MODULE 20 - Semantic Phase, Dependency Graph, Topological sorting

In this module we discuss the semantic phase of the compiler and understand the changes that happen in the semantic phase of the compiler.

20.1 Functions of the semantic phase of the compiler

The syntax phase of the compiler converted the input into a derivation tree and validated whether the input belongs to the grammar or not. Thus the output of the parser will be a derivation tree. This derivation tree needs to have information to generate code. The information is derived from the grammar of the programming language. The necessary information needs to be added to the derivation tree and this derivation tree is converted to a representation which is easier to generate assembly language code. Syntax Directed Definition (SDD) and Syntax Directed Translation (SDT) helps in converting the parse tree to an annotated parse tree which has information to generate code as well as the order of evaluating the derivation tree.

Thus SDD and SDT are the primary goals of the semantic phase of the compiler. After converting the derivation tree to a parse tree, the semantic phase of the compiler helps in writing semantic rules to verify the semantic correctness of all the statements. Type checking, flow checking are some of the semantic correctness verifying information. Figure 20.1 shows the integration of the lexical and syntactic phase of the compiler with the semantic phase.

As can be seen from figure 20.1, the YACC specification file could incorporate the semantic rules associated to perform SDD and SDT. Let us discuss the features of SDD and SDT in detail in this module.

20.2 Syntax Directed Definition

A syntax-directed definition adds set of semantic rules to productions. Terminals and non-terminals have attributes. A depth-first traversal algorithm is used to compute the values of the attributes in the parse tree using the semantic rules. After the traversal is completed, the attributes contain the translated form of the input.
For each production semantic rules are formulated to convert the derivation tree to another representation. The semantic rules for the expression grammar is given in Table 20.1. As an expression evaluates to a value, every grammar symbol is associated with the value and this is described in the semantic rule of the expression grammar.

Table 20.1 Semantic rules for Expression Grammar

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>L → E n</td>
<td>print (E.val)</td>
<td>This is the initial production. This simply assigns the value that is computed by the variable E to variable L.</td>
</tr>
<tr>
<td>E → E1 + T</td>
<td>E.val := E1.val + T.val</td>
<td>There is a variable E on both LHS and RHS of the production. To distinguish the LHS ‘E’ from the RHS ‘E, we use E1 for the RHS E. The semantic rule describes us to compute the addition of the values indicated in the RHS and assign it to the LHS variable</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val := T.val</td>
<td>This production simply copies the value of the RHS variable to the LHS variable’s value field</td>
</tr>
<tr>
<td>T → T1 * F</td>
<td>T.val := T1.val * F.val</td>
<td>Similar to the production E → E+T. Here we compute the product of the RHS variable T and F and assign it to the LHS variable</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val := F.val</td>
<td>Copies the value of the RHS into the LHS variable</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.val := E.val</td>
<td>The parenthesis is to give precedence to the expression E but however, for deriving the value the LHS value is assigned to the RHS variable F</td>
</tr>
<tr>
<td>F → digit</td>
<td>F.val := digit.lexval</td>
<td>This is the terminating production. Hence, the semantic rule describes getting the value of digit from the symbol table or the value itself (if it is a constant) and assign it to the LHS</td>
</tr>
</tbody>
</table>

The derivation tree and the manner in which the value of the expression is being evaluated is given in figure 20.2 for the expression “9+5+2”.

The derivation for this expression would be

E → E +T → E+T+T → T+T+T → F +F+F → 9 + 5 +2 → (20.1)

This is shown in the derivation tree of figure 20.3 with semantic rules annotated at every node.
Figure 20.2 Annotated derivation tree for the expression.

From figure 20.2, we can see that the value propagates in a bottom up fashion. The derivation described in equation (20.1) is evaluated from right to left, where the value of 9, 5, 2, propagates to the three variable F, F, F which is then copied to T, T, T using the rule T.val := F.val and then we evaluate it using E.val := E1.val + T.val to assign it to the LHS variable E. The algorithm to evaluate this is given in Algorithm 20.1

Algorithm 20.1

Visit_node (n : node)
{
    For each child m of n, from left to right
        Visit m;
    Evaluate semantic rules at node n;
    Return
}

The statement “visit m” of algorithm 20.1, describes evaluating the semantic rule at node “m” and propagates the value to its parent till the value is reached to the root.

20.3 Attributes to SDD

Attribute values can represent numbers (literal constants), Strings (literal constants), memory locations, such as a frame index of a local variable or function argument, data type for type checking of expressions, scoping information for local declarations and intermediate program representations. These attributes could be categorized into two groups: Synthesized and Inherited attributes.
Given a production

\[ A \rightarrow \alpha \]

then each semantic rule is of the form

\[ b := f(c_1, c_2, \ldots, c_k) \]

where \( f \) is a function and \( c_i \) are attributes of \( A \) and \( \alpha \), and either \( b \) is a synthesized attribute of \( A \) or \( b \) is an inherited attribute of one of the grammar symbols in \( \alpha \). We derive the value of ‘\( b \)’ from the production using the semantic rule.

An attribute is said to be synthesized if, for a node \( N \), for a non-terminal \( A \), attributes are defined by value of the children and the node itself. Thus we can conclude that terminals have only synthesized attributes. An attribute is said to be inherited if for a node \( N \), the value of this node is defined by the parent and/or siblings.

For the expression grammar, the value at a particular node is defined by its children and thus the non-terminals \( E, T, F \) are all synthesized attributes. Terminal ‘id’ is synthesized due to the terminating condition. Let us consider the variable declaration grammar. This grammar declares two types of variable, int and real. The grammar, semantic rule and attribute type of all the grammar symbols are given in Table 20.2

<table>
<thead>
<tr>
<th>Production</th>
<th>Attribute</th>
<th>Semantic Rule</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D \rightarrow TL )</td>
<td>( L ) is inherited</td>
<td>( L.\text{in} := T.\text{type} )</td>
<td>Type of ( L ) depends on its sibling’s type ( T ). Thus ( L ) is inherited attribute</td>
</tr>
<tr>
<td>( T \rightarrow \text{int} )</td>
<td>( T ) is synthesized and so is ( \text{int} )</td>
<td>( T.\text{type} := \text{int} )</td>
<td>int is a synthesized attribute as it is a terminal. ( T ) gets the type ‘int’ from its child and hence ( T ) is a synthesized attribute</td>
</tr>
<tr>
<td>( T \rightarrow \text{real} )</td>
<td>( T ) is synthesized, real is synthesized</td>
<td>( T.\text{type} := \text{real} )</td>
<td>( T ) is synthesized</td>
</tr>
<tr>
<td>( L \rightarrow L_1, \text{id} )</td>
<td>( L_1 ) will be the inherited attribute</td>
<td>( L_1.\text{in} \geq L.\text{in}; ) addtype(id.entry, L.in)</td>
<td>( L_1 ) and id are siblings of the root ( L ). ( L ) will have the type as an attribute. The attribute of ( L_1 ) depends on the attribute of id.</td>
</tr>
<tr>
<td>( L \rightarrow \text{id} )</td>
<td>( \text{id} ) is the synthesized attribute</td>
<td>addtype(id.entry, L.in)</td>
<td>( \text{id} ) is the terminal and hence is synthesized</td>
</tr>
</tbody>
</table>

The annotated parse tree for the declaration grammar is given in figure 20.3
Figure 20.3 Annotated parse tree for the declaration grammar

Figure 20.3 shows an example of declaring variables as real. We could see that there are four variable and the type real propagates from the left sub-tree’s variable T to the right sub-tree ‘L’ and gets propagated all the way down to the variable id. It can also be observed that T is a synthesized attribute while L is an inherited attribute as it depends on its sibling ‘T’ for its type.

In addition the attributes of SDD can be classified as S-attributed definition and L-attributed definition. A syntax-directed definition that uses synthesized attributes exclusively is called an S-attributed definition (or S-attributed grammar). Thus a parse tree of an S-attributed definition can be annotated with a simple bottom-up traversal. YACC parser supports S-attributed definitions. While, a syntax-directed definition is L-attributed if each inherited attribute of \( X_j \) on the right side of \( A \rightarrow X_1 \, X_2 \ldots \, X_n \) depends only on

1. the attributes of the symbols \( X_1, X_2, \ldots, X_{j-1} \)
2. the inherited attributes of \( A \)

An L-attributed definition does a normal order of evaluating attributes which is depth-first and left to right. Every S-attributed syntax-directed definition is also L-attributed. Consider figure 20.4 with the production \( A \rightarrow XY \). We would consider that the value of \( Y \) depends on the value of \( X \) and thus \( Y \) is inherited. On the other hand, if the value of \( X \) depends on the value of \( A \), the parent then \( X \) is said to be a synthesized attribute. In figure 20.4, the direction of arrow shows how the value is being propagated from one grammar symbol to the other.

Figure 20.4 Example indicating inherited and synthesized attribute
Thus we can summarize that the grammar symbols can either have synthesized or inherited attributes based on how they derive their values from. A grammar can have a combination of synthesized and inherited attributes. A grammar can also be said to be S-attributed or L-attributed depending upon whether a grammar has synthesized attributes alone or a combination of synthesized and inherited attributes.

### 20.4 Acyclic dependency graph

The parse tree can be annotated with synthesized or inherited attributes. The parse tree can also be indicated with an arrow mark to indicate the manner in which the value gets propagated between the nodes of the parse tree. This graph is called as dependency graph as it indicates the dependency between nodes for deriving the values. This graph is an acyclic graph which doesn’t have a cycle. The presence of a cycle indicates that the graph is incorrect as the dependence of nodes for deriving values cannot be predicted. Edges in the dependence graph show the evaluation order for attribute values and thus the graph is a directed one.

Consider figure 20.5 which shows the example of a dependency graph for a sample grammar.

![Figure 20.5 Example dependency graph](image)

Figure 20.5 is the dependency graph for a grammar \( A \rightarrow XY \). As can be seen from the first representation the value of \( A \) is derived from its children \( X \) and \( Y \) and thus there is an arrow from \( X, Y \) to \( A \). The equation indicates that \( A \) is derived from a function \( f() \) using \( X \) and \( Y \). Similarly from the second representation of figure 20.5, the value of \( X \) is derived from \( A \) and \( Y \) and thus there is an arrow from \( A \) and \( Y \) to \( X \). The third representation indicates the value of \( Y \) is dependent on \( A \) and \( X \).

Let us consider the variable declaration grammar. We have already seen the manner in which the variable \( T, L \) gets their attribute. This is shown in figure 20.6 as a dependency graph indicating
the order of the nodes.

\[ \text{Figure 20.6 Dependency graph for Declaration grammar} \]

Figure 20.6 indicates that the type of T is derived from its child. The variable L is inherited and thus derives the type from its sibling T which is indicated as an arrow from T to L. This type of L propagates all the way down till the last variable L which in turn set the type of the variable to real. Figure 20.6 also indicates that the graph is acyclic and thus the representation is correct. The algorithm for constructing dependency graph is given in algorithm 20.2.

Algorithm 20.2

\[
\begin{array}{l}
\{ \\
\quad \text{For each node ‘n’ in the parse tree} \\
\quad \quad \{ \\
\quad \quad \quad \text{For each attribute ‘a’ of the grammar symbol at ‘n’} \\
\quad \quad \quad \quad \text{Construct a node in the dependency graph for ‘a’;} \\
\quad \quad \} \\
\quad \} \\
\} \\
\{ \\
\quad \text{For each node ‘n’ in the parse tree} \\
\quad \quad \{ \\
\quad \quad \quad \text{For each semantic rule b: } = f(c_1, c_2, \ldots c_n) \\
\quad \quad \quad \quad \text{Associated with each production used at ‘n’} \\
\quad \quad \quad \quad \quad \{ \text{for } i \geq 1 \text{ to } n \\
\quad \quad \quad \quad \quad \quad \text{Construct an edge from node } c_i \text{ to } b; \\
\quad \quad \quad \quad \} \\
\quad \} \\
\} \\
\}
\]

As given in algorithm 20.2, the first for loop constructs nodes of the graph. The second for loop observes the semantic dependency and constructs the edge from all the dependent node to its depended node.

20.5 Topological Sorting

In general, we construct an annotated parse tree to indicate the semantic value of every node and then represent it as a dependency graph to indicate the manner in which every node derives its
values. After constructing the dependency graph, the graph needs to be traversed to actually derive values for each node. This is done with the help of topological sorting.

A topological sort of a directed acyclic graph (DAG) is any ordering $m_1, m_2, \ldots, m_n$ of the nodes of the graph, such that if $m_i \rightarrow m_j$ is an edge then $m_i$ appears before $m_j$. Any topological sort of a dependency graph gives a valid evaluation order for the semantic rules.

Consider the declaration grammar whose dependency graph is given in figure 20.6. If we evaluate this graph and perform topological sorting, then the sequence of nodes is as indicated in figure 20.7. We need to get the values of the variable id1, id2, id3. We begin with id1, as it is the leaf node of the annotated parse tree. Thus the topological sort indicates these three variables are in the order 1, 2, 3. Now these values should be assigned a data type. To do this we need to get the type of T and thus the 4th position in the topological sort is by this node T. This value is propagated to L1 which would be the 5th position and this assigns id3 as real in the 6th position of the topological sort. The 7th position is by L2 which gets its value from L1 and 8th position by id2 which gets its value from L2. In a similar situation, L3 and id3 gets position 9 and 10 respectively of the topological sort.

The sequence of the topological sort is summarized below:

1. Get id1.entry
2. Get id2.entry
3. Get id3.entry
4. $T_1$.type=’real’
5. $L_1$.in=$T_1$.type
6. addtype(id3.entry, $L_1$.in)
7. $L_2$.in=$L_1$.in
8. addtype(id2.entry, $L_2$.in)
9. $L_3$.in=$L_2$.in
10. addtype(id1.entry, $L_3$.in)
The translation scheme to incorporate the topological sort is given below.

\[ D \rightarrow T \{ L.in := T.type \} L \]
\[ T \rightarrow \text{int} \{ T.type := \text{‘integer’} \} \]
\[ T \rightarrow \text{real} \{ T.type := \text{‘real’} \} \]
\[ L \rightarrow \{ L_1.in := L.in \} L_1, id \{ \text{addtype}(id.entry, L.in) \} \]
\[ L \rightarrow id \{ \text{addtype}(id.entry, L.in) \} \]

### 20.6 Syntax Directed Translation (SDT)

Syntax directed translation is also part of the semantic phase of the compiler wherein the input will be a derivation tree which is an n-ary tree and output will be a binary tree. A parse tree is called a *concrete syntax tree* because it doesn’t provide many details on the values associated with every node. An *abstract syntax tree* (AST) is defined by the compiler writer as a more convenient intermediate representation. An abstract syntax tree is a binary tree in which every node has at most two children.

To convert the derivation tree to syntax tree the compiler defines semantic rules associated with every production of a grammar construct. The semantic rules to convert the derivation tree to syntax tree are given in Table 20.3. It uses two functions, mknode() and mkleaf() which are make node and make leaf respectively. An interior node will have two children and the leaf node will have one child. The leaf node’s child happens to be typically symbol table look up and the interior node will have to store pointers to the left and right children.

*Table 20.3 Semantic rules to convert to a syntax tree*

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E1 + T</td>
<td>E.nptr := mknode(‘+’, E1.nptr, T.nptr)</td>
<td>LHS node is created with three fields, data field to consist of the operator and two pointers to the left and right children respectively. The parse tree would have three children, +, E1, T and now the operator has been shifted to the node’s data field and thus E1, T are the two children.</td>
</tr>
<tr>
<td>E \rightarrow E1 - T</td>
<td>E.nptr := mknode(‘-’, E1.nptr, T.nptr)</td>
<td>This is similar to the E \rightarrow E1+T production but the operator is replaced by ‘-‘</td>
</tr>
<tr>
<td>E \rightarrow T</td>
<td>E.nptr := T.nptr</td>
<td>Interior node just having one child the right pointer is assumed to be null and the operator field is empty indicating no operation</td>
</tr>
<tr>
<td>T \rightarrow T1 * id</td>
<td>T.nptr := mknode(‘*’, T1.nptr, mkleaf(id, id.entry))</td>
<td>This is similar to E \rightarrow E1+T but here one operand is a symbol table entry and thus will be a leaf node and it is generated using mkleaf() function</td>
</tr>
</tbody>
</table>
Table 20.3 is explained in figure 20.8 showing each derivation tree is converted to a binary tree with multiple data fields in each node for a simple expression id+id*id.

Figure 20.8 Syntax tree synthesis of a simple expression id+id*id

Figure 20.8 also ensures that the operator precedence is maintained where the ‘*’ operator node is at a deeper level than the ‘+’ operator. Thus if a post order traversal of this binary is done then we get the correct order of evaluation of this expression.

Summary:

The semantic phase of the compiler deals with SDD and SDT and this has been discussed in this module. The various types of attributes like synthesized and inherited have been discussed leading to construction of the acyclic dependency graph and use of topological sorting to evaluate this graph was also discussed. The next module would discuss the type checking of productions which are also part of the semantic phase of the compiler.