# THEORY OF COMPUTATION AND COMPILERS

## Unit - III

## SEMANTIC ANALYSIS, INTERMEDIATE CODE GENERATOR & SYMBOL TABLE

### SEMANTIC ANALYSIS 

- Attributed Grammars
- Syntax Directed Translation

### INTERMEDIATE CODE GENERATOR

- Intermediate Forms of Source Programs - Abstract Syntax Tree, Polish Notation and Three Address Codes
- Intermediate Code Forms
- Type Checker

### SYMBOL TABLE

- Symbol Table Format
- Organization for Block Structures Languages
- Hashing

**Dr. R. Madana Mohana**
**Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning**
**CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY**
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

# SEMANTIC ANALYSIS

## Syntax-Directed Translation (SDT)

3.1.1. Syntax-Directed Definitions (SDD)

3.1.2. Evaluation Orders for SDD's

3.1.3. Applications of Syntax-Directed Translation

3.1.4. Syntax-Directed Translation Schemes

# Syntax Analysis

## Syntax-Directed Translation (SDT)

## Syntax-Directed Definitions (SDD)

- Inherited and Synthesized Attributes
- Evaluating an SDD at the Nodes of a Parse Tree

# Semantic Analysis

- Let us concentrate on the third phase of the compiler called **`semantic analysis`**.
- The main goal of the **`semantic analysis`** is to check the 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 on syntactically valid combination.
- In **`semantic analysis`** we check whether they form a sensible set of instructions in the programming language.

# 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.

# Semantic Analysis

**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 in **intermediate code generation**.
- Display appropriate **error messages**.

# Semantic Analysis

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

- **Syntax Directed Definition (SDD)**
- **Syntax Directed Translation (SDT)**

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

$$\mathbf{E \rightarrow E_1 + T}$$

# Semantic Analysis

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

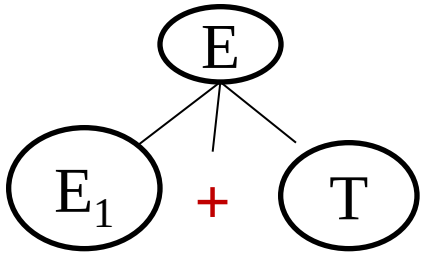
**Production:**

**$E \rightarrow E_1 + T$**

**Derivation:**

**$E \Rightarrow E_1 + T$**

**Parse Tree:**



# Semantic Analysis

- The non-terminal **E** on **LHS** of the production is called "**head of the production**".
- The string of grammar symbols "$\mathbf{E_1}$ **+ 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**.

# 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.

# Syntax Directed Definition (SDD)

**Example:**

A simple **Syntax Directed Definition (SDD)** for the production $E \rightarrow E_1 + T$ can be written as shown below:

| Production | Semantic Rule |
|---|---|
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$<br>Where **val** is **attribute** |

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

# Syntax Directed Definition (SDD)

**Attribute: 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 filed in the record or a structure.

# Syntax Directed Definition (SDD)

## Attribute: Examples

- **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.

# Syntax Directed Definition (SDD)

**Attribute:**

**Typical examples of attributes are:**

- The data types associated with variables 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 **procedure**
- The number of significant digits in a **number** and so on.

# Syntax Directed Definition (SDD)

## Attributed Grammar: Definition

**Attribute grammar** is a special form of context-free grammar where some additional information (**attributes**) are appended to one or more of its non-terminals in order to provide context-sensitive information.

Each **attribute** has well-defined domain of values, such as integer, float, character, string, and expressions.

**Attribute grammar** is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.

**Attribute grammar** (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

# Syntax Directed Definition (SDD)

**Semantic rule: 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:

$E.val = E_1.val + T.val$ // **Semantic rule**

# 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.

There are two types of attributes namely:

- **Synthesized attribute (S-attribute)**
- **Inherited attribute (I-attribute)**

# Inherited and Synthesized Attributes

**Synthesized Attribute (S-attribute):**

***Definition:***

The attribute value for 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.

# Inherited and Synthesized Attributes

**Synthesized Attribute (S-attribute):**

***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 vales $E_1$**.val** and **T.val** appearing on the RHS (body) of the production as shown below:

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

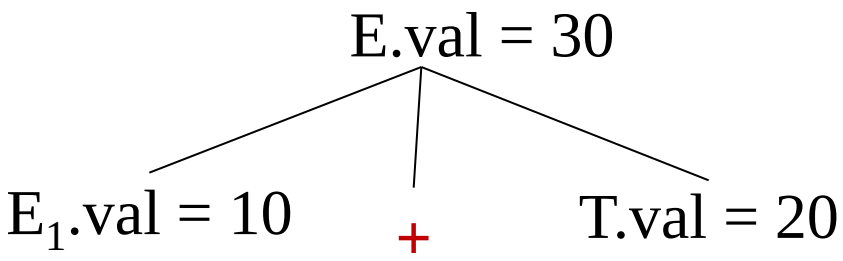
# Inherited and Synthesized Attributes

**Synthesized Attribute (S-attribute):**

***Example:***

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

**Parse tree with attribute values:**

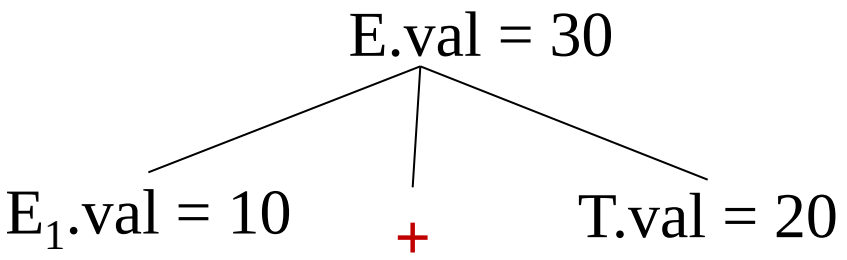


# Inherited and Synthesized Attributes

**Synthesized Attribute (S-attribute):**

***Example:***

**Parse tree with attribute values:**




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.

# Inherited and Synthesized Attributes

**Inherited Attribute (I-attribute):**

***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.

# Inherited and Synthesized Attributes

**Inherited Attribute (I-attribute):**

***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.

# Inherited and Synthesized Attributes

**Inherited Attribute (I-attribute):**

***Example:***

The production, semantic rule and parse tree along with attribute values are shown below:

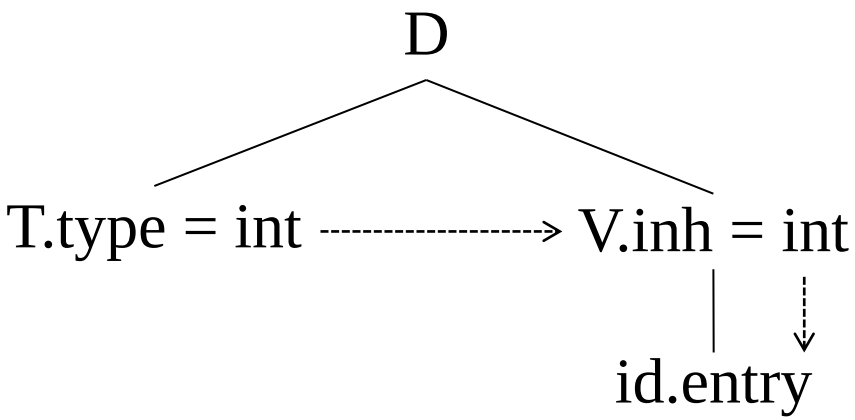
| Production | Semantic Rule |
|---|---|
| D → T V | V.inh = T.type |

# Inherited and Synthesized Attributes

**Inherited Attribute (I-attribute):**

***Example:***

**Parse tree with attribute values:**



# Inherited and Synthesized Attributes

**Inherited Attribute (I-attribute):**

***Example:***

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**.

# Inherited and Synthesized Attributes

**Inherited Attribute (I-attribute):**

***Example:***

Observe the following points from the above parse tree:

- On similar line, the value **int** stored on **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.

# Inherited and Synthesized Attributes

**Annotated parse tree:**

***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** (**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.

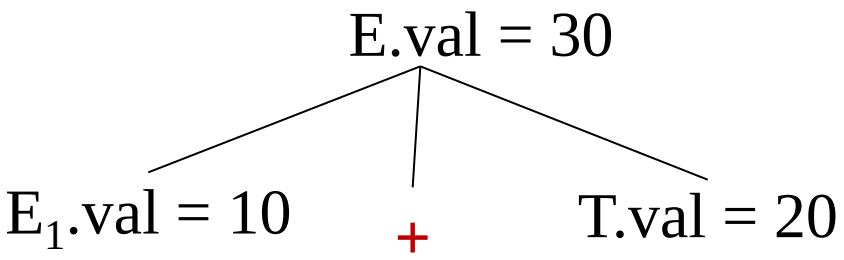
**Note:** Terminals can never have inherited attributes.

# Inherited and Synthesized Attributes

**Annotated parse tree:**

***Example:***

Consider the partial annotated parse 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.

# Inherited and Synthesized Attributes

**Example-1:**

Write the **SDD** for a simple desk calculator and show annotated parse tree for the expression (3+4)*(5+6)n

**S → En**

**E → E + T | E - T | T**

**T → T * F | T / F | F**

**F → ( E )| digit**

# Inherited and Synthesized Attributes

**Example:** ***Solution***

The given grammar is 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 *****. For the sake of convenience, let us consider part of the grammar written as shown below:

# Inherited and Synthesized Attributes

**Example:** ***Solution***

**S → Tn**

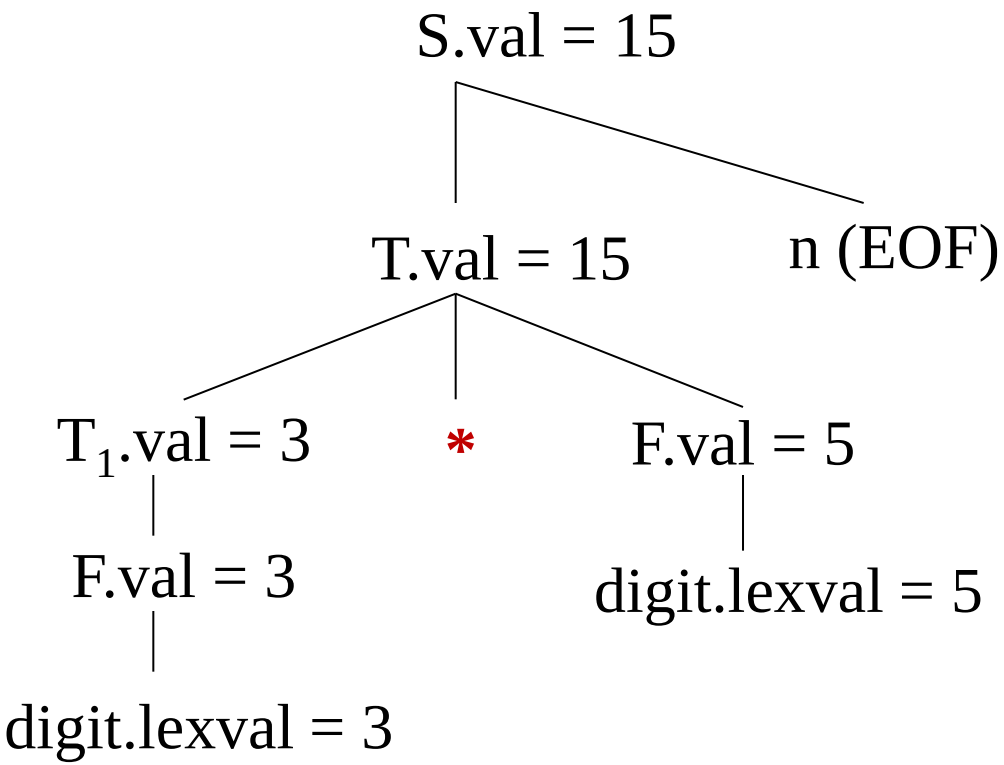
**T → T * F | T / F | F**

**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:

# Inherited and Synthesized Attributes

**Example:** ***Solution***



# Inherited and Synthesized Attributes

**Example:** ***Solution***

It is very easy to see how the values **3** and **5** are moved from bottom to top 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 |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| S → Tn | S.val = T.val |

# Inherited and Synthesized Attributes

**Example:** ***Solution***

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

| Productions | Semantic Rules |
|---|---|
| $S \rightarrow En$ | $S.val = E.val$ |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | $E.val = T.val$ |
| $F \rightarrow ( E )$ | $F.val = E.val$ |

# Inherited and Synthesized Attributes

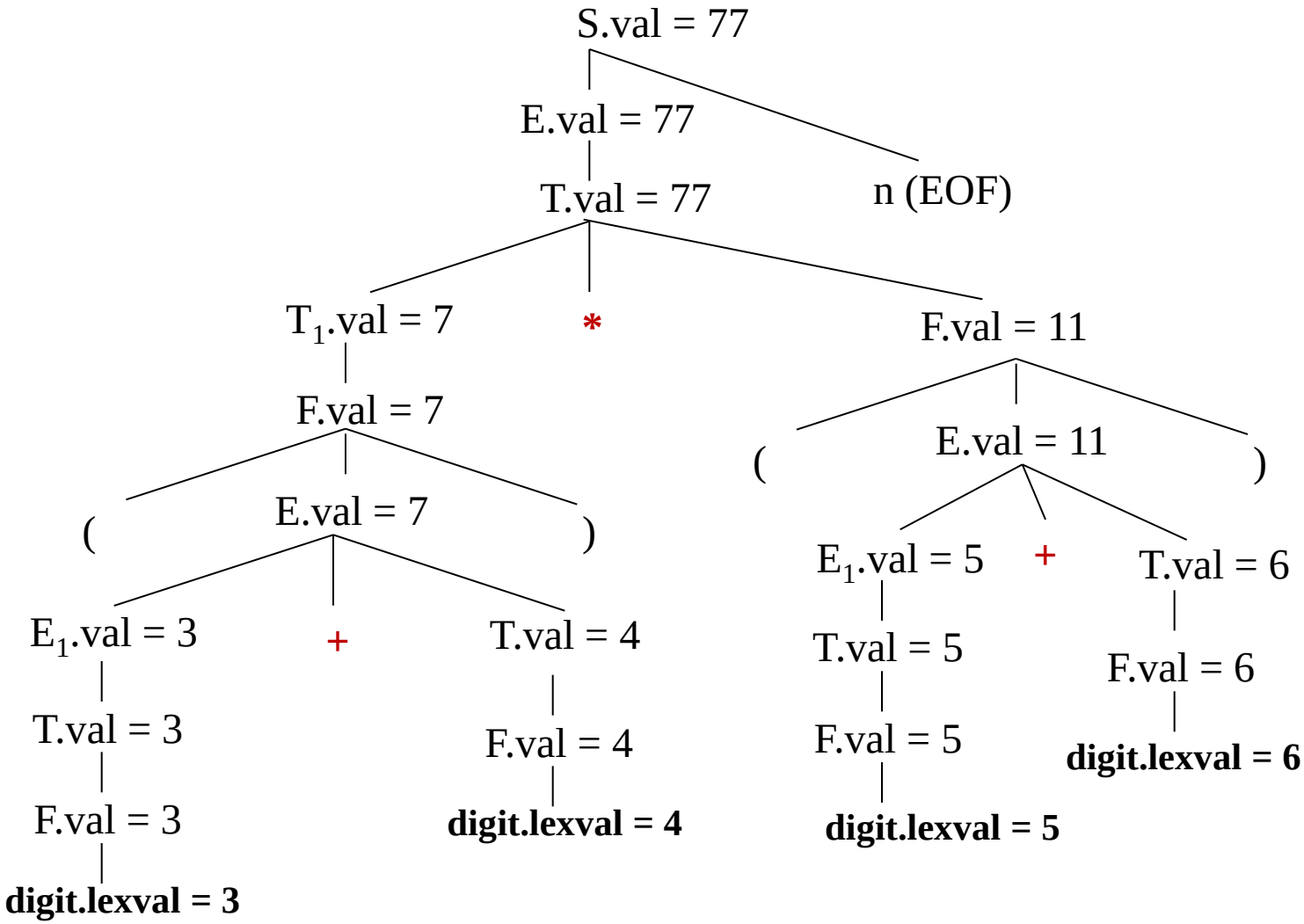
**Example:** ***Solution***

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 → 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 |

# Inherited and Synthesized Attributes

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



# Evaluating an SDD at the Nodes of a Parse Tree

We can easily obtain an **SDD** using the following 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**.

# Evaluating an SDD at the Nodes of a Parse Tree

***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.

# Evaluating an SDD at the Nodes of a Parse Tree

## *Circular dependency*

- 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.

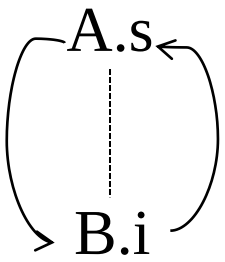
# Evaluating an SDD at the Nodes of a Parse Tree

## *Circular dependency*

- 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:

| Productions | Semantic Rules |
|---|---|
| A → B | A.s = B.i<br>B.i = A.s + 6 |

**Partial annotated parse tree:**



# Evaluating an SDD at the Nodes of a Parse Tree

***Circular dependency***

| Productions | Semantic Rules |
|---|---|
| A → B | A.s = B.i<br>B.i = A.s + 6 |

- In the above semantic rule ***s*** is synthesized attribute and ***i*** is inherited attribute.
- The above two semantic rules are circular in nature.
- To compute ***A.s*** we require the values 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.

# Evaluating an SDD at the Nodes of a Parse Tree

***Why evaluate inherited attributes***

Consider the following grammar

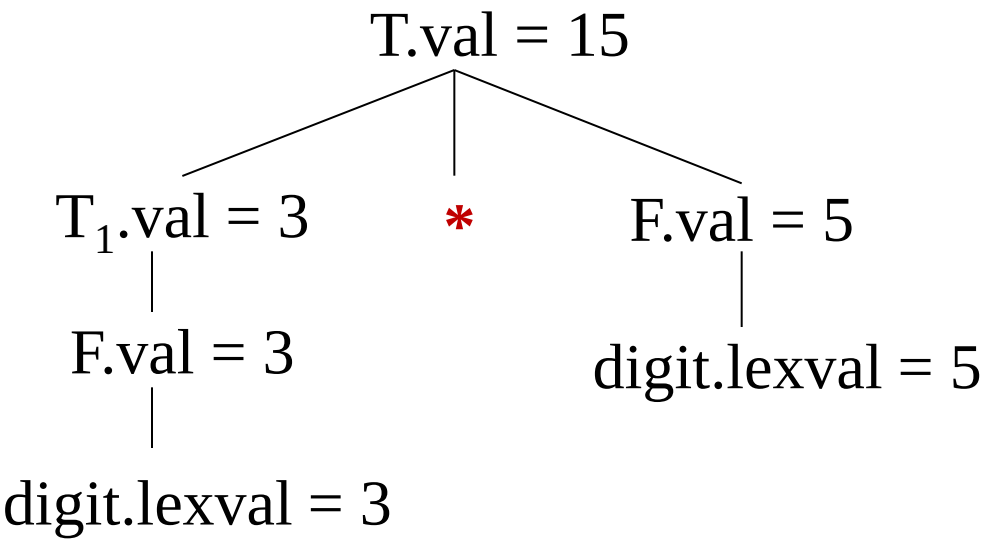
**T → T * F | F**

**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 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:

# Evaluating an SDD at the Nodes of a Parse Tree

***Why evaluate inherited attributes***




It is very easy to see how the values **3** and **5** are moved from bottom to top till we reach the root node to the value **15** as shown in the above tree.

# Evaluating an SDD at the Nodes of a Parse Tree

***Why evaluate inherited attributes***

| Semantic Rules | Productions |
|---|---|
| F.val = digit.lexval | F → id |
| 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.

# Evaluating an SDD at the Nodes of a Parse Tree

**Example:**

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

**S → En**

**E → E + T | T**

**T → T * F | F**

**F → ( E )| digit**

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

# Evaluating an SDD at the Nodes of a Parse Tree

**Example:** ***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, the following grammar is obtained:

**S → En**

**E → TE′**

**E′ → +TE′ | ϵ**

**T → FT′**

**T′ → *FT′ | ϵ**

**F → ( E ) | digit**

# Evaluating an SDD at the Nodes of a Parse Tree

**Example:** ***Solution***

**Note:**

The variables **S**, **E**, **T** and **F** are present both in given grammar and grammar obtained after eliminating 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.

# Evaluating an SDD at the Nodes of a Parse Tree

**Example:** ***Solution***

Consider the following productions:

**S → En**

**F → ( E )| 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 value of RHS (i.e., children) for the above productions and hence they have synthesized attributes.

# Evaluating an SDD at the Nodes of a Parse Tree

**Example:** ***Solution***

The productions, semantic rules and type of the attribute are shown below:

| Production | Semantic Rule | Type |
|---|---|---|
| S → E n | S.v = E.v | Synthesized |
| F → ( E ) | F.v = E.v | Synthesized |
| F → d | F.v = digit.lexval | Synthesized |

# Evaluating an SDD at the Nodes of a Parse Tree

## Example: *Solution*

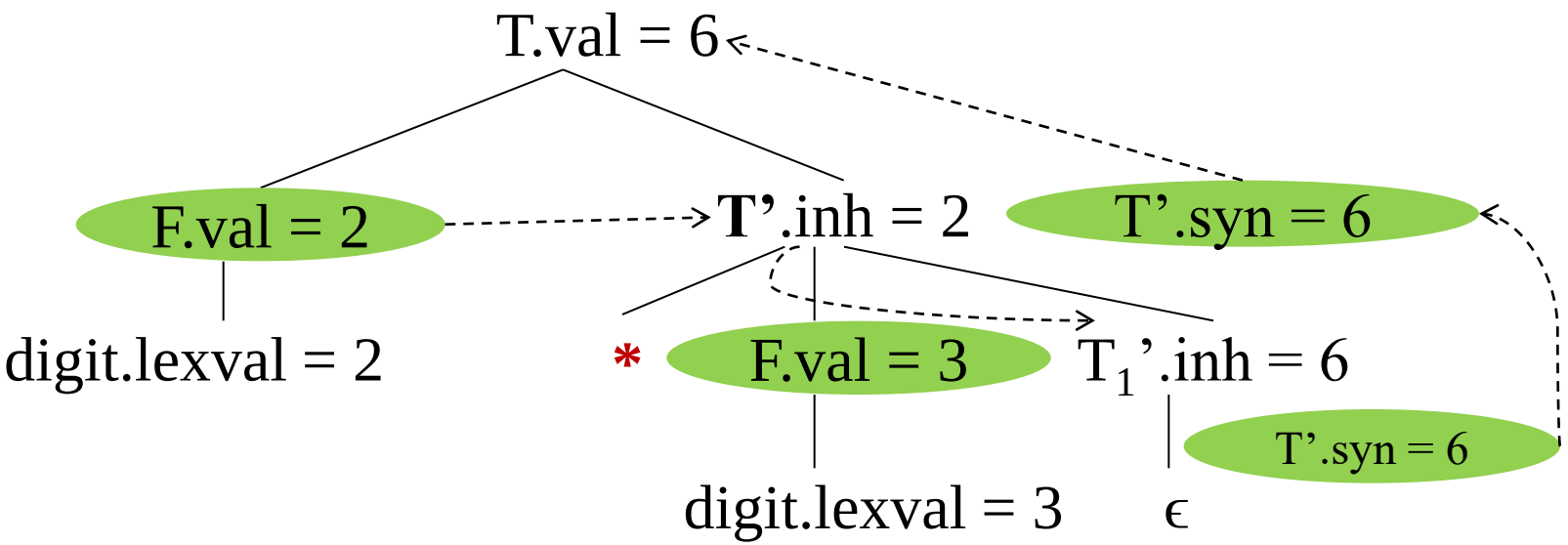
Consider the following productions and draw the annotated parse tree for the expression **2*3** with flow of information as shown below:

| Productions |
| --- |
| T → F T′ |
| T′ → * F T′ \| ϵ |
| F → digit |

# Evaluating an SDD at the Nodes of a Parse Tree

**Example:** ***Solution***

Annotated parse tree for the expression **2*3**:



# Evaluating an SDD at the Nodes of a Parse Tree

## Example: *Solution*

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 → F T′ |
| T′.inh * F.val is copied to $T_1'$.inh | $T_1'$.inh = T′.inh * F.val | T′ → * F T′ |
| $T_1'$.inh is copied to T′.syn | T′.syn = $T_1'$.inh | T′ → ϵ |
| T′.syn is moved to its parent | T′.syn = $T_1'$.syn | T′ → * F T′ |
| T′.syn is moved to its parent T | T.val = T′.syn | T → F T′ |

# Evaluating an SDD at the Nodes of a Parse Tree

## Example: *Solution*

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

| Production | Semantic Rules |
|---|---|
| $T \rightarrow F\ T'$ | $T'.inh = F.val$<br>$T.val = T'.syn$ |
| $T' \rightarrow *\ F\ T'$ | $T_1'.inh = T'.inh * F.val$<br>$T'.syn = T_1'.syn$ |
| $T' \rightarrow \epsilon$ | $T'.syn = T_1'.inh$ |

# Evaluating an SDD at the Nodes of a Parse Tree

## Example: *Solution*

Similar to the above, we can write the semantic rules for the other productions as shown below:

| Production | Semantic Rules |
|---|---|
| $E \rightarrow T\ E'$ | $E'.inh = T.val$<br>$E.val = E'.syn$ |
| $E' \rightarrow +\ T\ E'$ | $E_1'.inh = E'.inh + T.val$<br>$E'.syn = E_1'.syn$ |
| $E' \rightarrow \epsilon$ | $E'.syn = E_1'.inh$ |

# Evaluating an SDD at the Nodes of a Parse Tree

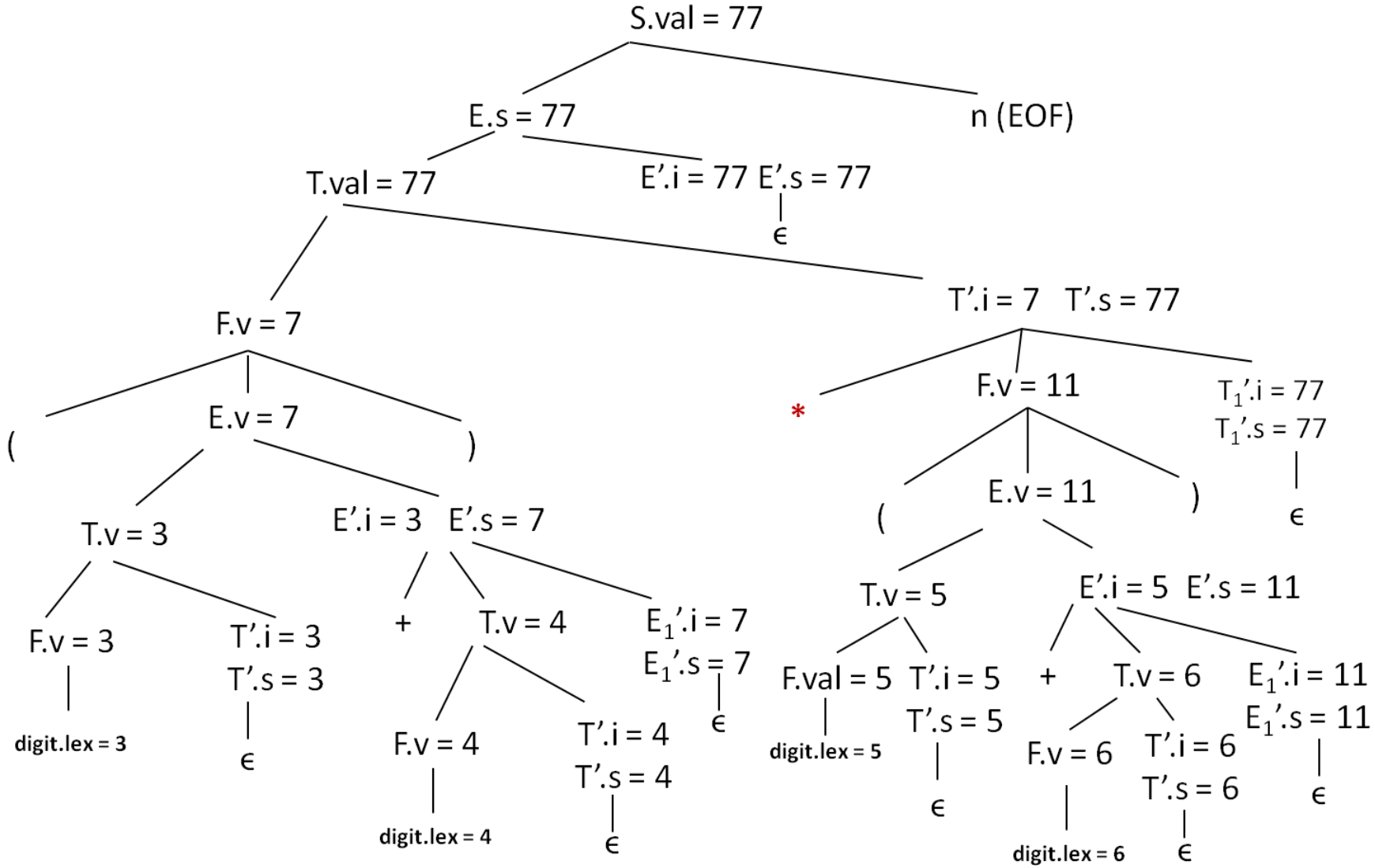
## Example: *Solution*

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

| Production | Semantic Rule | Type |
|---|---|---|
| S → E n | S.v = E.v | Synthesized |
| E → T E′ | E′.inh = T.val | Inherited |
| | E.val = E′.syn | Synthesized |
| E′ → + T E′ | $E_1$′.inh = E′.inh + T.val | Inherited |
| | E′.syn = $E_1$′.syn | Synthesized |
| E′ → ϵ | E′.syn = $E_1$′.inh | Synthesized |
| T → F T′ | T′.inh = F.val | Inherited |
| | T.val = T′.syn | Synthesized |
| T′ → * F T′ | $T_1$′.inh = T′.inh * F.val | Inherited |
| | T′.syn = $T_1$′.syn | Synthesized |
| T′ → ϵ | T′.syn = $T_1$′.inh | Synthesized |
| F → ( E ) | F.v = E.v | Synthesized |
| F → d | F.v = digit.lexval | Synthesized |

# Evaluating an SDD at the Nodes of a Parse Tree

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



# Summary...

## Syntax-Directed Definitions (SDD)

- Inherited and Synthesized Attributes
- Evaluating an SDD at the Nodes of a Parse Tree

*Reading:* Aho2, Section 5.1.1 & 5.1.2

*Next Lecture:* Evaluation Orders for SDD's