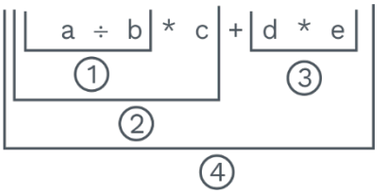


Fig. 4.12 Associativity

Therefore '÷' operator has the highest precedence, and '+' operator has the lowest precedence.

÷ > * > + {precedence order}

- Associativity of * operator is left to right associativity. The associativity of '+' and '÷' operators cannot be told because they are used only once in the expression.

Previous Years' Question

Consider the following parse tree for the expression a # b $ c $ d # e # f involving two binary operators # and $.

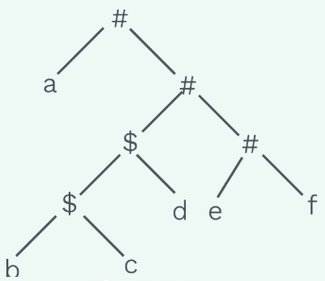


Fig. 4.13 Tree

Which one of the following is correct for the given parse tree?

a) $ has higher precedence and is left-associative; # is right-associative.
b) # has higher precedence and is left-associative; $ is right-associative.
c) $ has higher precedence and is left-associative; # is left-associative.
d) # has higher precedence and is right-associative; $ is left-associative.

Sol: a) **[GATE: CS-2018]**

4.4 TYPES OF SDT

Depending upon the type of attribute syntax directed translations are of two types:

1) S-attributed SDT

2) L-attributed SDT

Rack Your Brain

Terminals of any grammar can have synthesised attributes but not inherited attributes. Think why ?

S-attributed SDT:

- In these types of SDT, only synthesised attributes are present.
- For evaluation of S-attributed SDT, bottom-up execution, also known as post order evaluation is used.
- In these types of SDT, semantic actions will always be present on the right-hand side of the production at the right-most places.

Examples:

Ex 1: A $\rightarrow$ BCD {A.val = B.val + C.val}

Given SDT is S-attributed because A is a synthesised attribute and semantic action is present on the right-hand side, right-most place.

Ex 2: A $\rightarrow$ B {A.val = B.val}CD

Given SDT is not S-attributed because semantic action is present on the right-hand side in between the production.

Ex 3: A $\rightarrow$ BCD {C.val = A.val + D.val}

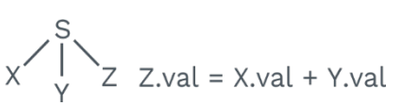
Given SDT is not S-attributed because attribute 'C' is an inherited attribute because the value of 'C' depends on the parent and sibling.

L-attributed SDT:

- In these types of SDT, both synthesised and inherited attributes are present.
- If inherited attributes are present, then that attribute can depend only upon the parent or left sibling. It should not depend on the right sibling.
- For evaluation of L-attributed SDT left to right evaluation is done.
- In this type of SDT, semantic actions can be present anywhere on the right-hand side of the production.

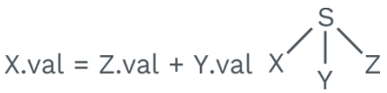
Examples:

Ex 1: S $\rightarrow$ XYZ {Z.val = X.val + Y.val}



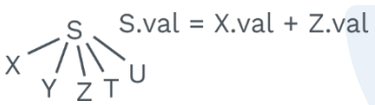
Given SDT is L-attributed because Z is an inherited attribute which depends on left sibling values X and Y.

Ex 2: S → XYZ {X.val = Y.val + Z.val}



Given SDT is neither L-attributed nor S-attributed the because attribute X is an inherited attribute which depends on the right sibling.

Ex 3: S → XYZ {S.val = X.val + Z.val}TU



Given SDT is L-attributed because attribute S is synthesised attribute and semantic action is present on the right-hand side in middle.

Rack Your Brain

Do you think all the S-attributed STD will also of type L-attributed. If yes then what is the reason ?

Comparison between L-attributes and S-attributes:

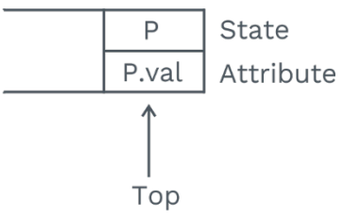
L-attributes		S-attributes	
1)	It consists of both synthesised and inherited attribute, but inherited attribute should not depend on the right sibling.	**1)**	It only consists of synthesised attribute that does not contain inherited attribute.
2)	For evaluation depth-first, the left to right evaluation approach is used.	**2)**	For evaluation bottom up approach is known as post order evaluation is used.
3)	Semantic actions are present anywhere on right-hand side of the production.	**3)**	Semantic action can be present at rightmost place on the right-hand side of the production.
4)	Every L-attributed can not be S-attributed.	**4)**	Every S-attributed is L-attributed.

Table 4.6 Comparison

4.5 BOTTOM-UP EVALUATION OF INHERITED ATTRIBUTES

- Bottom-up evaluation of inherited attributes can be done with the help of stack.
- Attribute of each grammar symbol can be put on the stack where they can be found during the reduction.
- Parser stack contains a record with a field of grammar symbol (or present state) and below it is a field for the attribute.
- For eg: There is a production P → QRS

- Grammar symbols SRQ are on the top of stack, so they are about to be reduced by the production P → QRS.
- After reduction top of the stack will be pointing at position 'Q' and in place of Q, there will be P, and in place of Q.val, there will be P.val.



- In inherited attribute, attribute value depend on the parent as well as on the sibling. We can apply bottom-up evaluation for that inherited attributes where siblings or parents are already present on the stack.
- For eg: P → QR {R.val = Q.val}

Suppose Q is an inherited attribute; its value is present on the stack, so R will use its value, and we can apply the bottom-up approach easily.

- In case if the production P → QR {R.val = Q.val + P.val}

Q.val is present in the stack, but P.val is not present in the stack. Then, in this case bottom-up evaluation can not be applied; we are forced to use depth-first evaluation.

Previous Years' Question

In a bottom-up evaluation of a syntax directed definition of inherited attributes can

a) Always be evaluated
b) Be evaluated only if the definition is L-attributed
c) Be evaluated only if the definition has synthesised attribute
d) Never be evaluated

Sol: c) **[GATE; CS-2003]**

SOLVED EXAMPLES

Q.10 **Given a SDT**

X → YZ {Z.in = Y.type}

Y → int {Y.type= int }

Y → Float {Y.type = float}

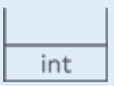
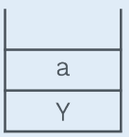
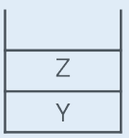
Y → real {Y.type = real}

Z → Z_1, id {Z_1.in = Z.in} {add type (id.entry, Z.in)}

Z → id {add type (id.entry, Z.in}

Input to SDT is "int a,b,c". Then show the implementation of the bottom-up evaluation.

Sol:

Input	Stack	Production
Int a, b, c	–	–
a, b, c		–
a, b, c		Y → int
, b, c		–
, b, c		Z → id

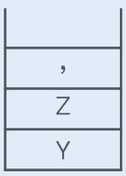
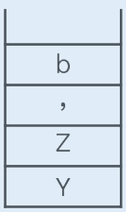
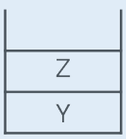
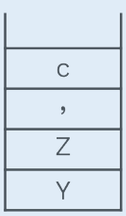
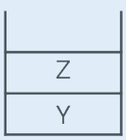
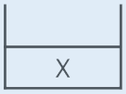
Input	Stack	Production
b, c		–
, c		–
, c		$Z \rightarrow Z_1$, id
c		–
–		–
–		$Z \rightarrow Z_1$, id
–		$X \rightarrow YZ$

Table 4.7 Implementation Using Stack

From the bottom, using productions and reducing them in the stack, we reach to head X.
This shows that "int a, b, c" is accepted by the given SDT.

Note

- During inherited attribute evaluation, semantic action may be present in between the production variables. In such case, semantic action can be replaced by non-terminal pointing to E.
  For e.g: A → B {print '+'}CD
  This can be converted simply as
  A → BMCD
  M → E {print '+'}

Q.11 **Construct SDT to create a syntax tree for the given arithmetic expression.**
I/P : Z = 10 + 2 * 5
Output:

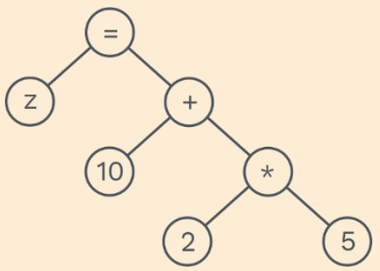


Fig. 4.14

Sol: **To construct SDT, we use a function make-node in the semantic action.**
Make-node (left-pointer, operator, right-pointer) is a function which returns the node address.

P → T = R	{P.ptr = make-node (T.ptr, =, R.ptr) and return(P.ptr)}
R → R + S	{R.ptr = make-node (R_1.ptr, +, S.ptr)}
R → S	{R.ptr = S.ptr}
S → S * Q	{S.ptr = make-node (S_1.ptr, *, Q.ptr)}
S → Q	{S.ptr = Q.ptr}
Q → num	{Q.ptr = make-node (null, num, null}
T → id	{T.ptr = make-node (null, id, null}

P.ptr, R.ptr, S.ptr, T.ptr and Q.ptr are the pointers that store the address of the node.

Parse Tree frame the above SDT

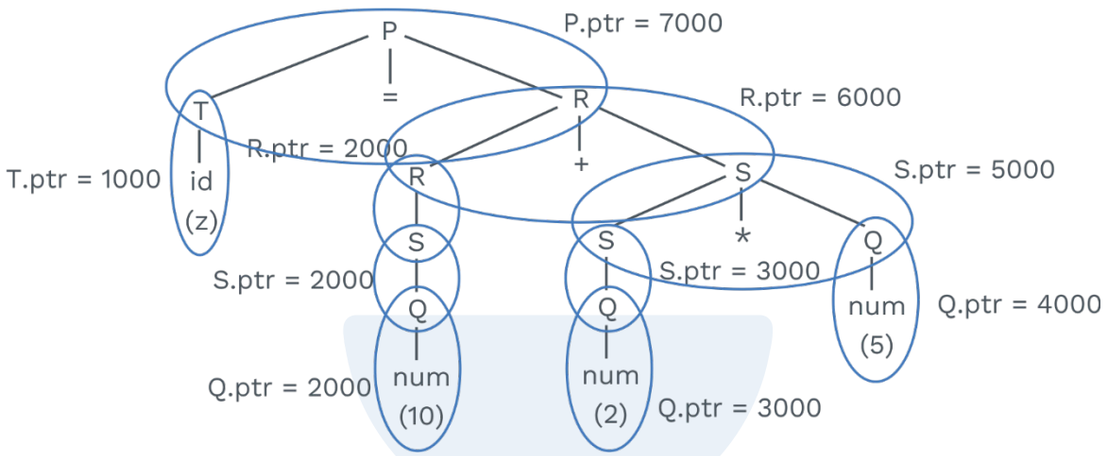


Fig. 4.15 Tree

1000, 2000, 3000, 4000, 5000, 6000, 7000; we have assumed that these are the addresses of the node returned by the make-node function after the formation of the node to the respective pointers.

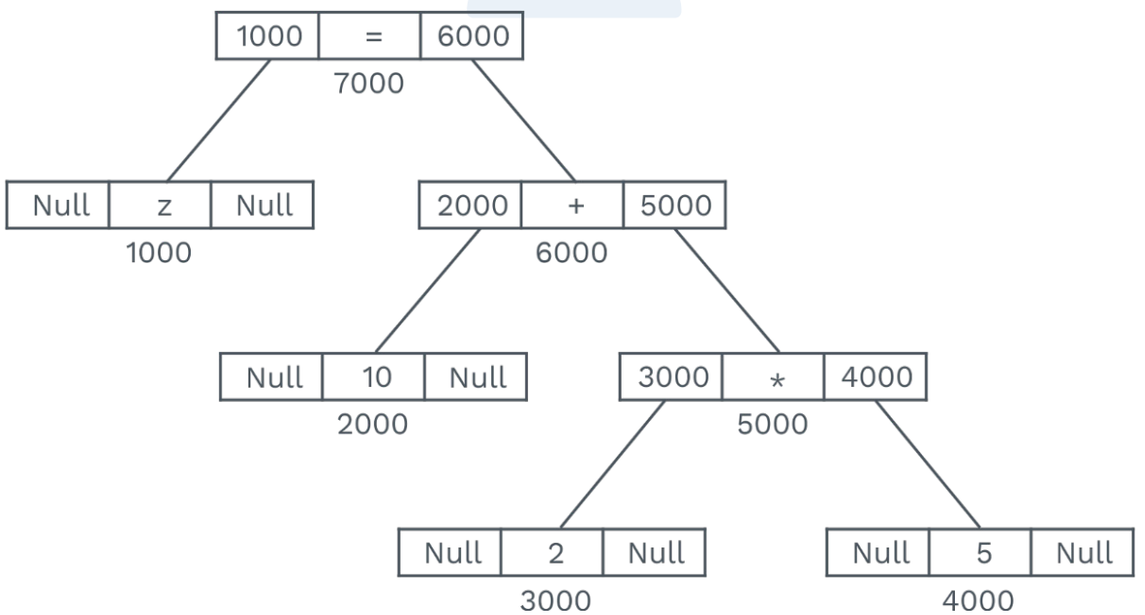


Fig. 4.16 Tree

Q.12 **Create syntax-directed translation for conversion of expression from infix to postfix.**
Input: a = p + q * r – s ÷ t
Output: xpqr*+st÷ – =

Sol:

P → Q = R	**{print (=)}**
R → R + S	{print (+)}
R → R – S	{print (–)}
R → S	{no semantic action}
S → S * Q	{print (*)}
S → S ÷ Q	{print (÷)}
S → Q	{no semantic action}
Q → id	{print (id)}

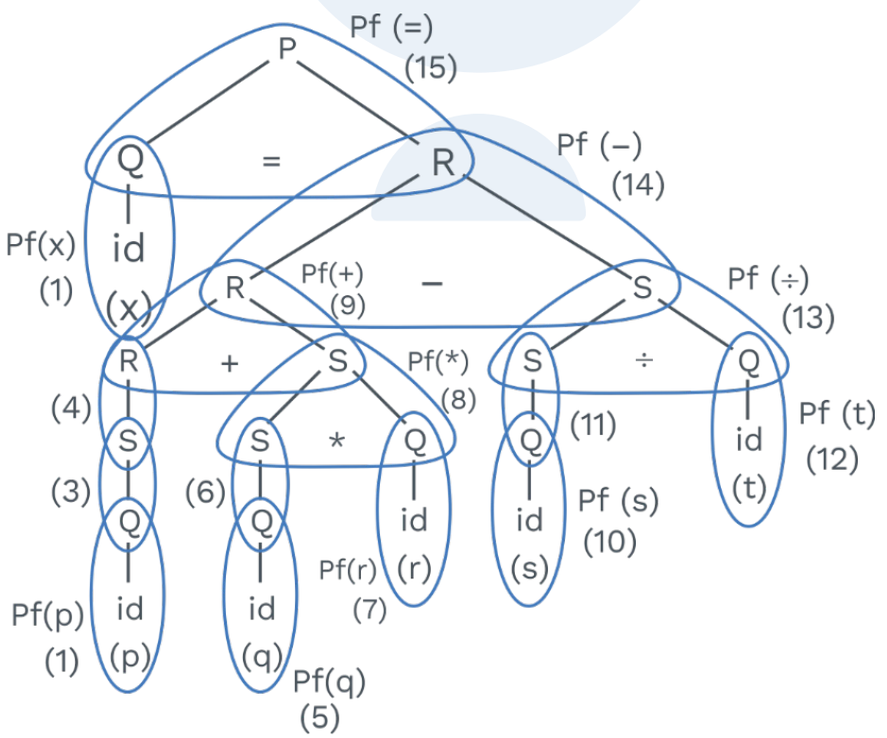


Fig. 4.17 Tree

Previous Years' Question

Consider the syntax directed definition shown below:

$S \rightarrow$ id: = E {gen (id.place = E.place;);}

$E \rightarrow E_1 + E_2$ {t = newtemp ( ); gen(t = E_1.place + E_2.place;); E.place = t}

$E \rightarrow$ id {E.place = id.place;}

Here ; gen is a function that generates the output code, and newtemp is a function that returns the name of a new temporary variable on every call. Assume that is t_1 are the temporary variable name generated by newtemp.

For the statement 'X : = Y + Z' the 3-address code sequence generated by this definition is.

a) X = Y + Z

b) t_1 = Y + Z ; X = t_1

c) t_1 = Y ; t_2 = t_1 + Z ; X = t_2

d) t_1 = Y ; t_2 = Z ; t_3 = t_1 + t_2 ; X = t_3

Sol: b) **[GATE: CS-2003]**

4.6 EVALUATION ORDERS FOR SDD's

- For evaluating the order of the attribute instance in a given parse tree dependency graph is a useful tool.
- While an annotated parse tree shows the values of attribute dependency graph helps in determining how those values can be computed.

Dependency graph:

Definitions

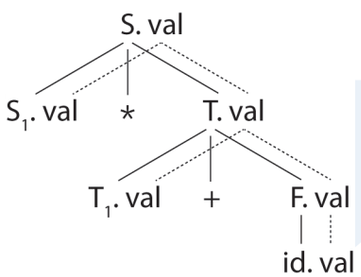
A dependency graph depicts the flow of information among the attribute instances in a particular parse tree, an edge from one attribute instance to another means that the value first is needed to be computed the second.

SOLVED EXAMPLES

Q.13 **Find the grammar and corresponding semantic rules to construct the dependency graph accordingly.**

$S = S_1 * T$; $S.val = S_1 val * T.val$
$T = T_1 + F$; $T.val = T_1.val + F.val$
$F = id$; $F.val = id.val$

Sol:



Note

- As a convention, we shall show the parse tree edges as dotted lines while the edges of the dependency graph with solid lines.

Ordering the evaluation of attributes:

- The order of the evaluation of the attributes at every node in the parse tree is characterized using a dependency graph.
- Topological sort is applied on the dependency graph to get the order of evaluation of attributes.
- In topological sort, first we choose the node whose in-degree is zero and remove it from the dependency graph and again select one node whose in-degree is zero.
- In this manner, we get a sequence of nodes and that sequence tells the order of evaluation of attributes in the SDT's.

SOLVED EXAMPLES

Q.14 **Given a dependency graph:**

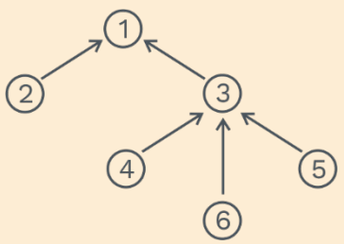


Fig. 4.18 Tree

Which among the following is topological order of given graph.

a) 2, 4, 6, 5, 3, 1 **b) 6, 3, 5, 4, 2, 1**

c) 2, 4, 5, 6, 3, 1 **d) 5, 4, 6, 2, 3, 1**

Sol: **Option a, c, d**

Option b) : 6, 3, 5, 4, 2, 1

We can remove node '6' first because it has a degree as 0.

But after removal of 6, we can not remove node 3 because it has in edge from node 4 and node 5.

Thus option b) is not correct.

Q.15 **Given a dependency graph:**

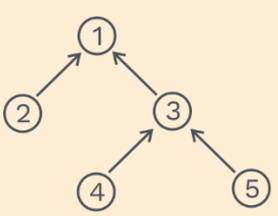


Fig. 4.19 Tree

How many orders of evaluations are possible ?

Sol: **Total number of evaluation orders is equal to the total number of topological order possible on the dependency graph.**

Following are the topological order possible on the dependency graph.

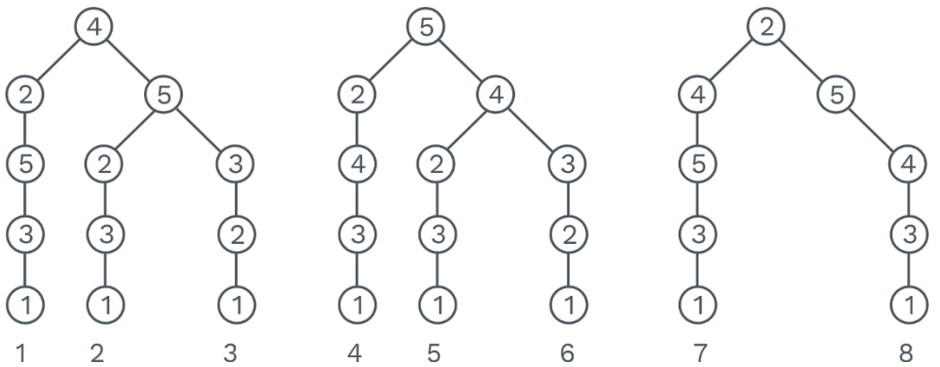


Fig. 4.20 Tree

Therefore total 8 evaluations order are possible.

Q.16 **Given a SDT:**

P → P = Q	**{P.val = Q.val}**
Q → Q + R	**{Q.val = Q_1.val + R.val}**
Q → R	**{Q.val = R.val}**
R → R * P	**{R.val = R.val * P.val}**
R → P	**{R.val = P.val}**
P → id	**{P.val = id}**

Input : x = u * v + w * t
Draw a parse tree and its dependency graph. {Operators have their usual precedence and associativity}

Sol:

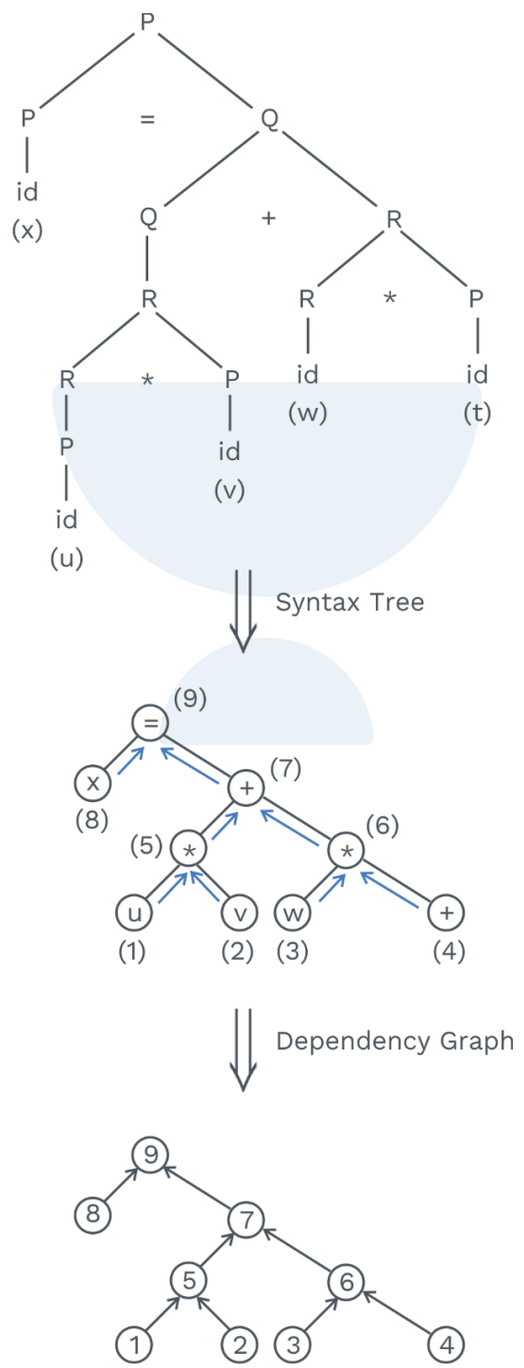


Fig. 4.21 Tree

Order of evaluation = Topological sort on the dependency graph.
= 8 1 2 5 3 4 6 7 9

Note

- There can be more than one topological order thus, the order of evaluation is not unique.
- Topological order equal to post-order traversal is closest to programming.
- If a cycle is present in the dependency graph, then there will be no evaluation order for that SDT.

Chapter Summary

- Syntax directed definition (SDD) is context-free grammar together with attributes and rules.
- Attributes are of two types:
  i) Synthesised attributes: Depend upon children.
  ii) Inherited attributes: Depend on siblings and parents.
- Syntax tree is a tree in which internal nodes represent operators and leaf nodes represent operands.
- Construction of syntax tree requires 3 functions:
  i) make-node (Operator, left-pointer, right-pointer)
  ii) make-leaf (id, entry to symbol table)
  iii) make-leaf (num, value)
- Associativity and precedence of the operators can be known by the analysis of the parse tree.
- Basically, there are two types of SDT:
  i) S-attributed SDT: only synthesised attributes are present in SDT.
  ii) L-attributed SDT: Both synthesised and inherited attributes are present but inherited attributes should not depend on the right sibling.
- Dependency Graph: It depicts the flow of information among attribute instances in a particular parse tree.
- Evaluation order of SDT's depend on the dependency graph.