Chapter 5: Syntax Directed Translation

What are we studying in this chapter?

- Syntax Directed Definitions
- Evaluation orders for SDD's
- Application of Syntax-Directed Translation
- Syntax directed translation schemes
- Implementing L-attributed SDD's

- 6 hours

5.1 Introduction

In the previous chapters, we have discussed first two phases of the compiler i.e., lexical analysis phase and syntax analysis phase. In this section, let us concentrate on the third phase of the compiler called *semantic analysis*. The main goal of the semantic analysis is to check for correctness of program and enable proper execution.

We know that the job of the parser is only to verify that the input program consists of tokens arranged in syntactically valid combination. In sematic analysis we check whether they form a sensible set of instructions in the programming language. For example,

- If an identifier is already declared, semantic analyzer checks whether the type of an identifier is respected in all expressions and statements used in the program. That is, it checks whether the type of RHS of an expression of an assignment statement match the type on LHS. It also checks whether the LHS needs to be properly declared and is it an assignable identifier or not.

  Ex 1: int a = 10 + 20; // valid

  Ex 2: char b[] = "Hello"; // valid
  int a = 10 + b; // syntactically valid but, semantically invalid statement;
  // invalid usage of identifier *b* in the expression

  Ex 3: int a[5][5]; // syntactically valid
  a = 10 + 20; // syntactically valid but, semantically invalid statement;

- It checks whether the type and number of parameters in the function definition and the type and number of arguments in the function call are same or not. If not appropriate error messages are displayed
- It checks whether the type of operands are same in an arithmetic operation. If not it displays appropriate error messages
- The index variable used in an array must be integer type else it is a semantic error.
- Appropriate type conversion is also done by semantic analysis phase

5.2 Syntax Directed Translation

Now, let us see "What is semantic analysis?"

Definition: Semantic analysis is the third phase of the compiler which acts as an interface between syntax analysis phase and code generation phase. It accepts the parse tree from the syntax analysis phase and adds the semantic information to the parse tree and performs certain checks based on this information. It also helps constructing the symbol table with appropriate information. Some of the actions performed semantic analysis phase are:

- Type checking i.e., number and type of arguments in function call and in function header of function definition must be same. Otherwise, it results in semantic error.
- Object binding i.e., associating variables with respective function definitions
- Automatic type conversion of integers in mixed mode of operations
- Helps intermediate code generation.
- Display appropriate error messages

The semantics of a language can be described very easily using two notations namely:

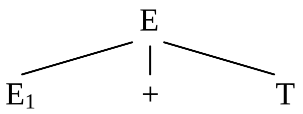
- Syntax directed definition (SDD)
- Syntax directed translation (SDT)

First, we shall discuss about *syntax-directed definition* and next we shall discuss about *syntax-directed translation.*

Note: Consider the production E → E + T. To distinguish E on LHS of the production and E on RHS of the production, we use E_1 on RHS of the production as shown below:

$$E \rightarrow E_1 + T$$

Now, let us consider the production, its derivation and corresponding parse tree as shown below:

Production	**Derivation**	**Parse tree**
$E \rightarrow E_1 + T$	$E \Rightarrow E_1 + T$	

- The non-terminal E on LHS of the production is called *"head of the production"*
- The string of grammar symbols "E + T" on RHS of the production is called *"body of the production"*
- So, in the derivation, head of the production will be the parent node and the symbols that represent body of the production will be the children nodes.

5.2 Syntax Directed Definition

Now, we shall see "What is syntax directed definition (SDD)?"

Definition: A *syntax-directed definition* (SDD) is a context free grammar with *attributes* and *semantic rules*. The attributes are associated with grammar symbols whereas the semantic rules are associated with productions. The semantic rules are used to compute the attribute values.

For example, a simple SDD for the production E→ E_1 + T can be written as shown below:

Production	**Semantic Rule**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

attributes

Observe that a semantic rule is associated with production where the attribute name *val* is associated with each non-terminal used in the rule.

Now, let us see "What is an attribute? Explain with example"

Definition: An *attribute* is a property of a programming language construct. Attributes are always associated with grammar symbols. If X is a grammar symbol and *a* is the attribute, then X.a denote the value of attribute *a* at a particular node X in the parse tree. If we implement the nodes of the parse tree by records or using structures, then the attribute of X can be implemented as a field in the record or a structure.

- **Ex 1:** If *val* is the attribute associated with a non-terminal E, then E.val gives the value of attribute *val* at a node E in the parse tree.
- **Ex 2:** If *lexval* is the attribute associated with a terminal **digit**, then **digit**.lexval gives the value of attribute *lexval* at a node **digit** in the parse tree.
- **Ex 3:** If *syn* is the attribute associated with a non-terminal F, then F.syn gives the value of attribute *syn* at a node F in the parse tree.

Typical examples of attributes are:

- The data types associated with variable such as **int, float, char** etc
- The value of an expression
- The location of a variable in memory
- The object code of a function or a procedure
- The number of significant digits in a number and so on.

Now, let us see "What is a semantic rule?" Explain with example"

5.4 Syntax Directed Translation

Definition: The rule that describe how to compute the attribute values of the attributes associated with a grammar symbol using attribute values of other grammar symbols is called *semantic rule.*

For example, consider the production $E \rightarrow E_1 + T$. The attribute value of E which is on LHS of the production denoted by E.val can be calculated by adding the attribute values of variables E and T which are on RHS of the production denoted by E_1.val and T.val as shown below:

$$E.val = E_1.val + T.val \qquad \text{// Semantic rule}$$

5.3 Inherited and Synthesized attributes

The attribute value for a node in the parse tree may depend on information from its children nodes or its sibling nodes or parent nodes. Based on how the attribute values are obtained we can classify the attributes. Now, let us see "What are the different types or classifications of attributes?" There are two types of attributes namely:

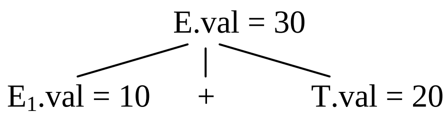
- Synthesized attribute
- Inherited attribute

5.3.1 Synthesized attribute

Now, let us see "What is synthesized attribute? Explain with example"

Definition: The attribute value of a non-terminal A derived from the attribute values of its children or itself is called *synthesized attribute.* Thus, the attribute values of synthesized attributes are passed up from children to the parent node in bottom-up manner.

For example, consider the production: $E \rightarrow E_1 + T$. Suppose, the attribute value *val* of E on LHS (head) of the production is obtained by adding the attribute values *E_1.val* and *T.val* appearing on the RHS (body) of the production as shown below:

Production	**Semantic Rule**	**Parse tree with attribute values**
$E \rightarrow E + T$	$E.val = E_1.val + T.val$	

Now, attribute *val* with respect to E appearing on head of the production is called *synthesized attribute.* This is because, the value of E.val which is 30, is obtained from the children by adding the attribute values 10 and 20 as shown in above parse tree.

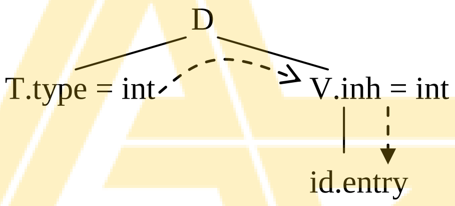
5.3.2 Inherited attribute

Now, let us see "What is inherited attribute? Explain with example"

Definition: The attribute value of a non-terminal A derived from the attribute values of its *siblings* or from its *parent* or *itself* is called *inherited attribute.* Thus, the attribute values of inherited attributes are passed from siblings or from parent to children in top-down manner. For example, consider the production: D → T V which is used for a single declaration such as:

int ***sum***

In the production, D stands for declaration, T stands for *type* such as *int* and V stands for the variable *sum* as in above declaration. The production, semantic rule and parse tree along with attribute values is shown below:

Production	**Parse tree with attribute values**	**Semantic Rule**
D → T V		V.inh = T.type

Observe the following points from the above parse tree:

- The type **int** obtained from the lexical analyzer is already stored in T.type whose value is transferred to its sibling V. This can be done using:

  V.inh = T.type

  Since attribute value for V is obtained from its sibling, it is inherited attribute and its attribute is denoted by *inh*.

- On similar line, the value **int** stored in V.inh is transferred to its child *id.entry* and hence *entry* is inherited attribute of *id* and attribute value is denoted by id.entry

Note: With the help of the annotated parse tree, it is very easy for us to construct SDD for a given grammar.

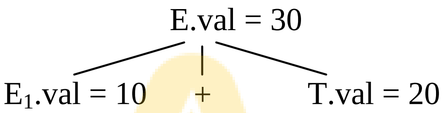
5.3.3 Annotated parse tree

Before proceeding further, it is necessary to know the definition of annotated parse tree. Now, let us see "What is annotated parse tree? Explain with example"

5.6 Syntax Directed Translation

Definition: A parse tree showing the attribute values of each node is called *annotated parse tree.* The terminals in the annotated parse tree can have only synthesized attribute values and they are obtained directly from the lexical analyzer. So, there are no semantic rules in SDD (short form **S**yntax **D**irected **D**efinition) to get the lexical values into terminals of the annotated parse tree. The other nodes in the annotated parse tree may be either synthesized or inherited attributes. **Note:** Terminals can never have inherited attributes

For example, consider the partial annotated tree shown below:



In the above partial annotated parse tree, the attribute values 10, 20 and 30 are stored in E_1.val, T.val and E.val respectively.

Example 5.1: Write the SDD for the following grammar:

S → En where *n* represent end of file marker
E → E + T | T
T → T * F | F
F → (E) | **digit**

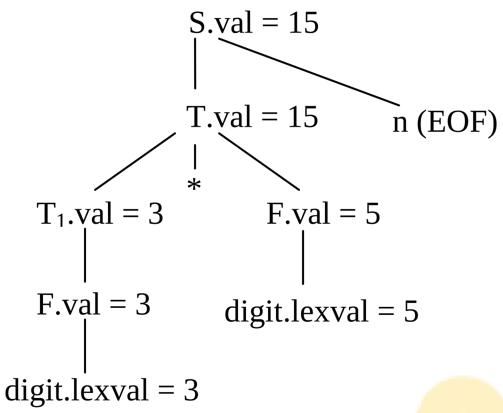
Solution: The given grammar is shown below:

S → En
E → E + T | T
T → T * F | F
F → (E) | **digit**

The above grammar generates an arithmetic expression consisting of parenthesized or un-parenthesized expression with operators + and *. For the sake of convenience, let us consider part of the grammar written as shown below:

S → Tn
T → T * F | F
F → digit

Using the above productions we can generate an un-parenthesized expression consisting of only * operator such as: 3*4 or 3*4*5 etc. The annotated parse tree for evaluating the expression 3*5 is shown below:



It is very easy to see how the values 3 and 5 are moved from bottom to top (propagated upwards) till we reach the root node to get the value 15. The rules to get the value 15 from the productions used are shown below:

Productions	**Semantic rules**
F → digit	F.val = digit.lexval
T → F	T.val = F.val
T → T_1 * F	T.val = T_1.val * F.val
S → Tn	S.val = T.val

On similar lines we can write the semantic rules for the following productions as shown below:

Productions	**Semantic rules**
S → En	S.val = E.val
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
F → (E) |	F.val = E.val

Now, the final SDD along with productions and semantic rules is shown below:

Productions	**Semantic Rules**
S → En	S.val = E.val
E → E + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.lexval

Example 5.2: Write the grammar and syntax directed definition for a simple desk calculator and show annotated parse tree for the expression (3+4)*(5+6)

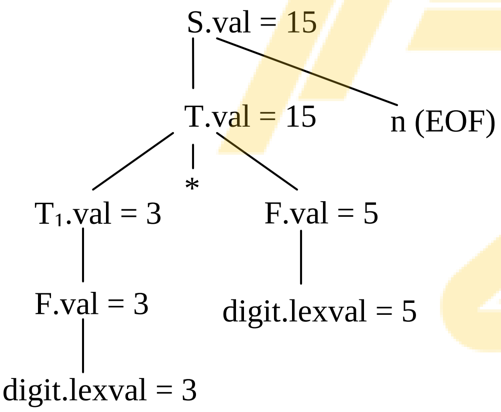
Solution: A simple desk calculator performs operations such as addition, subtraction, division and multiplication with or without parenthesis. The grammar for obtaining an arithmetic expression with or without parentheses can be written as shown below:

S → En
E → E + T | E – T | T
T → T * F | T / F | F
F → (E) | **digit**

The above grammar generates an arithmetic expression consisting of parenthesized or un-parenthesized expression with operators +, –, * and / operators. For the sake of convenience, let us consider part of the grammar written as shown below:

S → Tn
T → T * F | F
F → digit

Using the above productions we can generate an un-parenthesized expression consisting of only * operator such as: 3*4 or 3*4*5 etc. The annotated parse tree for evaluating the expression 3*5 is shown below:



It is very easy to see how the values 3 and 5 are moved from bottom to top (propagated upwards) till we reach the root node to get the value 15. The rules to get the value 15 from the productions used are shown below:

Productions	Semantic rules
F → digit	F.val = digit.lexval
T → F	T.val = F.val
T → T_1 * F	T.val = T_1.val * F.val
S → Tn	S.val = T.val

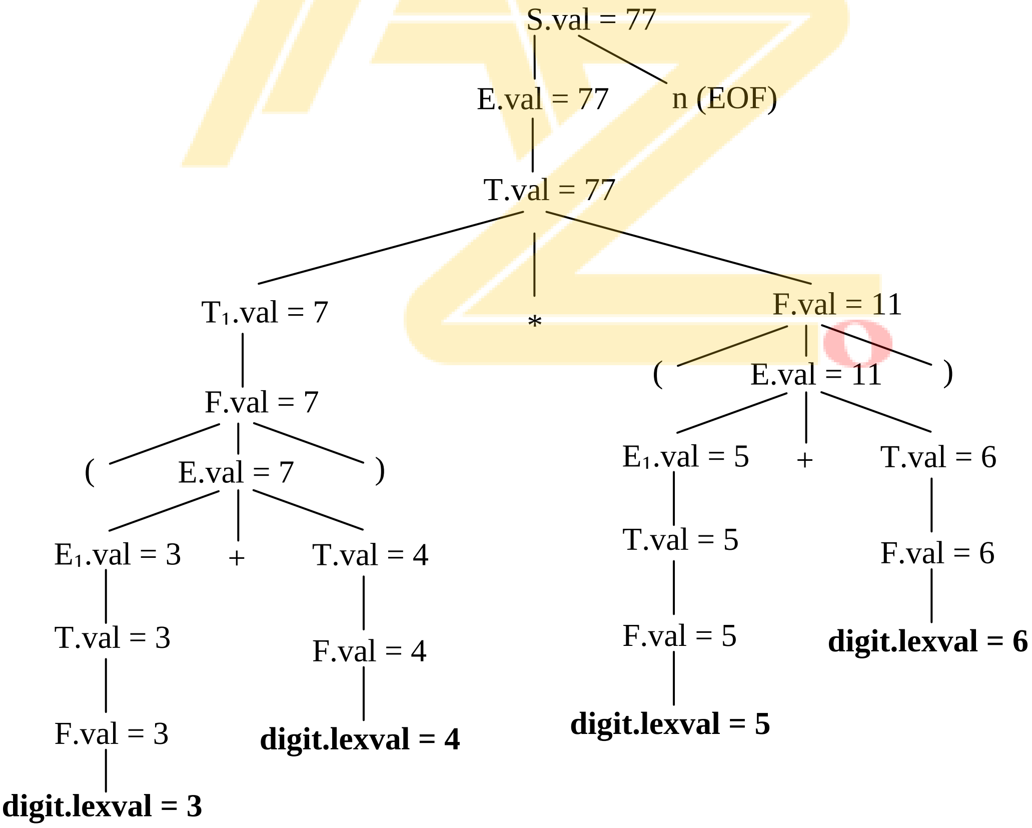
On similar lines we can write the semantic rules for the following productions as shown below:

Productions	Semantic rules
S → En	S.val = E.val
E → E_1 + T	E.val = E_1.val + T.val
E → E_1 – T	E.val = E_1.val – T.val
T → T_1 / F	T.val = T_1.val + F.val
E → T	E.val = T.val
F → (E)	F.val = E.val

So, the final SDD for simple desk calculator can be written as shown below:

Productions	Semantic Rules
S → En	S.val = E.val
E → E + T	E.val = E_1.val + T.val
E → E - T	E.val = E_1.val - T.val
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → T / F	T.val = T_1.val / F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.lexval

The annotated parse tree for the expression (3+4)*(5+6) consisting of attribute values for each non-terminal is shown below:



5.4 Evaluating an SDD at the Nodes of a Parse Tree

We can easily obtain an SDD using the following two steps:
Step 1: Construct the parse tree
Step 2: Use the rules to evaluate attributes of all the nodes of the parse tree.
Step 3: Obtain the attribute values for each non-terminal and write the semantic rules for each production. When complete annotated parse tree is ready, we will have the complete SDD

Now, the question is "How do we construct an annotated parse tree? In what order do we evaluate attributes?"

- If we want to evaluate an attribute of a node of a parse tree, it is necessary to evaluate all the attributes upon which its value depends.
- If all attributes are synthesized, then we must evaluate the attributes of all of its children before we can evaluate the attribute of the node itself.
- With synthesized attributes, we can evaluate attributes in any bottom up order.
- Whether synthesized or inherited attributes there is no single order in which the attributes have to be evaluated. There can be one or more orders in which the evaluation can be done.

For example, to see how an annotated parse tree can be constructed, *refer example 5.1.*

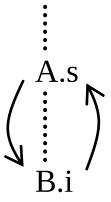
5.4.1 Circular dependency

Before proceeding further, let us see, "What is circular dependency when evaluating the attribute value of a node in an annotated parse tree?"

Definition: If the attribute value of a parent node depends on the attribute value of child node and vice-versa, then we say, there exists a circular dependency between the child node and parent node. In this situation, it is not possible to evaluate the attribute of either parent node or the child node since one value depends on another value.

For example, consider the non-terminal A with synthesized attribute A.s and non-terminal B with inherited attribute B.i with following productions and semantic rules:

Production	Semantic rule	Partial annotated parse tree

A → B	A.s = B.i
	B.i = A.s + 6
		

Note: In the above semantic rule *s* is synthesized attribute and *i* is inherited attribute

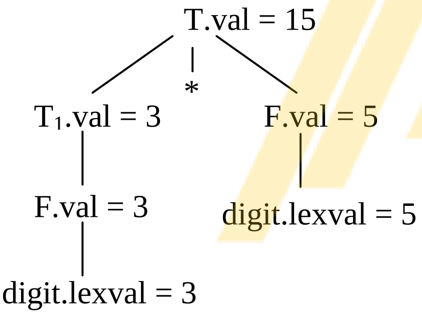
Note that the above two semantic rules are circular in nature (see above figure). Note that to compute A.s we require the value of B.i and to compute the value of B.i, we require the value of A.s. So, it is impossible to evaluate either the value of A.s or the value of B.i without evaluating other.

5.4.2 Why evaluate inherited attributes (Evaluate inherited attributes)

Now, let us see "What is the use of inherited attributes?" The reason for inherited attributes can be explained using an example. Consider the following grammar:

$$T \rightarrow T * F \mid F$$
$$F \rightarrow \text{digit}$$

Using the above productions we can generate an un-parenthesized expression consisting of only * operator such as: 3*4 or 3*4*5 etc. The above grammar has left-recursion and it is suitable for *bottom up parser* such as LR parser. The annotated parse tree for evaluating the expression 3*5 is shown below:



It is very easy to see how the values 3 and 5 are moved from bottom to top (propagated upwards) till we reach the root node to get the value 15 as shown below:

Semantic rules	**Productions**
F.val = digit.lexval	F → digit
T.val = F.val	T → F
T.val = T_1.val * F.val	T → T * F

Thus, using bottom-up manner, the values 3 and 5 are moved upwards to get the result 15 using the semantic rules associated with each production. So, far there is no problem.

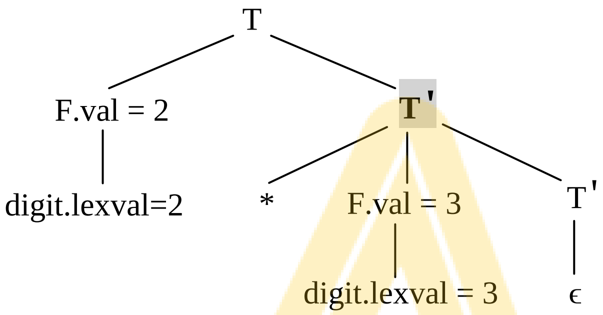
Example 5.3: Obtain SDD and annotated parse tree for the following grammar using top-down approach:

$$T \rightarrow T * F \mid F$$
$$F \rightarrow \text{digit}$$

Let us see, "What happens if top-down parser is used to parser the input 3*5 using the above grammar?" We have already seen earlier in previous chapters that when we use top-down parser, first we have to eliminate left-recursion and then do parsing. So, the given grammar is not suitable for top-down parser because of left-recursion. So, let us remove left-recursion. The grammar obtained after removing left-recursion is shown below:

$T \to F\ T'$
$T' \to *\ F\ T_1'$
$T' \to \epsilon$
$F \to \text{digit}$

To construct SDD for the above grammar, let us obtain the derivation tree for the expression 2*3 with some of the attributes as shown below:

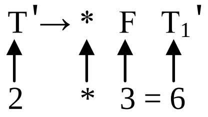


Observe the following points from above partial annotated parse tree

- The values 2, 3 and 4 are moved upwards till we get node F. Since the attribute value of F is obtained from its child, the attribute of F denoted by *val* will be synthesized attribute and F.val can be obtained using semantic rule by considering the production F → digit as shown below:

Production	**Semantic Rule**	**Type**
$F \to \text{digit}$	F.val = digit.lexval	synthesized

- Now, we need to multiply 2*3. Observe that there is no node with 2, * and 3 as the children. So, we can perform 2*3 using the production $T' \to *\ F\ T_1'$ from the top as shown below:

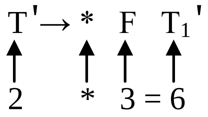


Now, the question is how to transfer 2 to T', how to multiply 2*3 and how to store the result 6 in T_1'. These activities can be done as shown below:

1) Observe from the partial annotated parse tree and above scenario that, the first operand 2 already present in F.val which is the left child of T must be transferred to right child T' using the production $T \to F\ T'$ as shown below:

Production	**Semantic Rule**	**Type**
$T \to F\ T'$	T'.inh = F.val	Inherited

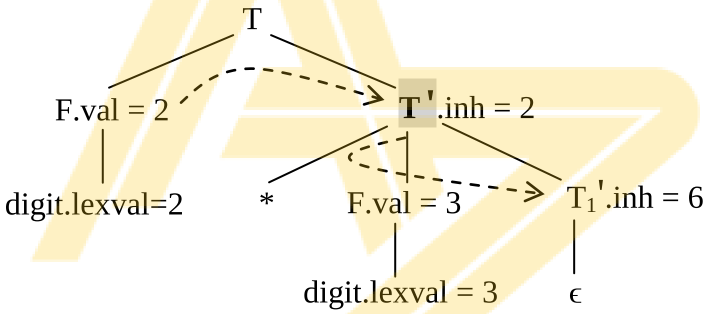
2) Now, 2*3 can be computed as shown below:



Observe from above figure that multiply 2*3 means we need to compute $T' * F$ and store the result in T_1'. That is, take the inherited attribute value T'.inh and multiply with synthesized attribute value F.val and store the result in T_1'.inh. This can be done using as shown below:

Production	**Semantic Rule**	**Type**
$T' \rightarrow *FT_1'$	T_1'.inh = T'.inh * F.val	Inherited

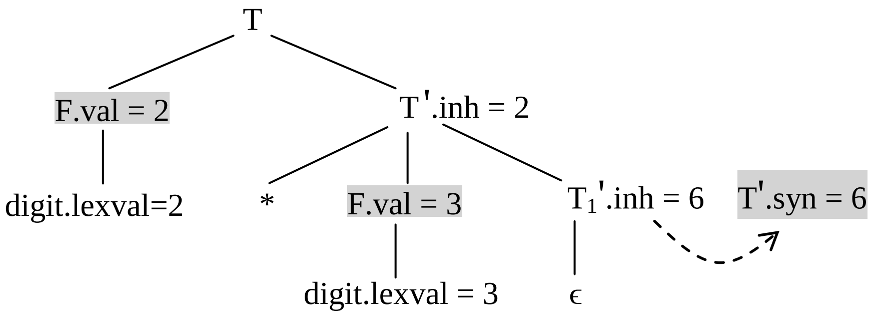
Note: Since T_1'.inh is computed using sibling attribute values, it is inherited attribute. Now, the partial annotated parse tree showing the attribute values computed so far is shown below:



- Observe from above figure that the node T_1'.inh which is same as T' produce ϵ and so, the synthesized attribute value T'.syn can be obtained using the production $T' \rightarrow \epsilon$ and its semantic rule as shown below:

Production	**Semantic Rule**	**Type**
$T' \rightarrow \epsilon$	T'.syn = T'.inh	Synthesized

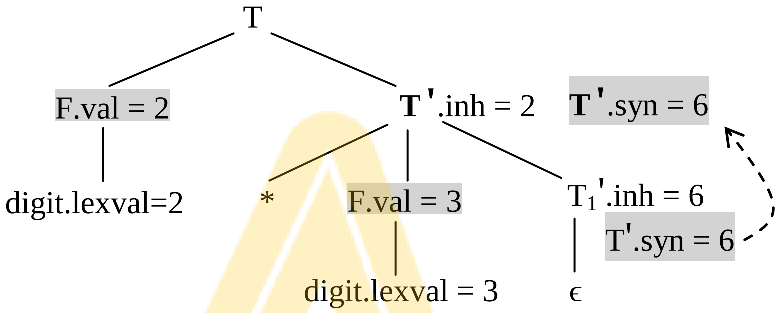
The partial annotated parse tree is shown below:



- The synthesized value $T'.syn = 6$ which we call as $T_1'.syn$ is transferred to its parent T' with attribute value $T'.syn = 6$ using the production $T' \rightarrow * F T_1'$ as shown below:

Production	Semantic Rule	Type
$T' \rightarrow * F T_1'$	$T'.syn = T_1'.sys$	Synthesized

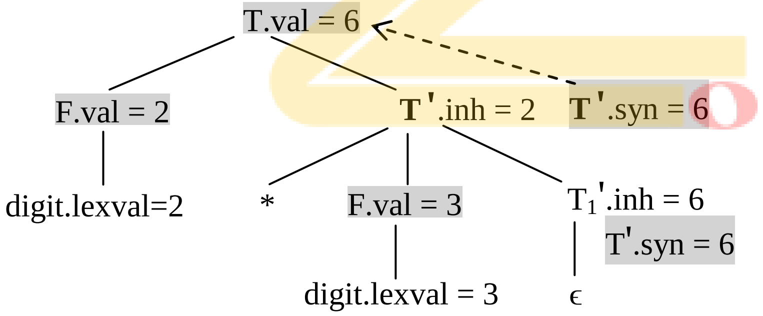
The partial annotated parse tree is shown below:



- Finally $T'.syn = 6$ is transferred to its parent T using the production $T \rightarrow F T'$ as shown below:

Production	Semantic Rule	Type
$T \rightarrow F T'$	$T.val = T'.syn$	Synthesized

The annotated parse tree that shows the value of T.val is shown below:

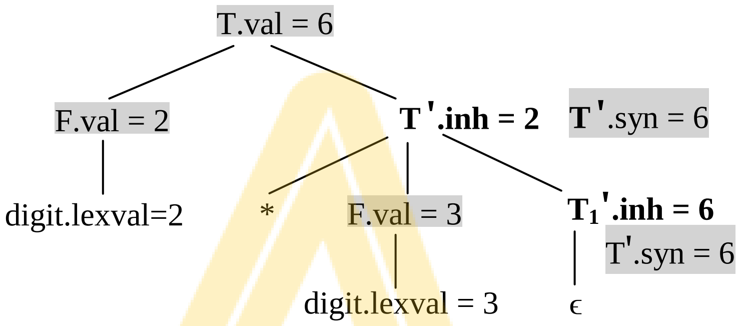


So, the final SDD for the given grammar can be written as shown below:

Productions	Semantic Rules	Type
$T \rightarrow F T'$	$T'.inh = F.val$	Inherited
	$T.val = T'.syn$	Synthesized

$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$	Inherited
	$T'.syn = T_1'.sys$	Synthesized
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$	Synthesized
$F \rightarrow digit$	$F.val = digit.lexval$	Synthesized

The final annotated parse tree is shown below:



Example 5.4: Obtain SDD for the following grammar using top-down approach:

$S \rightarrow En$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid digit$

and obtain annotated parse tree for the expression (3 + 4) * (5 + 6)n

Solution: The given grammar has left recursion and hence it is not suitable for top-down parser. To make it suitable for top-down parsing, we have to eliminate left recursion. After eliminating left recursion (see example 2.13 for details), the following grammar is obtained:

$S \rightarrow En$
$E \rightarrow T E'$
$E' \rightarrow + T E_1' \mid \epsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T_1' \mid \epsilon$
$F \rightarrow (E) \mid digit$

Note: The variables S, E, T and F are present both in given grammar and grammar obtained after left recursion. So, only for the variables S, E, T and F we use the attribute name *v* (stands for val) and for all other variables we use *s* for synthesized attribute and *i* for inherited attribute

Consider the following productions:

$S \rightarrow En$
$F \rightarrow (E) \mid digit$

They do not have left recursion and they are retained in the grammar which is obtained after eliminating left recursion. So, we can compute the attribute value of LHS (head) from the attribute values of RHS (i.e., children) for the above productions and hence they have synthesized attributes. The productions, semantic rules and type of the attribute are shown below:

Production	**Semantic Rule**	**Type**
$S \rightarrow E\ n$	S.v = E.v	Synthesized
$F \rightarrow (\ E\ )$	F.v = E.v	Synthesized
$F \rightarrow d$	F.v = d.lexval	Synthesized

Consider the following productions and write the annotated parse tree for the expression 2*3 with flow of information (See example 5.3 for detailed explanation) as shown below:

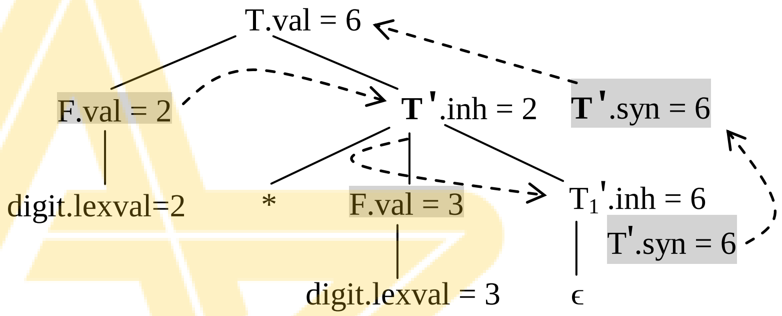
Productions

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T_1' \mid \epsilon$

$F \rightarrow digit$

Annotated parse tree for 2*3



By following the dotted arrow lines, we can write the various semantic rules for the corresponding productions as shown below:

	Semantic rule	**Production**
F.val = 2 is copied to T'.inh	T'.inh = F.val	$T \rightarrow F\ T'$
T'.inh * F.val is copied to T_1'.inh	T_1'.inh = T'.inh * F.val	$T' \rightarrow *\ F\ T_1'$
T_1'.inh is copied to T'.syn	T'.syn = T_1'.inh	$T' \rightarrow \epsilon$
T'.syn is moved to its parent	T'.syn = T_1'.syn	$T' \rightarrow *\ F\ T_1'$
T'.syn is moved to its parent T	T.val = T'.syn	$T \rightarrow F\ T'$

The above productions and their respective semantic rules along with the type of attribute are shown below:

Production	**Semantic rules**
$T \rightarrow F\ T'$	T'.inh = F.val T.val = T'.syn

$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$
	$T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$

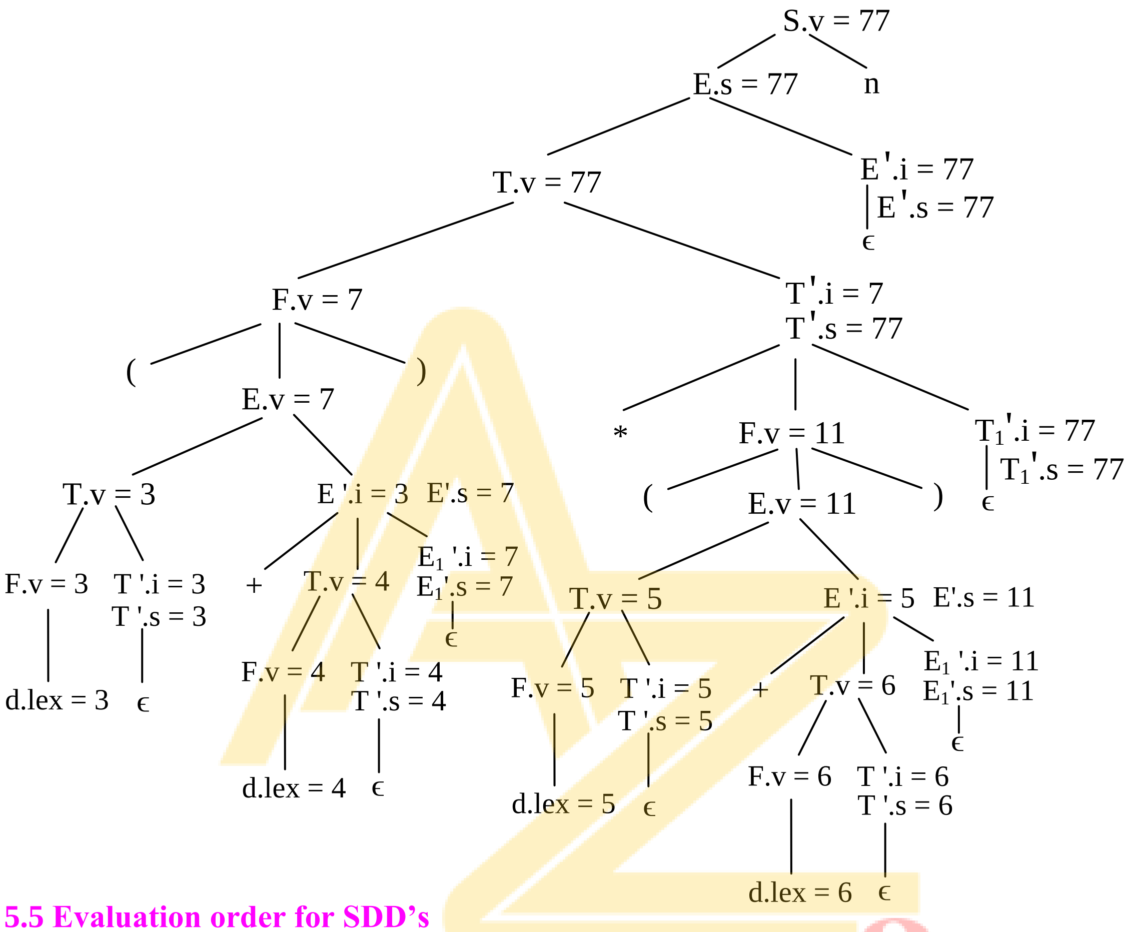
Exactly similar to the above we can write the semantic rules for the given productions as shown below:

Production	**Semantic rules**
$E \rightarrow T E'$	$E'.inh = T.val$
	$E.val = E'.syn$
$E' \rightarrow + T E_1'$	$E_1'.inh = E'.inh + T.val$
	$E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$

Combining all productions and semantic rules we can write the final SDD as shown below:

Production	**Semantic Rule**	**Type**
$S \rightarrow E n$	$S.v = E.v$	Synthesized
$E \rightarrow T E'$	$E'.inh = T.val$	Inherited
	$E.val = E'.syn$	Synthesized
$E' \rightarrow + T E_1'$	$E_1'.inh = E'.inh + T.val$	Inherited
	$E'.syn = E_1'.syn$	Synthesized
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$	Synthesized
$T \rightarrow F T'$	$T'.inh = F.val$	Inherited
	$T.val = T'.syn$	Synthesized
$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$	Inherited
	$T'.syn = T_1'.syn$	Synthesized
$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$	Synthesized
$F \rightarrow (E)$	$F.val = E.val$	Synthesized
$F \rightarrow d$	$F.val = d.lexval$	Synthesized

The annotated parse tree that shows the values of each attributed value while evaluating the following expression (3 + 4) * (5 + 6) is shown below:



5.5 Evaluation order for SDD's

The evaluation order to find attribute values in a parse tree using semantic rules can be easily obtained with the help of dependency graph. While annotated parse tree shows the values of attributes, a dependency graph helps us to determine how those values can be computed.

5.5.1 Dependency graphs

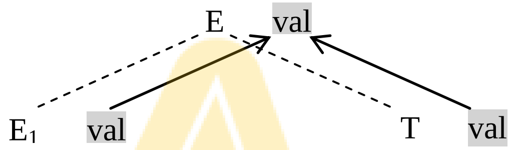
Now, let us see "What is a dependency graph?"

Definition: A graph that shows the flow of information which helps in computation of various attribute values in a particular parse tree is called *dependency graph*. An edge from one attribute instance to another attribute instance indicates that the attribute value of the first is needed to compute the attribute value of the second.

For example, consider the following production and rule:

Production	Semantic rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

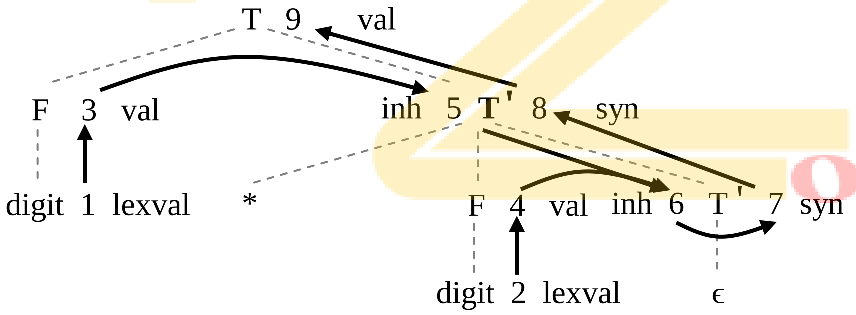
This production has left recursion and hence, it is suitable for bottom-up parser. In bottom up parser, the attribute value of LHS (head) depends on the attribute value of RHS (body of the production or children in the parse tree). So, the attribute value of E is obtained from its children E_1 and T. The portion of a dependency graph for this production can be written as shown below:



In the above figure, the dotted lines along with nodes connected to them represent the parse tree. The shaded nodes represented as val with solid arrows originating from one node and ends in another node is the dependency graph.

Example 5.5: Obtain the dependency graph for the annotated parse tree obtained in example 5.3

The dependency graph for the annotated parse tree obtained in example 5.3 can be written as shown below:



Observe the following points from the above dependency graph

- Nodes identified by numbers 1 and 2 represent the attribute *lexval* which is associated with two leaves labeled **digit**
- Nodes 3 and 4 represent the attribute *val* associated with two nodes labeled F.
- The edge from node 1 to node 3 and from node 2 to node 4 indicates that in the semantic rule, the attribute value F.val is obtained using attribute value digit.lexval
- Nodes 5 and 6 represent the inherited attribute T'.inh which is associated with each of the non-terminal T'

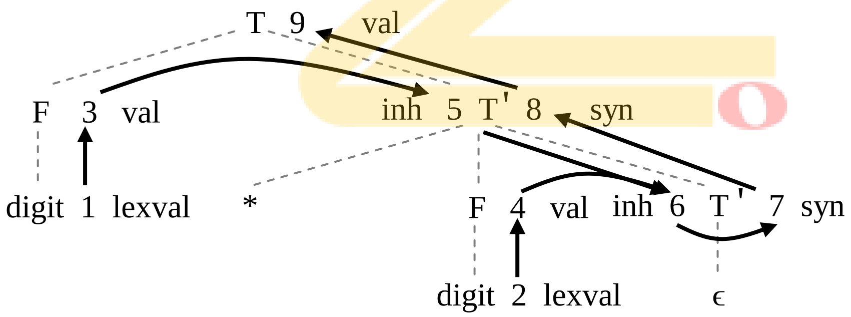
- The edge from 3 to 5 indicate that T'.inh is obtained from its sibling F.val and hence T' has an inherited attribute name *inh*
- The edge from 5 to 6 and another edge from 4 to 6 indicate that the two attribute values T'.inh and F.val are multiplied to get the attribute value at node 6
- The edge from 6 to 7 indicate that there is an ϵ-production and the attribute value is obtained from itself and hence its attribute value T'.syn is obtained from T'.inh
- The node 7 is obtained from itself, 8 is obtained from node 7 and 9 is obtained from node 8 and all are synthesized attributes.
- Finally, T.val at node 9 is obtained from its child at node 8.

5.5.2 Ordering the evaluation of attributes

Now, let us see "What is topological sort of the graph?"

Definition: Topological sort of a directed graph is a sequence of nodes which gives the order in which the various attribute values can be computed in a parse tree. Using the dependency graph, we can write the order in which we can evaluate various attribute values in the parse tree. This ordering is nothing but the topological sort of the graph. There may be one or more orders to evaluate attribute values. If the dependency graph has an edge from A to B, then the attribute corresponding to A must be evaluated before evaluating attribute value at node B.

Example 5.6: Give the topological sort of the following dependency graph



Solution: The various topological sorts that can be obtained using the dependency graph are shown below:

Topological sort 1: 1, 2, 3, 4, 5, 6, 7, 8, 9
Topological sort 2: 1, 3, 2, 4, 5, 6, 7, 8, 9
Topological sort 3: 1, 3, 5, 2, 4, 6, 7, 8, 9 and so on

Note: If there is a cycle in the dependency graph, topological sort does not exist.

5.5.3 S-attributed definitions

In this section, let us see Given an SDD, it is very difficult to tell whether there exist a cycle in the dependency graph corresponding a parse tree. But, in practice, translations can be implemented using classes of SDD's that will guarantee an evaluation order, since they do not permit dependency graphs with cycles.

Now, let us see "What are different classes of SDD's that guarantee evaluation order?" The two classes of SDD's that guarantee an evaluation order are:

- S-attributed definition
- L-attributed definition

Now, let us see "What is S-attributed definition?"

Definition: An SDD that contains only synthesized attributes is called S-attributed. In an S-attributed SDD, each semantic rule computes the attribute value of a non-terminal that occurs on LHS of the production that represent head of the production with the help of the attribute values of non-terminals on the right hand side of the production that represent body of the production.

For example, the SDD shown below is an S-attributed definition.

Productions	Semantic Rules
S → En	S.val = E.val
E → E + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.lexval

Note that all the attributes in above SDD such as S.val, E.val, T.val and F.val are synthesized attributes and hence the SDD is an S-attributed. Observe the following points:

- When an SDD is S-attributed, we can evaluate its attributes in any bottom up order of the nodes of the parse tree

- S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal.
- The postorder traversal corresponds exactly to the order in which LR parser reduces a production body (RHS of the production) to its head (LHS of the production)
- The attributes can be evaluated very easily by performing the postorder traversal of the parse tree at a node N as shown below:

```
postorder (N)
{
        for (each child C of N from left)
                postorder(C)
        end for

        evaluate the attributes associated with node N
}
```

Now, let us see "What is an attribute grammar?"

Definition: An SDD without any side effects is called *attribute grammar.* The semantic rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants. Attribute grammars have the following properties:

- They do not have any side effects
- They allow any evaluation order consistent with dependency graph.

For example, the SDD obtained from example 5.1 is an attribute grammar. For simplicity, the SDD's that we have seen so far have semantic rules without side effects. But, in practice, it is convenient to allow SDD's to have limited side effects, such as printing the result computed by a desk calculator or interacting with symbol table and so on.

5.5.4 L-attributed definitions

L-attributed definition is a second class of SDD. The idea behind the class is that "Between the attributes associated with a production body, dependency graph edges can go only from left to right, but not from right to left. Hence L-attributed. Now, formally, let us see "What is an L-attributed definition?"

Definition: An L-attributed definition is one of the following:

1) Synthesized or
2) Inherited, but with following rules. Consider the production $A \rightarrow X_1X_2\ldots\ldots\ldots X_n$ and let X_i.a is an inherited attribute. In this situation one of the following rules are applicable:

a) Inherited attributes must be associated with A which is LHS of the production (called head of the production)

or

b) All the symbols X_1, X_2, X_3…..X_{i-1} appearing to the left of X_i have either synthesized or inherited attributes

or

c) Inherited or synthesized attributes associated with this occurrence of X_i itself but without forming any cycles

For example, the SDD shown in example 5.3 is L-attributed. To see why, let us consider the following productions and semantic rules:

Productions	Semantic rules
$T \to F\ T'$	$T'.inh = F.val$
$\boxed{T' \to *\ F}\ T_1'$	$T_1'.inh = T'.inh * F.val$

Observe the following points from above two semantic rules:

- In the first rule observe that attribute value of F denoted by F.val is copied into attribute value of T' denoted by T'.inh shown using an edge going from left to right. Note that F appears to the left of T' in the production body as required
- In the second rule, the inherited attribute value of T' on LHS (head) of the production and synthesized attribute value of F present in body of the production are multiplied and the result is stored in attribute value of T_1' denoted by T_1'. Note that both T' and F appears to the left of T_1' ad required by the rule.
- In each of these cases, the rules use the information "from above or from the left" as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed.

5.5.5 Semantic rules with controlled side effects

Now, let us see "What is a side effect in a SDD?"

Definition: The main job of the semantic rule is to compute the attribute value of each non-terminal in the corresponding parse tree. Any other activity performed other than computing the attribute value is treated as side effect in a SDD.

For example, attribute grammars have no side effects and allow any evaluation order consistent with dependency graph. But, translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment. In practice, translation involves side effects:

- printing the result computed by a desk calculator
- Interacting with symbol table. That is, a code generator might enter the type of an identifier into a symbol table etc.

Now, let us see "How to control the side effects in SDD?" The side effects in SDD can be controlled in one of the following ways:

- Permitting side effects when attribute evaluation based on any topological sort of the dependency graph produces a correct translation
- Impose constraints in the evaluation order so that the same translation is produced for any allowable order

For example, consider the SDD given in example 5.1. This SDD do not have any side effects. Now, let us consider the first semantic rule and corresponding production shown below:

Production	**Semantic rule**
S → En	S.val = E.val

Let us modify the semantic rule and introduce a side effect by printing the value of E.val as shown below:

Production	**Semantic rule**
S → En	print (E.val)

Now, the semantic rule "print(E.val)" is treated as dummy synthesized attribute associated with LHS (head) of the production. The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end only after computing the value of E.val.

The complete SDD of desk calculator with side effects is shown below:

Example 5.7: Write the grammar and SDD for a simple desk calculator with side effect

Solution: The SDD for simple desk calculator along with side effect of printing the value of an attribute after evaluation can be written as shown below:

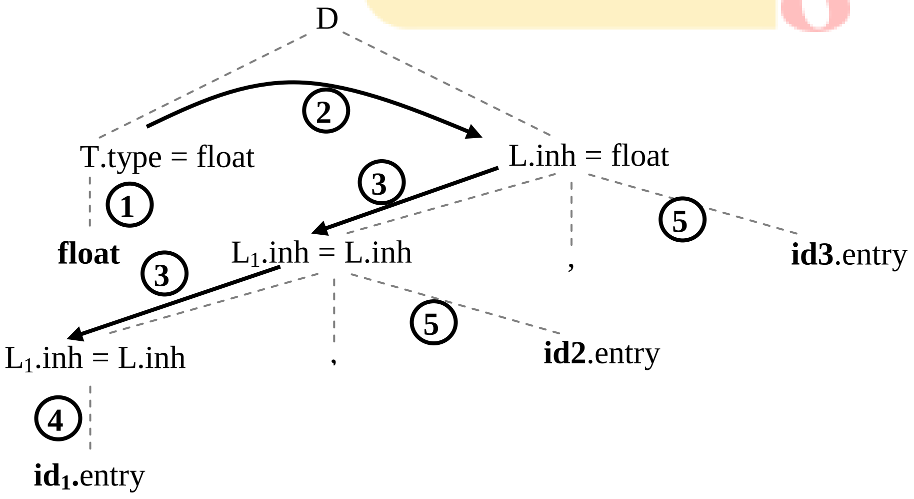
Productions	Semantic Rules
S → En	print(E.val)
E → E + T	E.val = E_1.val + T.val
E → E - T	E.val = E_1.val - T.val
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → T / F	T.val = T_1.val / F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.lexval

Example 5.8: Write the SDD for a simple type declaration and write the annotated parse tree and the dependency graph for the declaration "**float a, b, c**"

Solution: The grammar for a simple type declaration can be written as shown below:

D → T L
T → **int** | **float**
L → L_1 , **id** | **id**

Consider the parse tree for the declaration: "**float a, b, c**" where *a*, *b* and *c* are identifiers represented by *id1*, *id2* and *id3* respectively along with partial annotated parse tree showing the direction of evaluations:



1 The declaration D consists of a basic type T followed by a list L of identifiers. T can be either **int** or **float**. Thus, the tokens corresponding **int** and **float** such as *integer* and *float* obtained from the lexical analyzer are copied into attribute value of T. The production and its semantic rule can be written as shown below:

Productions	Semantic rules
T → **int**	T.type = **integer**
T→ **float**	T.type = **float**

2 Observe that the attribute value T.type available in the left substree should be transferred to right subtree to L. Since attribute value is transferred from left sibling to right sibling its attribute must be inherited attribute and it is denoted by L.inh and can be obtained using the production D → T L as shown below:

Productions	Semantic rules
D → T L	L.inh = T.type

3 The type L.inh must be transferred to identifier **id** and hence it has to be copied into L_1.inh which is the left most child in RHS of the production "L → L_1, id". This can be done as shown below:

Productions	Semantic rules
L → L_1, id	L_1.inh = L.inh

4 The attribute value of L.inh in turn must be entered as the type for identifier **id** using the production "L → id". This can be done as shown below:

Productions	Semantic rule with side effect
L → id	Addtype(id.entry, L.inh)

where Addtype() is a function which accepts two parameters:

- id.entry is a lexical value that points to a symbol table
- L.inh is the type being assigned to every identifier in the list
- The function installs the type L.inh as the type of the corresponding identifier

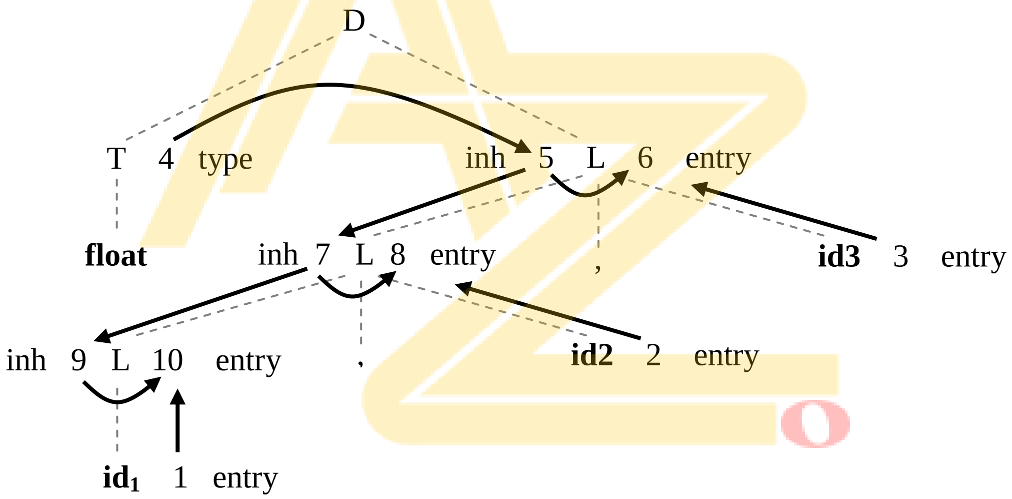
5 The attribute value of L.inh in turn must be entered as the type for identifier **id** which is the right most child in RHS of the production "L → L_1, id". This can be done as shown below:

Productions	Semantic rule with side effect
L → L_1, id	Addtype(id.entry, L.inh)

Finally, the SDD for the grammar can be written by looking into dependency graph as shown below:

Productions	Semantic Rules
D → T L	L.inh = T.type
T → **int**	T.type = integer
T → **float**	T.type = float
L → L_1, **id**	L_1.inh = L.inh Addtype(L.inh, id.entry)
L → **id**	Addtype(L.inh, id.entry)

Note: Thus, we have an L-attributed definition with side effect of adding the type into symbol table for the corresponding identifier. The dependency graph for type declaration statement "**float a, b, c**" is shown below:



Observe the following points from above diagram:

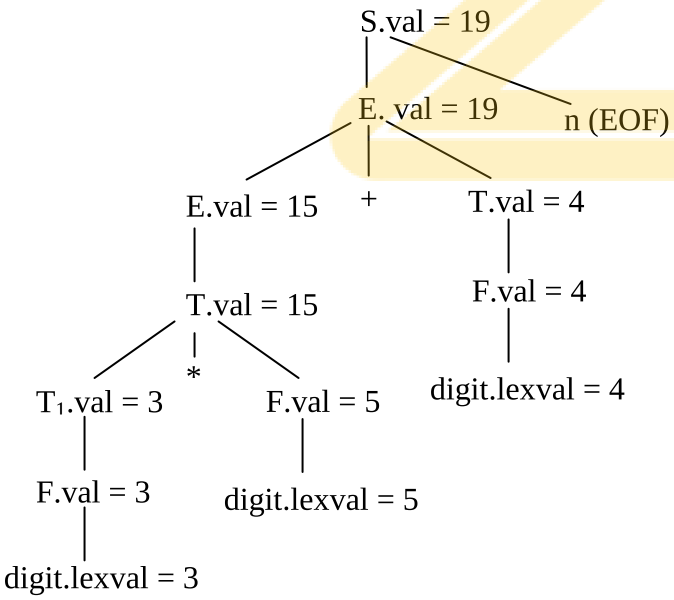
- Numbers 1 through 10 represent the nodes of the dependency graph.
- Nodes 1, 2 and 3 represent attribute *entry* associated with each of the leaves labeled **id**.
- Node 4 represent the attribute T.type and is actually where attribute evaluation begins
- Nodes 5, 7 and 9 represent the attribute L.inh associated with each occurrence of the non-terminal L.
- Nodes 6, 8 and 10 represent the dummy attributes that represent the application of the function addType() to a type and one of these *entry* values

Example 5.9: Write the SDD for a simple desk calculator. Write the annotated parse tree for the expression 3*5+4n

Solution: The SDD for a simple desk calculator is shown below: (See example 5.2 for detailed explanation)

Productions	**Semantic Rules**
S → En	S.val = E.val
E → E + T	E.val = E_1.val + T.val
E → E - T	E.val = E_1.val - T.val
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → T / F	T.val = T_1.val / F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.lexval

The annotated parse tree for the expression 3*5 + 4n can be written as shown below:



5.6 Syntax directed translation

Now, let us see "What is syntax directed translation?"

Definition: The **S**yntax **D**irected **T**ranslation (in short SDT) is a context free grammar with embedded semantic actions. The semantic actions are nothing but the sequence of steps or program fragments that will be carried out when that production is used in the derivation. The SDTs are used:

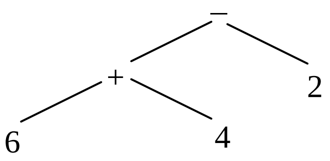
- To build syntax trees for programming constructs.
- To translate infix expressions into postfix notation
- To evaluate expressions

5.6.1 Construction of syntax trees

The main application of SDT in this section is the construction of syntax trees. Since some of the compilers use syntax trees as an intermediate representation, an SDD accepts the input string and produce a syntax tree. The syntax tree is also called *abstract syntax tree.* The parse tree is also called *concrete syntax tree.*

Before proceeding further, let us see "What is a syntax tree? What is the difference between syntax tree and parse tree?"

Definition: A *syntax tree* also called *abstract syntax tree* is a compressed form of parse tree which is used to represent language constructs. In a syntax tree for an expression, each interior node represents an operator and the children of the node represent the operands of the operator. In general, any programming construct can be handled by making up an operator for the construct and treat semantically meaningful components of that construct as operands. For example, the syntax tree for the expression "6 + 4 – 2" is shown below:



In the syntax tree observe the following points:

- The operator – represent the root node.
- The sub-trees of the root represent the sub-expressions "6+4" and 2
- Similarly, for the expression, + is the root and 6 and 4 are the operands representing the children

It is easy to construct syntax trees with the help of SDD's. The syntax trees resemble the parse trees to some extent but with following changes:

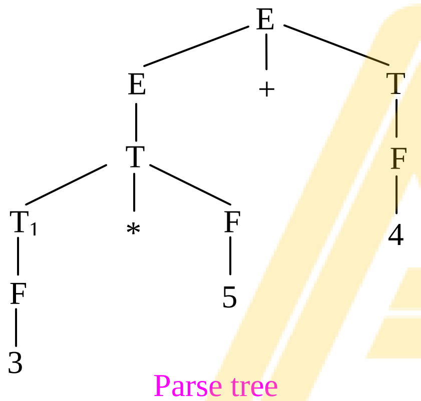
Note 1: In a syntax tree, all the operators and keywords appear as interior nodes.

Note 2: In a parse tree all operators and keywords appear as leaf nodes.

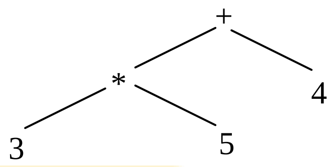
Example 5.10: For the following grammar show the parse tree and syntax tree for the expression 3 *5 + 4:

E → E + T | E – T | T
T→ T*F | T/F | F
F→ (E) | digit | id

Solution: The annotated parse tree for the expression 3*5+4 is shown in example 5.9. Now, the parse tree and syntax tree for the expression 3*5+4 are shown below:



Parse tree



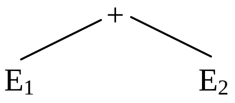
Syntax tree

Now, let us see "How to construct syntax trees?" The syntax tree for expressions can be constructed using two SDD's namely:

- S-attributed definition which is used for bottom-up parsing
- L-attributed definition which is used for top-down parsing

5.6.1.1 Syntax tree for S-attributed definition

The main application of SDT in this section is the construction of syntax trees. Now, let us see "How to construct semantic rules that help us to create syntax trees for the expressions?" Each node in a syntax tree represents a programming construct and the children of the node represent meaningful components of that construct. A syntax-tree node representing an expression E_1 + E_2 has label + and two children representing the sub-expressions E_1 and E_2 as shown below:



The nodes of a syntax tree can be implemented by creating objects where each object containing two or more fields. The two functions that are useful in constructing the syntax trees are shown below:

- **Leaf(op, val) :** This function is called only for the terminals and it is used to create only leaf nodes containing two fields namely:
  - *op* field holds the label for the node
  - *val* field holds the lexical value obtained from the lexical analyzer

- **Node(op, c_1, c_2, c_3,…..c_n) :** This function is called to create only interior nodes with various fields namely:
  - *op* field holds the label for the node
  - c_1, c_2, c_3, …c_n refer (or pointers) to children for the node labeled *op*.

Example 5.11: Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using bottom up approach

Solution: The grammar to generate an arithmetic expression is shown below:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T{*}F \mid T/F \mid F$$
$$F \rightarrow (E) \mid digit \mid id$$

It is easy to construct syntax trees with the help of SDD's (For detail of SDD of arithmetic expression see example 5.2) as shown below:

- For the production of the form $E \rightarrow E_1 + E_2$, we have to use a rule that creates a node with '+' for *op* fields two children E_1.node and E_2.node that represent sub-expressions. This can be done by creating a node with **new** operator using function Node() as shown below:

$$E \rightarrow \underbrace{E_1 + E_2}$$
$$\downarrow$$
$$E.node = new\ Node(\text{‘+’}, E_1.node, E_2.node)$$

On similar lines, we can write the semantic rules for other productions consisting of the operators +, –, * and / as shown below:

Production	Semantic rules
$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, E_2.node)
$E \rightarrow E_1 - T$	E.node = new Node('–', E_1.node, T.node)
$T \rightarrow T_1 * F$	T.node = new Node('*', T_1.node, F.node)
$T \rightarrow T_1 / F$	T.node = new Node('/', T_1.node, F.node)

- For the production of the form E → T and E→ (T) , no node is created, since E.node is the same as that of T.node. So, semantic rules for the productions of the above form can be written as shown below:

Production	**Semantic rules**
E → T	E.node = T.node
T → F	T.node = F.node
F → (E)	F.node = E.node

- For the production of the form A → a where *a* is a terminal, use the function Leaf() with *a* and *a.entry* as the parameter if *a* is and identifier or *a.val* as the parameter if *a* is a number of digit. So, semantic rules for the productions of the above form can be written as shown below:

Production	**Semantic rules**
F → digit	F.node = new Leaf(digit, digit.val)
F → id	F.node = new Leaf(id, id.entry)

So, the final set of SDDs used to construct syntax trees for simple expressions can be written as shown below:

Production	**Semantic rules**
E → E_1 + T	E.node = new Node('+', E_1.node, T.node)
E → E_1 – T	E.node = new Node('–', E_1.node, T.node)
E → T	E.node = T.node
T → T_1 * F	T.node = new Node('*', T_1.node, F.node)
T → T_1 / F	T.node = new Node('/', T_1.node, F.node)
T → F	T.node = F.node
F → (E)	F.node = E.node
F → digit	F.node = new Leaf(digit, digit.val)
F → id	F.node = new Leaf(id, id.entry)

Note: The above semantic rules show a left-recursive grammar that is S-attributed (so all attributes are synthesized). Now, let us see how to create a syntax tree using above semantic rules.

Example 5.12: Create a syntax tree for the expression "a – 4 + c"

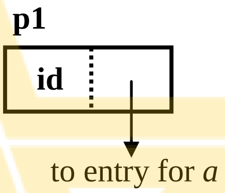
Solution: Consider for the arithmetic expression "**a - 4 + c**". Here, we make use of the following functions to create the nodes of syntax trees for expressions with binary operators.

- Node(*op* , *left* , *right*) creates an operator node with label *op* and two fields containing pointers to *left* and *right* sub-tree
- Leaf(**id** , *entry*) creates a identifier node with label **id** and a field containing *entry*, which pointer to the symbol-table entry for the identifier.
- Leaf(**digit** , *val*) creates a number node with label **digit** and a field containing *val* which represent the value of the number.

The following sequence of function calls creates the syntax tree for the arithmetic expression **a – 4 + c** as shown below:

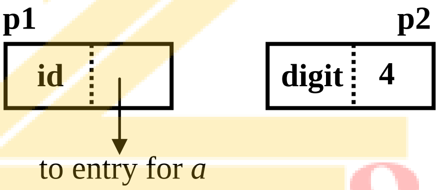
Step 1: Create a leaf node for identifier *a* using the function Leaf() as shown below:

p1 = new Leaf(**id** , *id.entry*);



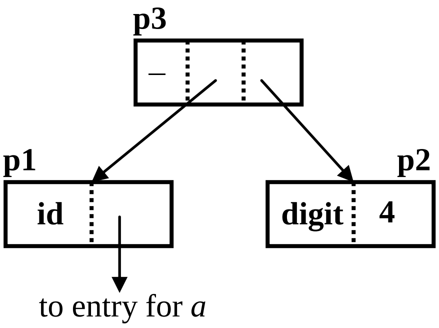
Step 2: Create a leaf node for digit 4 using the function Leaf() as shown below:

p2 = new Leaf(**digit** , *4*);



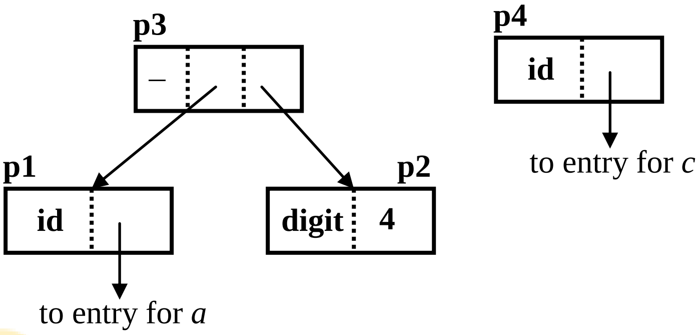
Step 3: Create an interior node by passing '–' as the first argument and pointers to the first two leaves p1 and p2 with the help of function Node() as shown below:

p3 = new Node('–' , p1, p2);



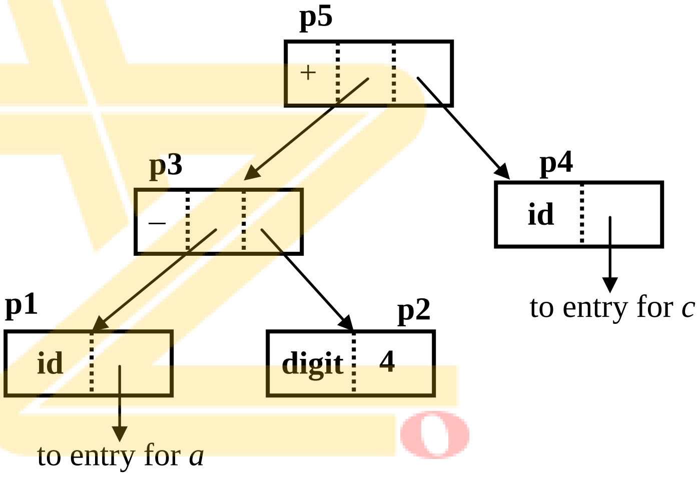
Step 4: Create a leaf node for identifier *c* using the function Leaf() as shown below:

p4 = new Leaf(id, id.entry);



Step 5: Create an interior node by passing '+' as the first argument and pointers to the left sub-tree identified by *p3* and pointer to the right sub-tree identified by *p4* with the help of function Node() as shown below:

p5 = new Node('+', p3, p4)



Thus, to get the above syntax tree for the expression "a – 4 + c", the various steps that are used are shown below:

p1 = new Leaf(**id** , *id.entry*);

p2 = new Leaf(**digit** , *4*);

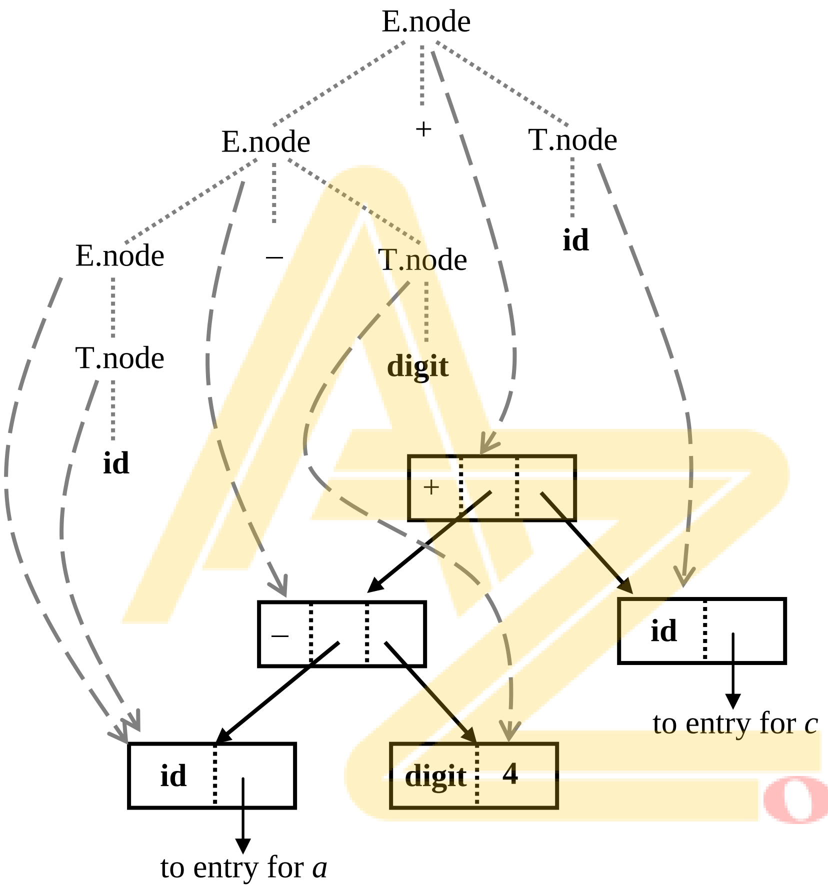
p3 = new Node(**'–'** , p1, p2);

p4 = new Leaf(id, id.entry);

p5 = new Node('+', p3, p4)

Note: If the above rules are evaluated during a postorder traversal of the parse tree or with reductions during bottom-up parse, then above sequence of steps ends with p5 pointing to the root of the constructed parse tree.

The parse tree, annotated parse tree depicting the construction of a syntax tree for the arithmetic expression “a - 4 + c” is shown below:



Observe the following points:

- The nodes of the syntax tree are shown as records with *op* as the first field
- Syntax tree edges are shown using solid lines
- The parse tree which need not actually be constructed is shown with dotted edges.
- The third type of line, shown dashed represents the values of E.node and T.node where each line points to the appropriate syntax-tree node
- At the bottom we see leaves for *a*, 4 and *c* constructed with the help of function Leaf()
- The interior nodes for the operators are created with the help of function Node()

5.6.1.2 Syntax tree for L-attributed definition

The method of constructing syntax tree for L-attributed definition remains same as the method of constructing syntax tree for S-attributed definition. The functions Leaf() and Node() are used with same number and type of parameters.

Example 5.13: Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using top-down approach with operators + and –

Solution: The grammar to generate an arithmetic expression with two operators + and – is shown below:

$$
\begin{aligned}
&E \rightarrow E + T \mid E - T \mid T \\
&T \rightarrow (E) \mid \text{digit} \mid \text{id}
\end{aligned}
$$

The above grammar is not suitable for top-down parser since it has left recursion. After eliminating left recursion, we get the following grammar:

$$
\begin{aligned}
&E \rightarrow T\,E' \\
&E' \rightarrow +\,T\,E' \mid -\,T\,E' \mid \epsilon \\
&T \rightarrow (E) \mid \text{digit} \mid \text{id}
\end{aligned}
$$

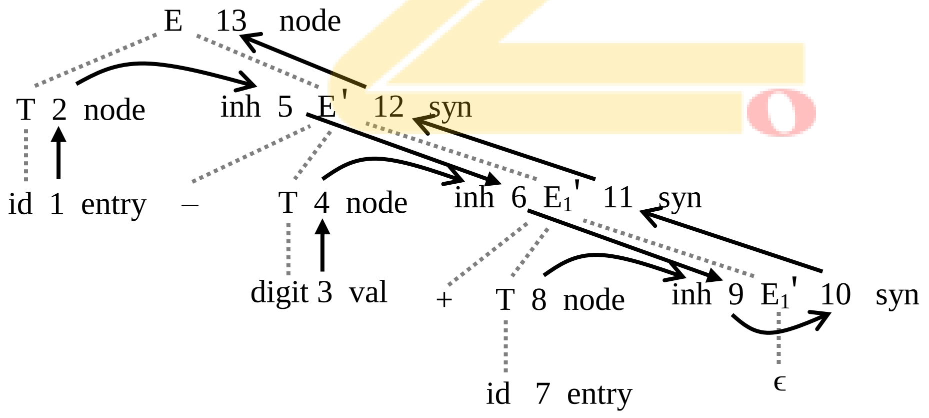
To construct syntax tree, first obtain the SDD for the above grammar. The SDD for the above grammar is shown below (For details refer example 5.4)

Production	**Semantic Rule**
$E \rightarrow T\,E'$	$E'.inh = T.val$ $E.val = E'.syn$
$E' \rightarrow +\,T\,E_1'$	$E_1'.inh = E'.inh + T.val$ $E'.syn = E_1'.syn$
$E' \rightarrow -\,T\,E_1'$	$E_1'.inh = E'.inh - T.val$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$
$T \rightarrow (\,E\,)$	$T.val = E.val$
$T \rightarrow \text{digit}$	$T.val = \text{digit.lexval}$
$T \rightarrow \text{id}$	$T.val = \text{id.entry}$

The functions Leaf() and Node() functions are used to create the syntax tree. Now, constructing syntax trees during top-down parsing can be obtained using the following semantic rules:

Production	Semantic Rule
$E \rightarrow T\,E'$	$E'.inh = T.node$ $E.node = E'.syn$
$E' \rightarrow +\,T\,E_1'$	$E_1'.inh$ = new Node('+', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
$E' \rightarrow -\,T\,E_1'$	$E_1'.inh$ = new Node('–', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$
$T \rightarrow (\,E\,)$	$T.node = E.node$
$T \rightarrow$ digit	$T.node$ = new Leaf(digit, digit.lexval)
$T \rightarrow$ id	$T.node$ = new Leaf(id, id.entry)

The annotated parse tree along with dependency graph for the expression "a – 4 + c" can be written as shown below:



Example 5.14: Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions with operators -, +, * and / using top-down approach

Solution: The grammar to generate an arithmetic expression is shown below:

$$
\begin{aligned}
&E \rightarrow E + T \mid E - T \mid T\\
&T \rightarrow T * F \mid T / F \mid F\\
&F \rightarrow (E) \mid \text{digit} \mid \text{id}
\end{aligned}
$$

The above grammar is not suitable for top-down parser since it has left recursion. After eliminating left recursion, we get the following grammar:

$$
\begin{aligned}
&E \rightarrow T\,E'\\
&E' \rightarrow +\,T\,E' \mid -\,T\,E' \mid \epsilon\\
&T \rightarrow F\,T'\\
&T' \rightarrow *\,F\,T' \mid /\,F\,T' \mid \epsilon\\
&F \rightarrow (E) \mid \text{digit} \mid \text{id}
\end{aligned}
$$

To construct syntax tree, first obtain the SDD for the above grammar. The SDD for the above grammar is shown below (For details refer example 5.4)

Production	Semantic Rule
$E \rightarrow T\,E'$	$E'.\text{inh} = T.\text{val}$ $E.\text{val} = E'.\text{syn}$
$E' \rightarrow +\,T\,E_1'$	$E_1'.\text{inh} = E'.\text{inh} + T.\text{val}$ $E'.\text{syn} = E_1'.\text{syn}$
$E' \rightarrow -\,T\,E_1'$	$E_1'.\text{inh} = E'.\text{inh} - T.\text{val}$ $E'.\text{syn} = E_1'.\text{syn}$
$E' \rightarrow \epsilon$	$E'.\text{syn} = E_1'.\text{inh}$
$T \rightarrow F\,T'$	$T'.\text{inh} = F.\text{val}$ $T.\text{val} = T'.\text{syn}$
$T' \rightarrow *\,F\,T_1'$	$T_1'.\text{inh} = T'.\text{inh} * F.\text{val}$ $T'.\text{syn} = T_1'.\text{syn}$
$T' \rightarrow /\,F\,T_1'$	$T_1'.\text{inh} = T'.\text{inh} / F.\text{val}$ $T'.\text{syn} = T_1'.\text{syn}$

$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$
$F \rightarrow id$	$F.val = id.entry$

The same functions Leaf() and Node() functions are used to create the syntax tree. Now, constructing syntax trees during top-down parsing can be obtained using the following semantic rules:

Production	**Semantic Rule**
$E \rightarrow T\, E'$	$E'.inh = T.node$ $E.node = E'.syn$
$E' \rightarrow +\, T\, E_1'$	$E_1'.inh = new\ Node('+', E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow -\, T\, E_1'$	$E_1'.inh = new\ Node('-', E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$
$T \rightarrow F\, T'$	$T'.inh = F.node$ $T.node = T'.syn$
$T' \rightarrow *\, F\, T_1'$	$T_1'.inh = new\ Node('*', T'.inh, F.node)$ $T'.syn = T_1'.syn$
$T' \rightarrow /\, F\, T_1'$	$T_1'.inh = new\ Node('/', T'.inh, F.node)$ $T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$
$F \rightarrow (E)$	$F.node = E.node$
$F \rightarrow digit$	$F.node = new\ Leaf(digit, digit.lexval)$
$F \rightarrow id$	$F.node = new\ Leaf(id, id.entry)$

5.6.2 The structure of a type

Now, let us see "What is the use of inherited attributes?" During top-down parsing, the grammar should not have left recursion. if the grammar has left recursion, we have to eliminate left recursion. The resulting grammar need inherited attributes. Inherited attributes are useful when the structure of the parse tree differs from the syntax tree for the specified input. The attributes can then be used to carry information from one part of the parse to another part of the parse tree. But sometimes, even though the grammar does not have left recursion, the language itself demands inherited attributes. This can be explained by considering array type as shown below:

Example 5.15: "Give the syntax directed translation of type **int [2][3]** and also given the semantic rules for the respective productions"

Solution: The grammar for multi-dimensional array can be written as shown below:

$$\begin{aligned}
&T \rightarrow B\ C\\
&B \rightarrow \textbf{int}\\
&B \rightarrow \textbf{float}\\
&C \rightarrow [\text{num}]\ C_1\\
&C \rightarrow \epsilon
\end{aligned}$$

Observe the following points from above grammar:

- The non-terminal T generates either a basic type such as **int** or **float**. This is achieved using the following derivation:

$$T \Rightarrow BC \Rightarrow \textbf{int}\ C \Rightarrow \textbf{int} \qquad \text{[Finally replacing C with } \epsilon]$$

or

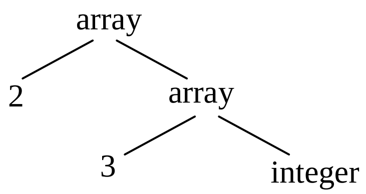
$$T \Rightarrow BC \Rightarrow \textbf{float}\ C \Rightarrow \textbf{float} \qquad \text{[Finally replacing C with } \epsilon]$$

- The non-terminal T also generates array of components consisting of a sequence of integers where each integer is surrounded by brackets as shown below:

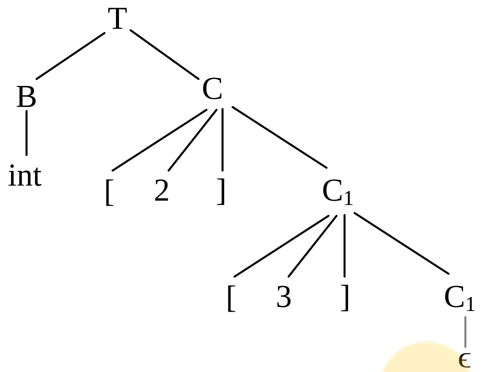
$$T \Rightarrow BC \Rightarrow \textbf{int}\ C \Rightarrow \textbf{int}\ [\text{num}]\ C \Rightarrow \textbf{int}\ [\text{num}]\ [\text{num}]\ C \qquad \Rightarrow \textbf{int}\ [\text{num}]\ [\text{num}]$$

Finally replacing C with ϵ

In C, **int [2][3]** is interpreted as "Array of 2 arrays of 3 integers" which can be denoted by array(2, array(3, integer)) can be written in the form of a tree as shown in the figure:



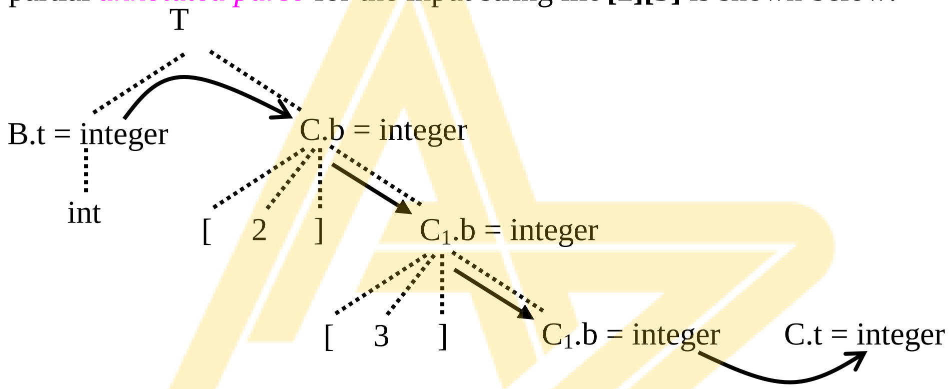
The parse tree to get the string **int [2][3]** is shown below:



It is very easy for us to write SDD, if we have annotated parse tree along with directions that gives the flow. So, let us use the following attribute names for the non-terminals:

- For the synthesized attribute, we use attribute name as *t*
- For the inherited attribute, we use attribute name as *b*

The partial *annotated parse* for the input string **int [2][3]** is shown below:



Observe the following points from above partial annotated parse tree:

- The type **int** is moved to parent B and attribute *t* is synthesized. The production and equivalent semantic rule is shown below:

Production	Semantic rule
B → **int**	B.t = integer
B → **float**	B.t = float

- The type *integer* has to be transferred from B.t on the left to C on the right using the production T → B C as shown using the arrow mark and hence it is inherited attribute denoted by *b*. The production and equivalent semantic rule is shown below:

Production	Semantic rule
T → B C	C.b = B.t

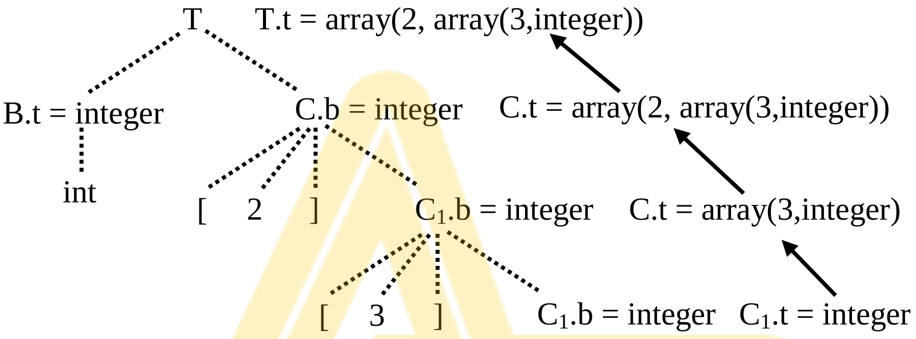
- Now, the attribute value of C.b moves down and copied into C_1 using the production C → [num] C_1. It must be inherited. The equivalent semantic rule is shown below:

Production	Semantic rule
C → [num] C_1	C_1.b = C.b

- Because of the production C → ϵ, the right most C in the above tree, takes its synthesized value C.t from itself using inherited attribute value C.b. The semantic rule is shown below:

Production	Semantic rule
C → ϵ	C.t = C.b

Now, the synthesized attribute value *integer* has to be moved from right most non-terminal to the root node as shown below in partial annotated parse tree:



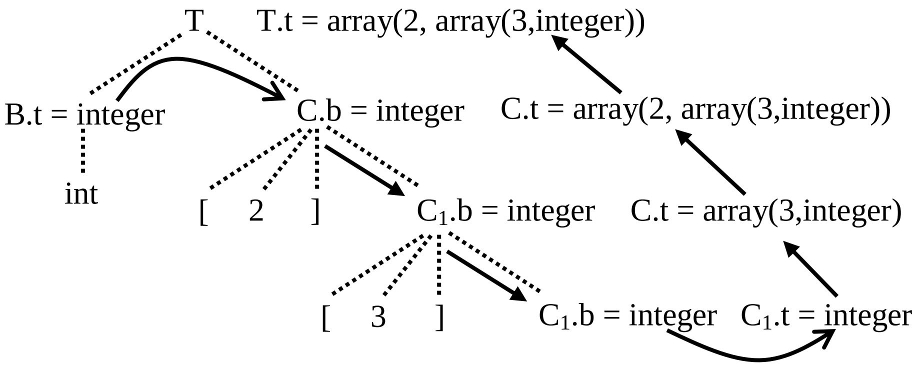
- The attribute value C_1.t is moved to C.t upwards using the production C → [num] C_1. The semantic rule can be written as shown below:

Production	Semantic rule
C → [num] C_1	C.t = array(num.val, C_1.t)

- Finally, attribute value of C.t is moved to T.t using the production T → B C and equivalent semantic rule is shown below:

Production	Semantic rule
T → B C	T.t = C.t

So, the final annotated parse tree is shown below:



The final semantic rules are shown below:

Production	Semantic rule	Type
T → B C	C.b = B.t	Inherited
	T.t = C.t	Synthesized
B → **int**	B.t = integer	Synthesized
B → **float**	B.t = float	Synthesized
C → [num] C_1	C_1.b = C.b	Inherited
	C.t = array(num.val, C_1.t)	Synthesized
C → ϵ	C.t = C.b	Synthesized

5.7 Syntax directed translation schemes

Now, let us see "What is syntax directed translation scheme?"

Definition: A syntax-directed translation scheme is a context free grammar with program fragments embedded within production bodies. The program fragments are called semantic actions and can appear at any position within the production body. By convention, semantic actions are enclosed within braces. If braces are used as grammar symbols in the production body then we quote them.

Note the following points:

- Any SDT can be implemented by first building a parse tree and then performing the actions in a pre-order manner.
- But, typically, SDT's are implemented during parsing, without building a parse tree
- The SDT's are used to implement two important classes of SDD's namely:
  1) Underlying grammar is LR-parsable and the SDD is S-attributed
  2) Underlying grammar is LL-parsable and the SDD is L-attributed
- The semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.

5.7.1 Postfix translation schemes

In this section, let us see "What is Postfix Syntax-directed-translation or Postfix SDT?"

Definition: In an SDD implementation, we parse the grammar bottom-up and the SDD is S-attributed. An SDT is constructed such that the actions to be executed are placed at the end of the production and are executed only when the right hand side of the production is reduced to left hand side of the production i.e., reduction of the body to the head of the production. The SDT's with all actions at the right end of the production bodies are called *postfix SDT's* or *postfix syntax-directed-translations.*

Example 5.16: Obtain Postfix SDT implementation of the desk calculator to evaluate the given expression

SDT can be easily obtained by looking at the SDD shown in example 5.7. Observe that all the semantic rules in SDD enclosed within braces results in SDT. The postfix SDT implementation of the desk calculator is shown below:

Productions	Actions
$S \rightarrow E\ n$	{ print(E.val) }
$E \rightarrow E_1 + T$	{ $E.val = E_1.val + T.val$ }
$E \rightarrow T$	{ $E.val = T.val$ }
$T \rightarrow T_1 * F$	{ $T.val = T_1.val * F.val$ }
$T \rightarrow F$	{ $T.val = F.val$ }
$F \rightarrow (E)$	{ $F.val = E.val$ }
$F \rightarrow digit$	{ $F.val = digit.lexval$ }

In above SDT observe that actions enclosed within braces are present at the end of each production.

Note: In a typical SDT, actions can be placed either at the beginning of the production or at the end of the production or somewhere in between. Now, instead of evaluating the arithmetic expression, let us convert the infix expression to postfix expression

Example 5.17: Write the SDD and annotate parse tree for converting an infix to postfix expression

The grammar to generate un-parenthesized arithmetic expression expressed using infix notation consisting of the operators + and * is shown below:

$E \rightarrow E_1 + T$
$E \rightarrow T$
$T \rightarrow T_1 * F$
$T \rightarrow F$
$F \rightarrow 0$
$F \rightarrow 1$
…….
…….
$F \rightarrow 9$

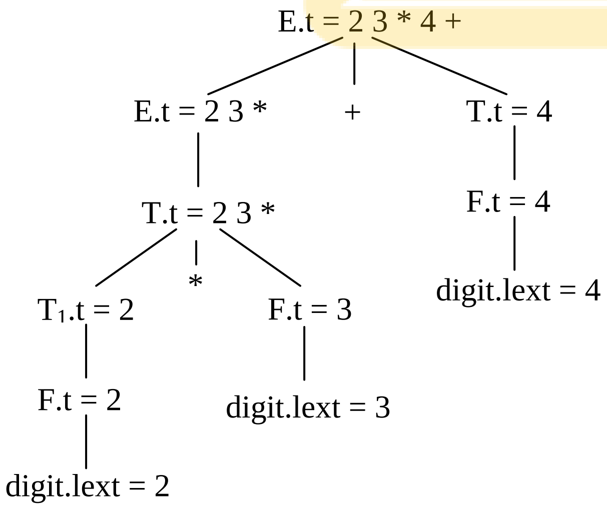
Note: Infix expression: a + b Postfix expression: a b +

Instead of writing the postfix expression as: ab + we write as: a || b || '+'
where the symbol || is treated as concatenation operator.

So, SDD for infix to postfix translation is written as shown below:

Production	Semantic rules (Postfix notation)
$E \rightarrow E_1 + T$	$E.t = E_1.t$ || $T.t$ || '+'
$E \rightarrow T$	$E.t = T.t$
$T \rightarrow T_1 * F$	$T.t = T_1.t$ || $F.t$ || '*'
$T \rightarrow F$	$T.t = F.t$
$F \rightarrow 0$	$F.t$ = '0'
$F \rightarrow 1$	$F.t$ = '1'
…….	
…….	
$F \rightarrow 9$	$F.t$ = '9'

Note: The semantic rule for the production $E \rightarrow E_1 + T$ is $E.t = E_1.t + T.t$. But, in the postfix notation, it can be written as: $E.t = E_1.t$ || $T.t$ || '+' where the symbol || is used as concatenation operator. The annotated parse tree for converting the expression 2*3+4 can be written by writing the parse tree for the given expression as shown below:



Example 5.18: Write the SDT for converting an infix to postfix expression. Show the actions for translating the expression 2*3+4 into its equivalent postfix expression

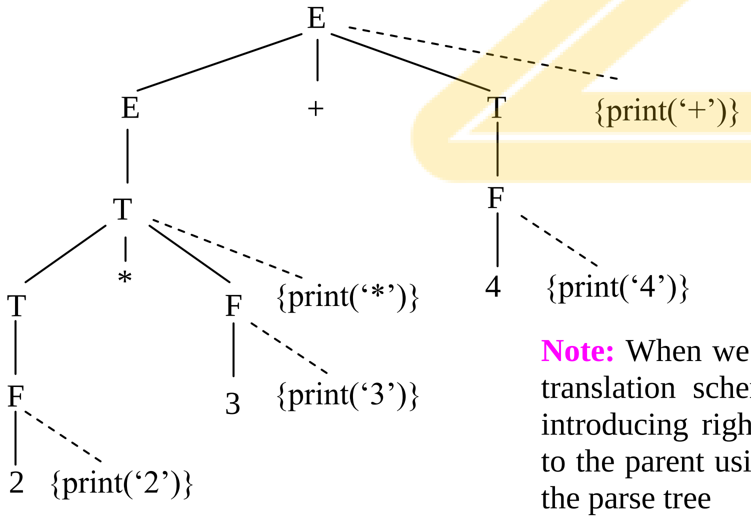
The grammar to generate un-parenthesized arithmetic expression expressed using infix notation consisting of the operators + and * is shown below:

$$
\begin{aligned}
&E \rightarrow E_1 + T\\
&E \rightarrow T\\
&T \rightarrow T_1 * F\\
&T \rightarrow F\\
&F \rightarrow 0\\
&F \rightarrow 1\\
&\ldots\ldots\\
&\ldots\ldots\\
&F \rightarrow 9
\end{aligned}
$$

To convert an infix expression to postfix expression, write the parse tree to get the expression 2*3+4 and print the operator or operand using the following rules:

- If only child is present (representing an operand), introduce one right sibling printing the operand
- If any one of the children contains an operator, introduce right most sibling to print the operator
- The rightmost sibling introduced is shown in dotted arrow for convenience

The parse tree to get the expression "2 * 3 + 4" is shown below:



Note: When we draw the parse tree for the translation scheme, indicate an action by introducing rightmost child and connect it to the parent using dashed line as shown in the parse tree

Now, traversing in postorder and executing only the actions we get the postfix expression:

2 3 * 4 +

Now, the SDT for converting an infix expression to postfix expression can be easily written using the above parse tree and is shown below:

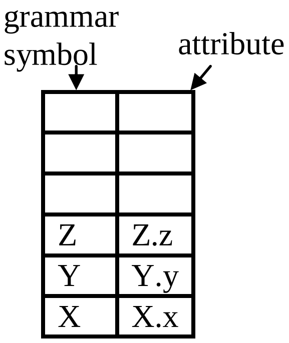
Actions present at the end of productions (Postfix)

$E \rightarrow E_1 + T$	{ print('+') }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ print('*') }
$T \rightarrow F$	
$F \rightarrow 0$	{ print('0') }
$F \rightarrow 1$	{ print('1') }
…….	
…….	
$F \rightarrow 9$	{ print('9') }

5.7.2 Parser-Stack Implementation of Postfix SDT's

Now, let us "Explain parser-stack implementation of Postfix SDT's?" Postfix SDT's can be implemented during LR parsing by executing the actions whenever reduction occurs. This can be explained as shown below:

- The grammar symbols to be reduced to LHS of the production are present on top of the stack. For example, for the production of the form: A → X Y Z and assume stack contains X, Y and Z and they are the symbols to be reduced
- Apart from placing grammar symbols on the top of the stack, place the attribute values of grammar symbols also on the top of the stack
- A grammar symbol may have more than one attribute and hence all the attributes associated with a grammar symbol should be passed on to the stack
- So, the stack should be implemented as an array of records where each record on the stack has the grammar symbol and its associated attributes. For the sake of convenience only one attribute of grammar symbol is shown in figure.
- If the attributes are synthesized and actions are present at the end of the production, then we can compute the attribute value of LHS of the production using the attribute values of RHS of the production. For example,



Z	Z.z
Y	Y.y
X	X.x

Stack

If we reduce by a production such as A → X Y Z, then we have all the attributes of X, Y and Z on the top of the stack. Remove those grammar symbols from the stack and push the grammar symbol A on to the stack along with its attribute.

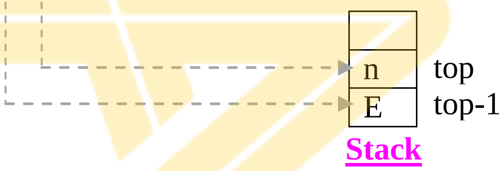
Example 5.18: Write the actions of desk calculator SDT so that they manipulate the parser explicitly

Solution: The SDT for desk calculator can be written as shown below: (See example 5.16 for details)

S → E n	{ print(E.val) }
E → E_1 + T	{ E.val = E_1.val + T.val }
E → T	{ E.val = T.val }
T → T_1 * F	{ T.val = T_1.val * F.val }
T → F	{ T.val = F.val }
F → (E)	{ F.val = E.val }
F → digit	{ F.val = digit.lexval }

The desk calculator can be implemented using the stack as shown below:

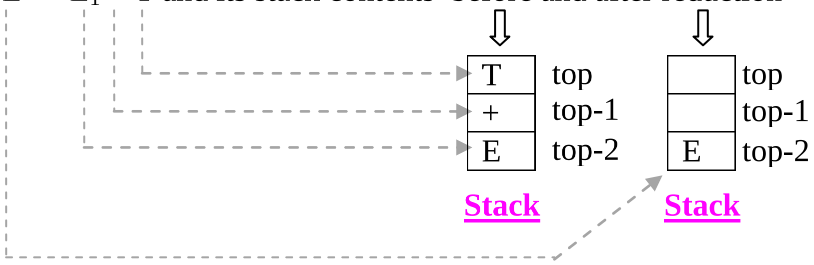
- Consider the production S → E n and its stack contents



Using the above SDD, the attribute value E.val which is in stack at position top-1 has to be printed. This can be done using the statement:

Print(stack[top-1]);
Top = top -1

- Consider the production E → E_1 + T and its stack contents before and after reduction



The equivalent statements are:

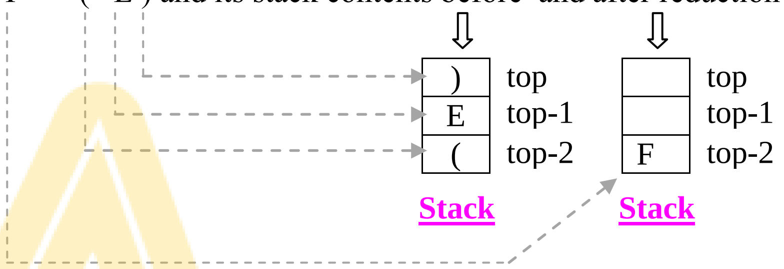
stack[top-2].val = stack[top].val + stack[top-2].val
top = top -2

Similarly, for the production T → T_1 * F, we can use the following statements:

stack[top-2].val = stack[top].val * stack[top-2].val
top = top -2

- But, for the productions E → T and T→ F, no action is necessary because the length of the stack will not change.

- Consider the production F → (E) and its stack contents before and after reduction



The equivalent statements are:

stack[top-2].val = stack[top-1].val
top = top -2

The final actions for the respective productions are shown below:

Productions	**Actions**

S → E n	{ print(stack[top-1]);
	top = top -1}

E → E_1 + T	{ stack[top-2].val = stack[top].val + stack[top-2].val top = top -2}
E → T	
T → T_1 * F	{ stack[top-2].val = stack[top].val * stack[top-2].val top = top -2}
T → F	
F → (E)	{ stack[top-2].val = stack[top-1].val top = top -2}
F → digit	

5.7.3 SDT's with actions inside productions

Now, let us see "When action part specified in the production is executed?" An action can be placed at any position within the body of the function. The action is performed immediately after all symbols to its left are processed. Thus, if we have the production:

$$B \rightarrow X \{ a \} Y$$

then action *a* is performed after we have recognized X (if X is a terminal) or all terminals derived from X (if X is a non-terminal)

Example 5.18: Write the SDT for converting an infix to prefix expression. Show the actions for translating the expression 2*3+4 into its equivalent prefix expression

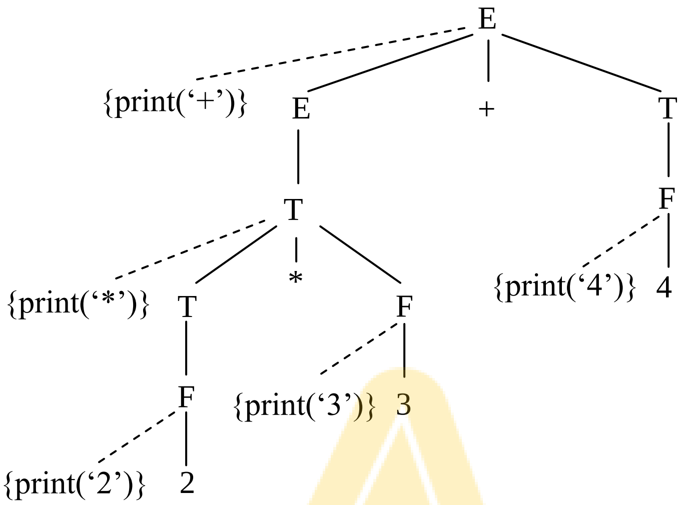
The grammar to generate un-parenthesized arithmetic expression expressed using infix notation consisting of the operators + and * is shown below:

$$E \rightarrow E_1 + T$$
$$E \rightarrow T$$
$$T \rightarrow T_1 * F$$
$$T \rightarrow F$$
$$F \rightarrow 0$$
$$F \rightarrow 1$$
$$\ldots\ldots.$$
$$\ldots\ldots.$$
$$F \rightarrow 9$$

To convert an infix expression to prefix expression, write the parse tree to get the expression 2*3+4 and print the operator or operand using the following rules:

- If only child is present (representing an operand), introduce one left sibling printing the operand
- If any one of the children contains an operator, introduce leftmost sibling to print the operator
- The leftmost sibling introduced is shown in dotted arrow for convenience

Note: When we draw the parse tree for the translation scheme, indicate an action by introducing leftmost child and connect it to the parent using dashed line as shown in the parse tree. The parse tree to get the expression "2 * 3 + 4" is shown below:



Now, traversing in preorder and executing only the actions we get the prefix expression:

+ * 2 3 4

Now, the SDT for converting an infix expression to postfix expression can be easily written using the above parse tree and is shown below:

Actions present in beginning of productions (Prefix)

E → { print('+') } E_1 + T

E → T

T → { print('*') } T_1 * F

T → F

F → 0 { print('0') }

F → 1 { print('1') }

…….

…….

F → 9 { print('9') }

Exercises:

1) What is semantic analysis? What is syntax directed definition (SDD)?

2) What is an attribute? Explain with example. What are the different types or classifications of attributes?

3) What is a semantic rule? Explain with example

4) What is synthesized attribute? Explain with example

5) What is inherited attribute? Explain with example

6) What is annotated parse tree? Explain with example

7) Write the SDD for the following grammar:

S → En where *n* represent end of file marker

E → E + T | T

T → T * F | F

F → (E) | **digit**

8) Write the grammar and syntax directed definition for a simple desk calculator and show annotated parse tree for the expression (3+4)*(5+6)

9) What is circular dependency when evaluating the attribute value of a node in an annotated parse tree

10) What is the use of inherited attributes

11) Obtain SDD and annotated parse tree for the following grammar using top-down approach:

T → T * F | F

F → digit

12) Obtain SDD for the following grammar using top-down approach:

S → En

E → E + T | T

T → T * F | F

F → (E) | digit

13) and obtain annotated parse tree for the expression (3 + 4) * (5 + 6)n

14) What is a dependency graph

15) Obtain the dependency graph for the annotated parse tree obtained in example 5.3

16) What is topological sort of the graph

17) Give the topological sort of the following dependency graph

18) What are different classes of SDD's that guarantee evaluation order

19) What is S-attributed definition

20) What is an attribute grammar

21) What is an L-attributed definition

22) What is a side effect in a SDD

23) How to control the side effects in SDD

24) Write the grammar and SDD for a simple desk calculator with side effect

25) Write the SDD for a simple type declaration and write the annotated parse tree and the dependency graph for the declaration "**float a, b, c**"

26) Write the SDD for a simple desk calculator. Write the annotated parse tree for the expression 3*5+4n

27) What is syntax directed translation

28) What is a syntax tree? What is the difference between syntax tree and parse tree?"

29) For the following grammar show the parse tree and syntax tree for the expression 3 *5 + 4:

    E → E + T | E – T | T
    T→ T*F | T/F | F
    F→ (E) | digit | id

30) How to construct semantic rules that help us to create syntax trees for the expressions?

31) Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using bottom up approach

32) Create a syntax tree for the expression "a – 4 + c"

33) Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using top-down approach with operators + and –

34) Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions with operators -, +, * and / using top-down approach

35) Give the syntax directed translation of type **int [2][3]** and also given the semantic rules for the respective productions

36) What is syntax directed translation scheme

37) What is Postfix Syntax-directed-translation or Postfix SDT

38) Obtain Postfix SDT implementation of the desk calculator to evaluate the given expression

39) Write the SDD and annotate parse tree for converting an infix to postfix expression

40) Write the SDT for converting an infix to postfix expression. Show the actions for translating the expression 2*3+4 into its equivalent postfix expression

41) Explain parser-stack implementation of Postfix SDT's

42) Write the actions of desk calculator SDT so that they manipulate the parser explicitly

43) When action part specified in the production is executed

44) Write the SDT for converting an infix to prefix expression. Show the actions for translating the expression 2*3+4 into its equivalent prefix expression