SYSTEM SOFTWARE and COMPILERS- 18CS61

Prof. Sampada K S, Assistant Professor
DEPT. OF CSE | RNSIT

MODULE-5

Syntax Directed Translation, Intermediate code generation, Code generation Text book 2: Chapter 5.1, 5.2, 5.3, 6.1, 6.2, 8.1, 8.2

SEMANTIC ANALYSIS

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)

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).
2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions:

The syntax-directed definition (SDD) is a CFG that includes attributes and rules. In an augmented CFG, the attributes are associated with the grammar symbols (i.e. nodes of the parse tree). And the rules are associated with the productions of grammar.

• **Syntax Directed Definitions** are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of **Attributes**;

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

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

attributes

Attribute is a property of a programming language construct. Associated with grammar symbols.

Such formalism generates Annotated **Parse-Trees** where each node of the tree is a record with a field for each attribute. If X is a grammar symbol and 'a' is a attribute then X.a denote the value of attribute 'a' at a particular node X in a parse tree.

- 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.

2. Productions are associated with **Semantic Rules** for computing the values of attributes.

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

Example:

E.val= E1.val+T.val //Semantic rule

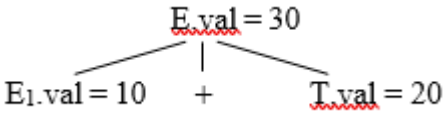
where E.val on RHS can be computed using E1.val and T.val on RHS

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:

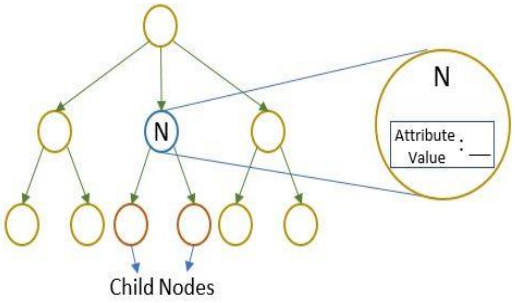
- *Synthesized attribute*
- *Inherited attribute*

Synthesized Attributes: 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 → E1 + T. Suppose, the attribute value val of E on LHS (head) of the production is obtained by adding the attribute values E1.val and T.val appearing on the RHS (body) of the production as shown below:

Production	Semantic Rule	Parse tree with attribute values
E → 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.



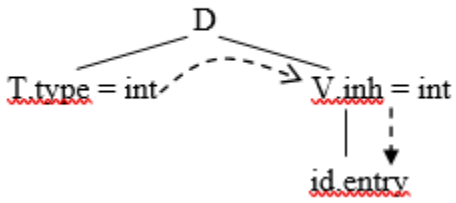
SDD with Synthesized Attributes

PRODUCTION	SEMANTIC RULE
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	$F.val :=$digit.$lexval$

Inherited attribute : 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:

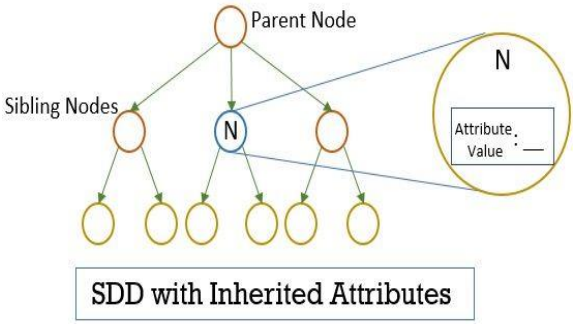
- The type int obtained from the lexical analyzer is already stored in T.type whose value is transferred to its sibling V. This can be done using:

  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.



SDD with Inherited Attributes

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow$ int	$T.type :=$ *integer*
$T \rightarrow$ real	$T.type :=$ *real*
$L \rightarrow L_1$, id	$L_1.in := L.in$; *addtype*(id.*entry, L.in*)
$L \rightarrow$ id	*addtype*(id.*entry, L.in*)

Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for "type declarations":

• The non terminal T has a synthesized attribute, type, determined by the keyword in the declaration.
• The production D → T L is associated with the semantic rule L.in := T .type which set the inherited attribute L.in.
• Note: The production L → L_1, id distinguishes the two occurrences of L.
To evaluate translation rules, identify the best-suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals (because SDT rules don't impose any specific order on evaluation)

5.1 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.

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

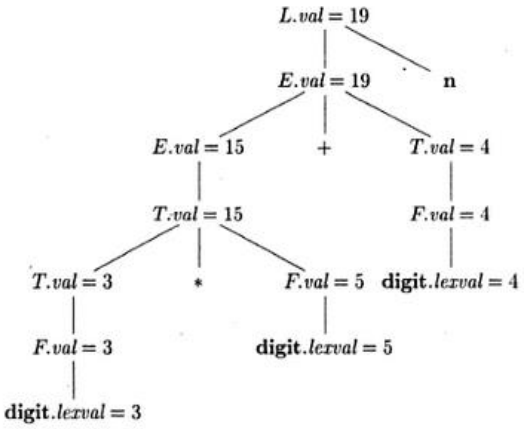
- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

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

Consider the SDD

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

Here we can see the production rules of grammar along with the semantic actions. And the input string provided by the lexical analyzer is 3 * 5 + 4 n.



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

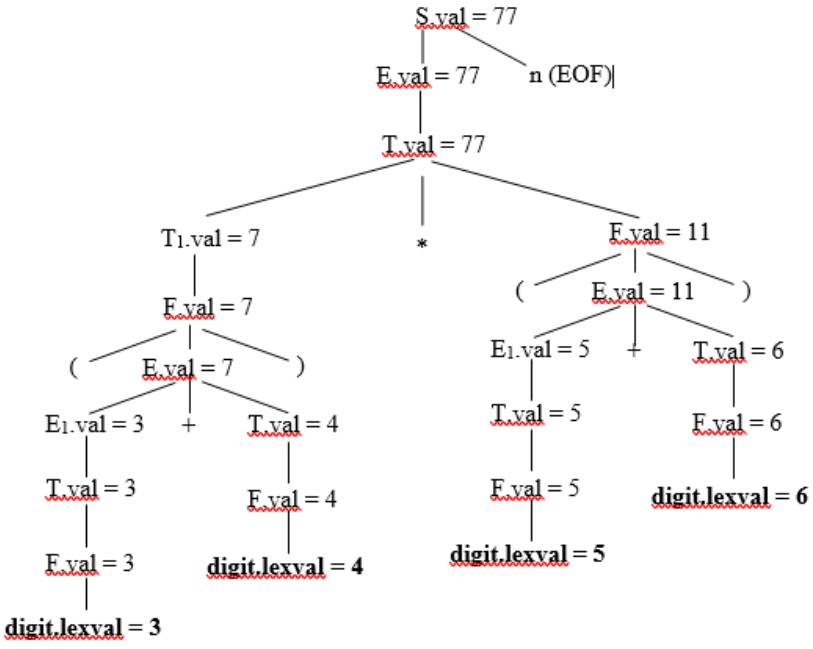
Productions	Semantic Rules
L → En	L.val = E.val
E → E + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.lexval

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

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

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
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → T_1 / F	T.val = T_1.val + F.val
T → F	E.val = T.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 foreach non-terminal is shown below:



Dependency Graph

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.

Example of Dependency Graph:

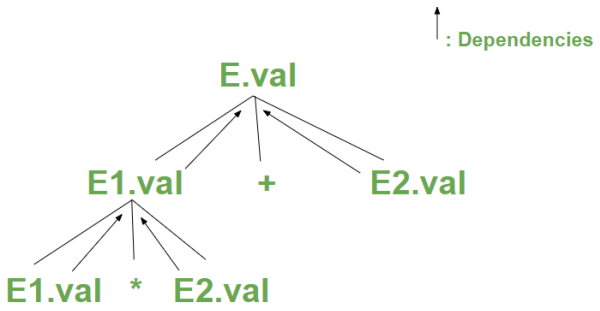
Design dependency graph for the following grammar:

E -> E1 + E2

E -> E1 * E2

PRODUCTIONS	SEMANTIC RULES
E -> E1 + E2 E -> E1 * E2	E.val -> E1.val + E2.val E.val -> E1.val * E2.val

Required dependency graph for the above grammar is represented as –



Evaluation Orders for SDD

"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.

There can be two classes of syntax-directed translations

- S-attributed translation
- L-attributed translation.

6.3 S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

Evaluation Order. Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

- 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:

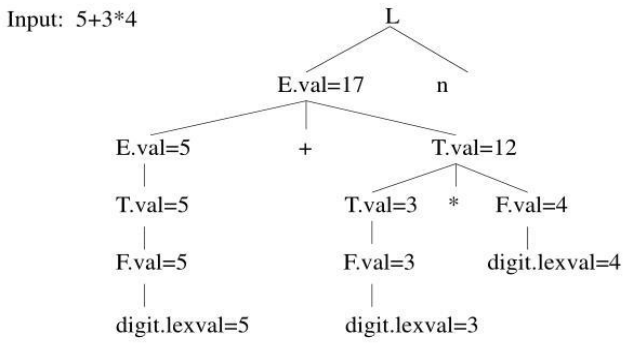
```
postorder (N)
{
for (each child C of N from left)
      postorder(C)
end for
evaluate the attributes associated with node N
}
```

- **Example.** The arithmetic grammar is an example of an S-Attributed Definition.

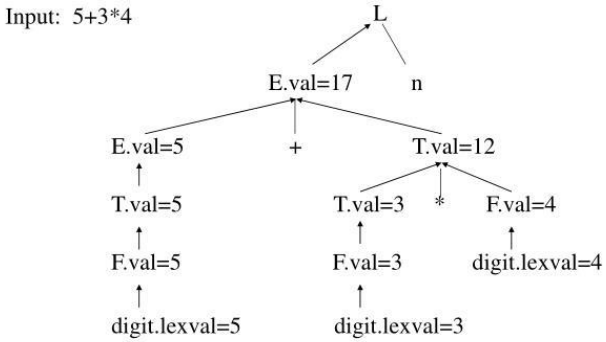
PRODUCTION	SEMANTIC RULE
$L \rightarrow E$n	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	$F.val :=$digit.$lexval$

The annotated parse-tree for the input 5+3*4 is:

Annotated Parse Tree -- Example



Dependency Graph



6.4 L-attributed definition

Definition: An L-attributed definition is one of the following:
Synthesized or Inherited, but with following rules.
Consider the production A → $X_1 X_2 \ldots\ldots Xn$ and let Xi.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, $X_3 \ldots . X_{i-1}$ appearing to the left of X_i have eithersynthesized or inherited attributes

or

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

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

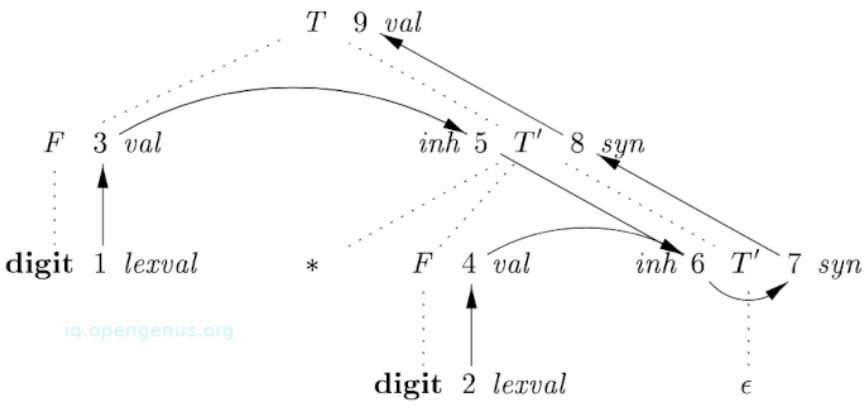
Productions	Semantic rules
$T \rightarrow F\ T'$	$T'.inh = F.val$
$T' \rightarrow *\ F\ T_1'$	$T_1'.inh = T'.inh * F.val$

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.

PRODUCTION	SEMANTIC RULES
T → FT'	T'.inh = F.val
T' → * FT1'	T1'.inh = T'.inh x F.val, T'.syn = T1'.syn
T' → ϵ	T'.syn = T'.inh
F → digit	F.val = digit.lexval

Now its complete dependency graph.



"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.

Semantic rules with controlled side effects

"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 becontrolled 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.valas 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.

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: