COMPILER CONSTRUCTION (Syntax Directed Translation) **Fall 2019**

Lecture 23: Attributed Grammars

Instructor: K.R. Chowdhary *: Professor of CS*

23.1 Inherited and Synthesized Attributes

A *syntax-directed definition* (SDD) is a context-free grammar together with, *attributes* and *rules*. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labelled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. Attributes may be of any kind: numbers, types, table references, or strings, for example, The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

We shall deal with two kinds of attributes for nonterminals:

1. A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
2. An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

Accordingly, we do not allow an inherited attribute at node N to be defined in terms of attribute values at the children of node N. But, we do allow a synthesized attribute at node N to be defined in terms of inherited attribute values at node N itself.

Terminals can have synthesized attributes, but not inherited attributes. Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal, however, it exists in the non-terminals, as semantics of a non-terminal is computed using the semantics of its children.

Example 23.1 *Syntax-directed definition of a simple desk calculator.*

The SDD in Table 23.1 is based on the familiar grammar for arithmetic expressions with operators + and $*$. It evaluates expressions terminated by an endmarker "**n**". In the SDD,

each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

Table 23.1: Syntax-directed definition (SDD) of a simple desk calculator (S-attributed).

S.No.	Production	Semantic Rules
1.	$L \to E$ **n**	$L.val = E.val$
2.	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3.	$E \to T$	$E.val = T.val$
4.	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5.	$T \to F$	$T.val = F.val$
6.	$F \to (E)$	$F.val = E.val$
7.	$F \to$ **digit**	$F.val =$ **digit**$.lexval$

The production rule 1, $\to E$ **n**, sets $L.val$ to $E.va1$, which we shall see is the numerical value of the entire expression (the various symbols stand for: L: left associative, E: expression, T: term, E_1: instance of an expression, and T_1: instance of a term).

Production rule 2, $E \to E_1 + T$, also has one rule, which computes the *val* attribute for the head E as the sum of the values at E_1 and T. At any parse-tree node N labeled E, the value of *val* for E is the sum of the values of *val* at the children of node N labeled E and T.

Production rule 3, $E \to T$, has a single rule that defines the value of *val* for E to be the same as the value of *val* at the child for T. Production rule 4 is similar to the second production; its rule multiplies the values at the children instead of adding them. The rules for productions 5 and 6 copy values at a child, like that for the third production. Production 7 gives $F.val$ the value of a **digit**, that is, the numerical value of the token digit that the lexical analyzer returned. □

An SDD that involves only synthesized attributes is called *S-attributed*; the SDD in Table 23.1 has S-attributed property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

Example 23.2 *Construction of Annotated parse tree for* $3 * 5 + 4$ **n**.

Figure 23.1 shows an annotated parse tree for the input string $3 * 5 + 4$ **n**, constructed using the grammar and rules of Table 23.1. The values of *lexval* are presumed supplied by the lexical analyzer.

Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled $*$, after computing $T.val = 3$ and $F.val = 5$ at its first and third children, we apply the rule that says $T.val$ is the product of these two values, or 15. □

Inherited attributes are useful when the structure of a parse tree does not "match" the abstract syntax of the source code. The next example shows how inherited attributes can be used to overcome such a mismatch due to a grammar designed for parsing rather than translation.

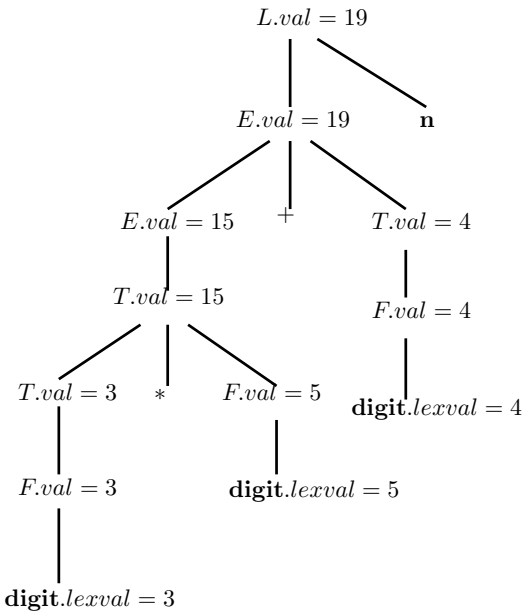

Figure 23.1: Annotated parse tree for $3 * 5 + 4$ **n**.

23.2 Evaluating attributes through SDD

To visualize the translation specified by an SDD, it is helpful if we work with parse trees. However, even though a translator need not actually build a parse tree. Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s), we called as an *annotated parse tree*.

How do we construct an annotated parse tree? In what order do we evaluate attributes? Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends. For example, if all attributes are synthesized, as in Example 23.1, then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a post-order traversal of the parse tree; the evaluation of S-attributed definitions is discussed later sections.

For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes. For instance, consider non-terminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, respectively, along with the production and rules as follows:

Production :
$A \to B$

$$\begin{aligned} &\textit{Semantic Rules}: \\ &A.s = B.i \\ &B.i = A.s + 1 \end{aligned}$$

We note that these rules are circular, hence, it is impossible to evaluate either $A.s$ at a node N or $B.i$ at the child of N without first evaluating the other. The circular dependency of $A.s$ and $B.i$ at some pair of nodes in a parse tree is shown in Figure 23.2.

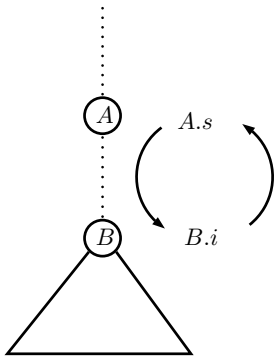

Figure 23.2: The circular dependency of $A.s$ and $B.i$ on one another.

It is computationally difficult to determine whether or not there exist any circularities in any of the parse trees that a given SDD could have to translate. Fortunately, there are useful subclasses of SDD's that are sufficient to guarantee that an order of evaluation exists.

Example 23.3 *An SDD based on a grammar suitable for top-down parsing.*

The SDD in Table 23.3 computes terms like $4 * 6$ and $4 * 6 * 7$. The top-down parse of input $4 * 6$ begins with the production $T \to FT'$. Here, F generates the digit 4, but the operator $*$ is generated by T'. Thus, the left operand 4 appears in a different subtree of the parse tree from $*$. An inherited ($.inh$) attribute will therefore be used to pass the operand to the operator.

Table 23.2: An SDD based on a grammar suitable for top-down parsing.

S.No.	Production	Semantic Rules
1.	$T \to FT'$	$T'.inh = F.val$
		$T.val = T'.syn$
2.	$T' \to *FT'_1$	$T'_1.inh = T'_1.inh \times F.val$
		$T'.syn = T'_1.syn$
3.	$T' \to \varepsilon$	$T'.syn = T'.inh$
4.	$F \to$ **digit**	$F.val =$ **digit**$.lexval$

The grammar in this example is an excerpt from a non-left-recursive version of the familiar expression grammar; we used such a grammar as a running example to illustrate top-down parsing earlier.

Each of the nonterminals T and F has a synthesized attribute val; the terminal **digit** has a synthesized attribute $lexval$. The nonterminal T' has two attributes: an inherited attribute inh and a synthesized attribute syn.

The semantic rules are based on the idea that the left operand of the operator $*$ is inherited. More precisely, the head T' of production $T' \to * \ F \ T_1'$ inherits the left operand of $*$ in the production body. Given a term $x * y * z$, the root of the subtree for $*y * z$ inherits x. Then, the root of the subtree for $*z$ inherits the value of $x * y$, and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

To see how the semantic rules are used, consider the annotated parse tree for $4*6$ in Fig. 23.6. The leftmost leaf in the parse tree, labeled **digit**, has attribute value $lexval = 4$, where the 4 is supplied by the lexical analyzer. Its parent is for production 4, $F \to$ **digit**. The only semantic rule associated with this production defines $F.val =$ **digit**$.lexval$, which equals 4.

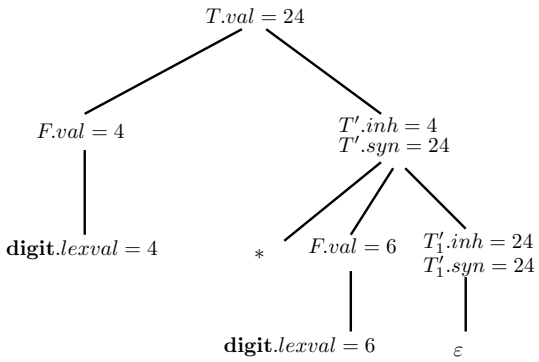

Figure 23.3: Annotated parse tree for $4 * 6$.

At the second child of the root, the inherited attribute $T'.inh$ is defined by the semantic rule $T'.inh = F.val$ associated with production 1. Thus, the left operand, 4, for the $*$ operator is passed from left to right across the children of the root.

The production at the node for T' is $T' \to * \ F \ T_1'$. (We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for T'.) The inherited attribute $T_1'.inh$ is defined by the semantic rule $T_1'.inh = T'.inh \times F.val$ associated with production 2.

With $T'.inh = 4$ and $F.val = 6$, we get $T_1'.inh = 24$. At the lower node for T_1', the production is $T' \to \varepsilon$. The semantic rule $T'.syn = T'.inh$ defines $T_1'.syn = 24$. The syn attributes at the nodes for T' pass the value 24 as synthesized attributes up the tree to the node for T, where $T.val = 24$. □

23.3 Evaluation Orders for SDD's

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an *annotated parse tree* shows the values of attributes, a *dependency graph* helps us determine how those values can be computed. In the following, in addition to dependency graph, we define two important classes of SDD's: the "S-attributed" and the more general "L-attributed" SDDs. The translations specified

by these two classes fit well with the parsing methods we have studied, and most translations encountered in practice can be written as per the requirements of at least one of these classes.

23.3.1 Dependency Graphs

A dependency graph indicate the flow of information among the instances of attributes in a given parse tree. An edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules. Following is more detailed introduction:

- For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.

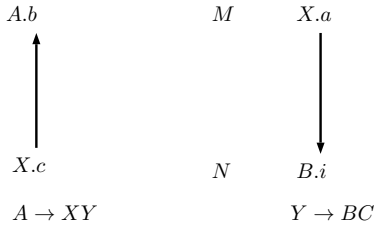

Figure 23.4: Sample Dependency graphs.

- Suppose that a semantic rule associated with a production p defines the value of synthesized attribute $A.b$ in terms of the value of $X.c$ (the rule may define $A.b$ in terms of other attributes in addition to $X.c$) . Then, the dependency graph has an edge from $X.c$ to $A.b$. More precisely, at every node N labeled as A where production p is applied, create an edge to attribute b at N, from the attribute c at the child of N corresponding to this instance of the symbol X in the body of the production[1].
- Suppose that a semantic rule associated with a production p defines the value of inherited attribute $B.i$ in terms of the value of $X.a$. Then, the dependency graph has an edge from $X.a$ to $B.i$. For each node N labeled B that corresponds to an occurrence of this B in the body of production p, create an edge to attribute i at N from the attribute a at the node M that corresponds to this occurrence of X. Note that M could be either the parent or a sibling of N.

Example 23.4 *Synthesize $E.val$ from $E_1.val$ and $E_2.val$.*

Consider the following production and rule:

$$\begin{aligned} &\textit{Production Rule}: \\ &\quad E \to E_1 + T \end{aligned}$$

[1]Since a node N can have several children labeled X, we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

$$\begin{array}{c} \textit{Semantic Rule}: \\ E.val = E_1.val + T.val \end{array}$$

At every node N labeled E, with children corresponding to the body of this production, the synthesized attribute val at N is computed using the values of val at the two children, labeled E and T. Thus, a portion of the dependency graph for every parse tree in which this production is used looks like in Fig. 23.5. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

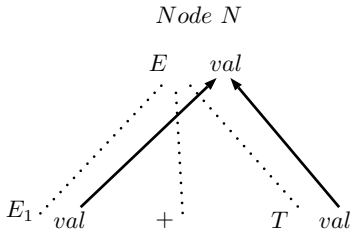

Figure 23.5: $E.val$ is synthesized from $E_1.val$ and $E_2.val$.

□

Example 23.5 *Construct a Dependency graph for the annotated parse tree of Fig. 23.6, whose SDD is given in Table 23.3.*

Table 23.3: An SDD based on a grammar suitable for top-down parsing.

S.No.	Production	Semantic Rules
1.	$T \to FT'$	$T'.inh = F.val$
		$T.val = T'.syn$
2.	$T' \to *FT'_1$	$T'_1.inh = T'_1.inh \times F.val$
		$T'.syn = T'_1.syn$
3.	$T' \to \varepsilon$	$T'.syn = T'.inh$
4.	$F \to$ **digit**	$F.val =$ **digit**$.lexval$

The nodes of the dependency graph, represented by the numbers 1 through 9, in Figure 23.7 correspond to the attributes in the annotated parse tree in Fig. 23.6.

Nodes 1 and 2 represent the attribute lexval associated with the two leaves labeled **digit**. Nodes 3 and 4 represent the attribute val (i.e., 4 and 6) associated with the two nodes labeled F. The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that defines $F.val$ in terms of **digit**$.lexua1$. In fact, $F.val$ equals **digit**$.lexval$, but the edge represents dependence, not equality.

Nodes 5 and 6 represent the inherited attribute $T'.inh$ associated with each of the occurrences of nonterminal T' ($T' \to *FT'_1$ and $T' \to \varepsilon$). The edge from 3 to 5 is due to the rule $T'.inh = F.val$, which defines $T'.inh$ at the right child of the root from $F.va1$ at the left child. We see edges to node 6 from node 5 for $T'.inh$ and from node 4 for $F.val$, because these values are multiplied to evaluate the attribute inh at node 6.

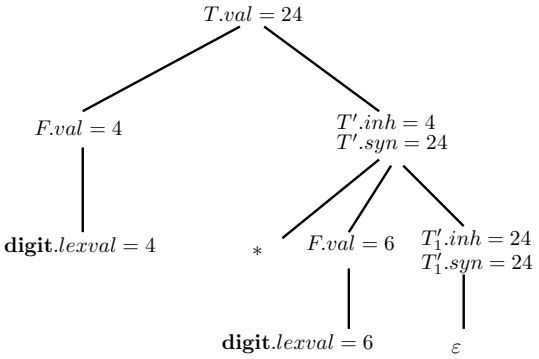

Figure 23.6: Annotated parse tree for $4 * 6$.

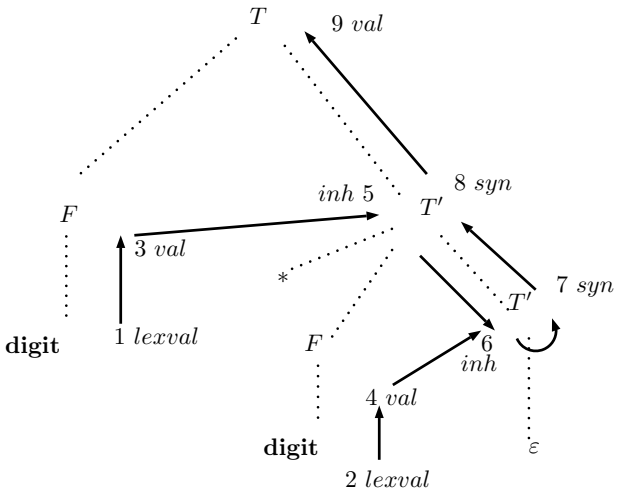

Figure 23.7: Dependency graph for the annotated parse tree of Fig. 23.6.

Nodes 7 and 8 represent the synthesized attribute *syn* associated with the occurrences of T'. The edge to node 7 from 6 is due to the semantic rule $T'.syn = T'.inh$ associated with production $T' \to \varepsilon$. The edge to node 8 from 7 is due to a semantic rule associated with production 2.

Finally, node 9 represents the attribute $T.val$. The edge from 8 to 9 is due to the semantic rule, $T.val = T'.syn$, associated with production 1 in Table 23.3. □

23.3.2 Ordering the evaluation of Attributes

The dependency graph specify the possible orders in which we can evaluate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes N_1, N_2, ..., N_k such that if there is an edge of the dependency graph from N_i to N_j; where $i < j$. Such an ordering transforms a directed graph into a linear order, called *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts possible, that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort. To see why, since there are no cycles, we can surely find a node with no edge entering. For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle. Make this node the first in the topological order, remove it from the dependency graph, and repeat the process on the remaining nodes.

Example 23.6 *Topological Sort.*

The dependency graph of Fig. 23.7 has no cycles. One topological sort is the order in which the nodes have already been numbered: 1,2,. . . ,9. Notice that every edge of the graph goes from a node to a higher-numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as 1,3,5,2,4,6,7,8,9. □

23.3.3 S-Attributed Definitions

As mentioned earlier, given an SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles. In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit dependency graphs with cycles. Moreover, the two classes introduced in this section can be implemented efficiently in connection with top-down or bottom-up parsing.

The first class is defined as follows:

Definition 23.7 *An SDD is S-attributed if every attribute is synthesized.*

Example 23.8 *The SDD of Table. 23.4 is an example of an S-attributed definition. Each attribute, L.val, E.va*1*, T.val, and F.val is synthesized.* □

Table 23.4: Syntax-directed definition of a simple desk calculator (S-attributed).

	Production	Semantic Rules
1.	$L \to E$ **n**	$L.val = E.val$
2.	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3.	$E \to T$	$E.val = T.val$
4.	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5.	$T \to F$	$T.val = F.val$
6.	$F \to (E)$	$F.val = E.val$
7.	$F \to$ **digit**	$F.val =$ **digit**$.lexval$

When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node N when the traversal leaves N for the last time. That is, we apply the function postorder, defined below, to the root of the parse tree:

Algorithm 1 Postorder(N)

```
1: Postorder(N){
2: for (each child C of N, from left to right) postorder(C); do
3:    evaluate the attributes associated with node N;
4: end for
5: }
```

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal. Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head.

23.3.4 L-Attributed Definitions

The second class of SDD's is called L-attributed definitions. For these definitions, edges of dependency graph, corresponding to the attributes associated with production body, can go from left to right but not from right to left in production symbols. More precisely, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1X_2...X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:
   (a) Inherited attributes associated with the head A.
   (b) Either inherited or synthesized attributes associated with the occurrences of symbols X_1, X_2, ... , X_{i-1} occurring to the left of X_i.
   (c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

Theorem 23.9 *Show that the SDD given in Table 23.3 is L-attributed.*

Proof: The SDD in Table 23.3 is L-attributed. To see why, consider the semantic rules for inherited attributes: 1 and 2.

The first of these rules defines the inherited attribute $T'.inh$ using only $F.val$, and F appears to the left of T' in the production body, as required. The second rule defines $T'_1.inh$ using the inherited attribute $T'.inh$ associated with the head, and $F.va1$, where F appears to the left of T'_1 in the production body.

In each of these cases, the rules use information "from above or from the left ," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed. ■

Example 23.10 *Any SDD containing the following production and rules cannot be L-attributed:*

$$\begin{aligned} &\textit{Production}: \\ &A \to BC. \end{aligned}$$

$$\begin{aligned} &\textit{Semantic Rules}: \\ &A.s = B.b; \\ &B.i = f(C.c, A.s) \end{aligned}$$

The first rule, $A.s = B.b$, is a legitimate rule in either an S-attributed or L-attributed SDD. It defines a synthesized attribute $A.s$ in terms of an attribute b at child B (that is, a symbol within the production body).

The second rule defines an inherited attribute $B.i$, so the entire SDD cannot be S-attributed. Further, although the rule is legal, the SDD cannot be L-attributed, because the attribute $C.c$ is used to help define $B.i$, and C is to the right of B in the production body. While attributes at siblings in a parse tree may be used in L-attributed SDD's, they must be to the left of the symbol whose attribute is being defined. □

23.4 Exercises

1. How a graph can be used for computing attributes? Explain.
2. Construct parse-trees using S-attributed grammar for the following expressions:
   (a) $(3 + 4) * (5 + 6)$
   (b) $3 + 4 * (5 + 6)$
   (c) $3 + 4 * 5 + 6$
3. What are all the topological sorts for the dependency graph of Fig. 23.7?
4. Explain the process for topological sorting of a graph. Take an example, and execute it.
5. Why the graphs used in attribute grammars are called dependency graphs.
6. Prove that SDD in Table 23.3 is L-attributed grammar.
7. Show that, the SDD with following production and rule is not L-attributed.

$$\begin{aligned} A &\to BC \\ A.s &= B.b \\ B.i &= f(C.c, A.s) \end{aligned}$$

8. What is the only fundamental difference between S.attributed and L-attributed definitions?

9. In Figure 23.7, explain how the attributes at all the nodes are computed?

10. Does the multiplication in Fig. 23.7 take place at node 5 or 6? Justify.

11. What is search in a topological order in a graph? Explain with example.

12. Describe the method to list the nodes in topological order in a dependency graph, if it exists.

13. Write an algorithm to list the nodes in a dependency graph in topological order. If no such order exists, print "No".

14. List all the possible topological orders of nodes of dependency graph of Figure 23.7. Also, justify these orders.

15. Describe the method you will use to find whether a topological of nodes exists in a dependency graph.

16. What are the differences between S-attributed and L-attributed SDD?

17. Using the SDD of Table 23.1, construct the annotated parse trees for the following expressions:

    (a) $2 + 3 + 4$

    (b) $2 + 3 * 4$

    (c) $(2 + 3) * (5 + 6)\mathbf{n}$.

    (d) $1 * 2 * (3 + 4)\mathbf{n}$.

18. Extend the SDD of Table 23.3 to handle expressions as in Table 23.1.

19. Which source of attributes are common in synthesized and inherited attributes?

20. Shot answer questions.

    (a) Can an inherited attribute be computed in terms of attributes of its children?

    (b) Can a synthesized attribute be computed in terms of attributes of its children?

    (c) How an inherited attribute is computed (its process)? What are its dependencies?

    (d) Why an inherited attribute is complex to compute?

    (e) Can a terminal node have inherited attributes?

    (f) Can a terminal node have synthesized attributes?

21. Explain, why the expression $a * b * c$ cannot be computed using synthesized attributes only, i.e., using $T.val = T_1.val \times F.val$, $T.val = F.val$, $F.val = digit.lexval$?

Ans. Otherwise it would be left-recursive.

22. What are the circular rules of attributes? Give examples. Also, explain the challenges of circular dependencies.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.