CHAPTER FOUR
4. SYNTAX-DIRECTED TRANSLATION

4.1. Introduction

This chapter develops the translation of languages guided by context-free grammars. We associate information with a programming language construct by attaching attributes to the grammar symbols representing the construct. Values for attributes are computed by "semantic rules" associated with the grammar productions.

There are two notations for associating semantic rules with productions, syntax directed definitions and translation schemes. Syntax-directed definitions are high-level specifications for translations. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

Production	**Semantic Rule**
$E \rightarrow E_1+T$	$E.code=E_1.code\|\|T.code\|\|'+'$

This production has two non-terminals, *E* and *T*; the subscript in E_1 distinguishes the occurrence of *E* in the production body from the occurrence of *E* as the head. Both *E* and *T* have a string-valued attribute *code*. The semantic rule specifies that the string *E.code* is formed by concatenating $E_1.code$, *T.code*, and the character '+'. While the rule makes it explicit that the translation of *E* is built up from the translations of E_1, *T*, and '+', it may be inefficient to implement the translation directly by manipulating strings.

Translation schemes indicate the order in which semantic rules are to be evaluated, so they allow some implementation details to be shown. A syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$E \rightarrow E_1+T$ {print '+'}

By convention, semantic actions are enclosed within curly braces. (If curly braces occur as grammar symbols, we enclose them within single quotes, as in '{' and '}'.) The position of a semantic action in a production body determines the order in which the action is executed. In general, semantic actions may occur at any position in a production body.

Between the two notations, syntax-directed definitions can be more readable, and hence more useful for specifications. However, translation schemes can be more efficient, and hence more useful for implementations.

The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree. In many cases, translation can be done during parsing, without building an explicit tree. We shall therefore study a class of syntax-directed translations called "L-attributed translations" (L for left-to-right), which encompass virtually all translations that can be performed during parsing. We also study a smaller class, called "S-attributed translations" (S for synthesized), which can be performed easily in connection with a bottom-up parse.

4.2. Syntax Directed Definitions

A syntax directed definition is a generalization of the CFG in which each grammar symbol has an associated set of attributes (synthesized and inherited) and rules. Attributes are associated with grammar symbols and rules 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 labeled *X*. An attribute can represent anything we choose (a string, a number, a type, a memory location, etc.).

4.2.1. Inherited and Synthesized Attributes

We shall deal with two kinds of attributes for non-terminals:

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. The attributes of the parent depend on the attributes of the children.

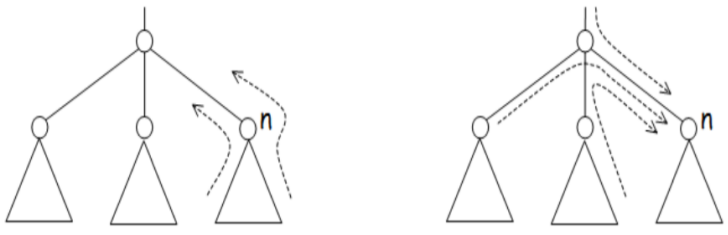
   These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production: S → ABC. If *S* is taking values from its child nodes (*A, B, C*), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S. As in our former example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

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. The attributes of the children depend on the attributes of the parent.

   In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production: S → ABC. *A* can get values from *S, B,* and *C*. *B* can take values from *S, A,* and *C*. Likewise, *C* can take values from *S, A,* and *B*.

The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree; the value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node in the parse tree.

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



(a) Synthesized at node n (b) Inherited at node n

Figure 4.1: Synthesized and inherited attributes

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.

Example 4.1: The SDD in Fig. 4.2 is based on our familiar grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an end marker ***n***. In the SDD, each of the non-terminals 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.

Production	Semantic rules
1) L → E n	L.val = E.val
2) E → E_1 + T	E.val = E_1.val + T.val
3) E → T	E.val = T.val
4) T → T_1 * F	T.val = T_1.val * F.val
5) T → F	T.val = F.val
6) F → (E)	F.val = E.val
7) F → digit	F.val = digit.lexval

Figure 4.2: Syntax-directed definition of a simple desk calculator

The rule for production 1, L → E **n**, sets *L.val* to *E.val*, which we shall see is the numerical value of the entire expression.

Production 2, E → 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 3, E → 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 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 Fig. 4.2 has this 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.

4.2.2. Evaluating an SDD at the Nodes of a Parse Tree

To visualize the translation specified by an SDD, it helps to work with parse trees, 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) is called an **annotated parse tree or decorated 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 4.1, 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 postorder traversal of the parse tree.

Example 4.2: Figure 4.3 shows an annotated parse tree for the input string 3 * 5 + 4 n, constructed using the grammar and rules of Fig. 4.2.

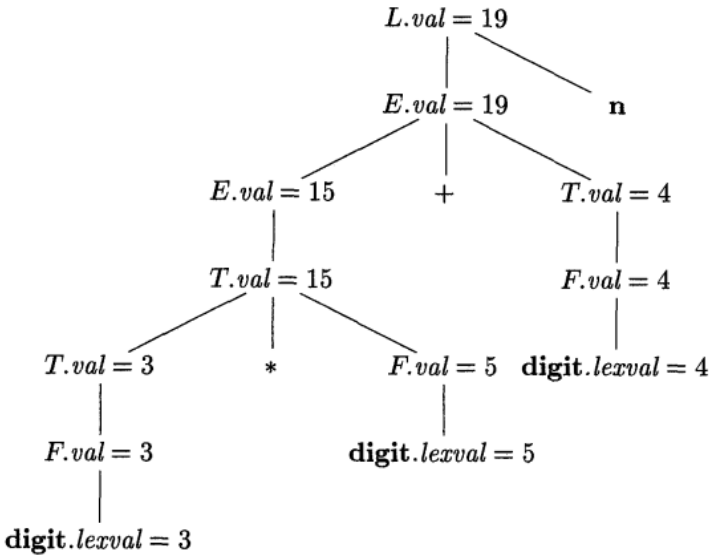


Figure 4.3: Annotated parse tree for 3 * 5 + 4 **n**

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.

Example 4.3: The SDD in Fig. 4.4 computes terms like 3 * 5 and 3 * 5 * 7. The top-down parse of input 3 * 5 begins with the production $T \rightarrow F\ T'$. Here, F generates the **digit** 3, but the operator * is generated by T'. Thus, the left operand 3 appears in a different subtree of the parse tree from *. An inherited attribute will therefore be used to pass the operand to the operator. The grammar in this example is an excerpt from a non-left-recursive version of the familiar expression grammar.

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Figure 4.4: An SDD based on a grammar suitable for top-down parsing

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 the production $T' \rightarrow * F T_1'$ 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 3 * 5 in Fig. 4.5. The leftmost leaf in the parse tree, labeled ***digit***, has attribute value *lexval* = 3, where the 3 is supplied by the lexical analyzer. Its parent is for production 4, $F \rightarrow digit$. The only semantic rule associated with this production defines *F. val = digit. lexval*, which equals 3.

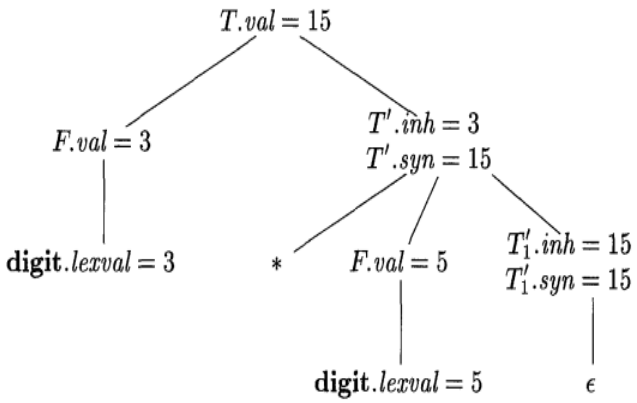


Figure 4.5: Annotated parse tree for 3 * 5

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, 3, for the * operator is passed from left to right across the children of the root.

The production at the node for *T'* is $T' \rightarrow * FT_1'$. (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 = 3* and *F.val = 5*, we get $T_1'.inh = 15$. At the lower node for T_1', the production is $T' \rightarrow \varepsilon$. The semantic rule *T'.syn = T'.inh* defines $T_1'.syn = 15$. The *syn* attributes at the nodes for *T'* pass the value 15 up the tree to the node for *T*, where *T.val = 15*.

4.2.3. Dependency graph

If an attribute *b* at a node in a parse tree depends on an attribute *c*, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.

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.

A dependency graph depicts the flow of information among the attribute instances in a particular 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. In more detail:

- ✓ 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*.
- ✓ 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 *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.
- ✓ Suppose that a semantic rule associated with a production *p* defines the value of inherited attribute *B.c* in terms of the value of *X.a*. Then, the dependency graph has an edge from *X.a* to *B.c*. 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 *c* 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*.

For example, consider the following production and semantic rule:

Production	**Semantic Rule**
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + T.val$

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 the following figure. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

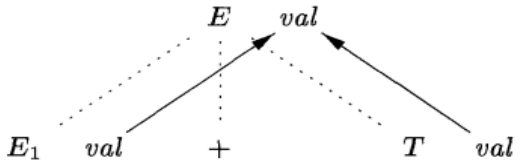


Figure 4.6: *E.val* is synthesized from $E_1.val$ and $E_2.val$

An example of a complete dependency graph appears in the following figure (Fig: 4.7). The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 4.5.

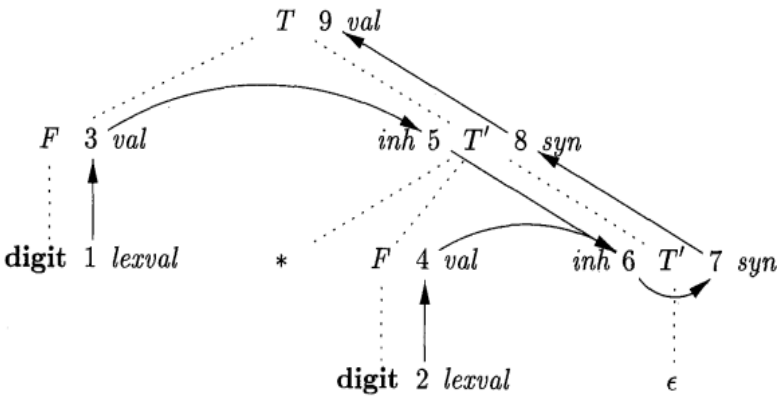


Figure 4.7: Dependency graph for the annotated parse tree of Fig. 4.5

Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled **digit**. Nodes 3 and 4 represent the attribute *val* 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****.lexva1*. 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'*. The edge to 5 from 3 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 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.

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 3 in Fig. 4.4. 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 to 9 from 8 is due to the semantic rule, *T.val = T'.syn*, associated with production 1.

A dependency graph for the input 5+3*4 constructed using the grammar and rules of Fig. 4.2 is shown below.

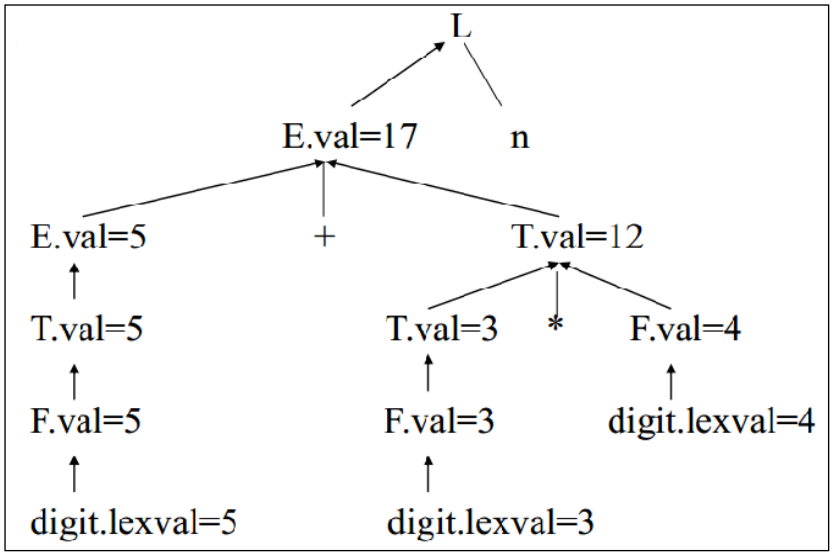
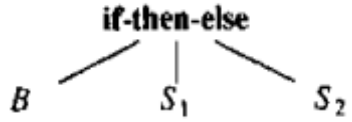


Figure 4.8: Dependency graph for 5+3*4

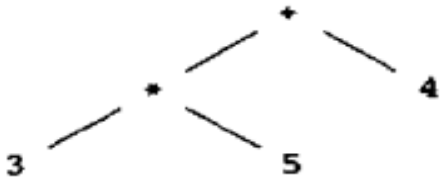
4.3. Construction of Syntax trees

❖ Syntax Trees

An (abstract) syntax tree is a condensed form of parse tree useful for representing language constructs. The production S → **if** *B* **then** S_1 **else** S_2 might appear in a syntax tree as follows.



In a syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the parse tree. Another simplification found in syntax trees is that chains of single productions may be collapsed; the parse tree of Fig. 4.3 becomes the following syntax tree.



Syntax-directed translation can be bad on syntax trees as well as parse trees. The approach is the same in each case; we attach attributes to the nodes as in a parse tree.

❖ Constructing Syntax Trees for Expressions

The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form. We construct subtrees for the subexpressions by creating a node for each operator and operand. The children of an operator node are the roots of the nodes representing the subexpressions constituting the operands of that operator.

Each node in a syntax tree can be implemented as a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes for the operands. The operator is often called the *label* of the node. When used for translation, the nodes in a syntax tree may have additional fields to hold the values (or pointers to values) of attributes attached to the node. In this section, we use the following functions to create the nodes of syntax trees for expressions with binary operators. Each function returns a pointer to a newly created node.

1. *mknode*(*op, left, right*) creates an operator node with label *op* and two fields containing pointers to *left* and *right*.
2. *mkleaf* (*id, entry*) creates an identifier node with label **id** and a field containing *entry*, a pointer to the symbol-table entry for the identifier.
3. *mkleaf* (*num, val*) creates a number node with label **num** and a field containing *val*, the value of the number.

Example: The following sequence of functions calls creates the syntax tree for the expression **a – 4 + c** in Fig. 4.9. In this sequence, p_1, p_2. . . . , p_s are pointers to nodes, and *entry-a* and *entry-c* are pointers to the symbol-table entries for identifiers *a* and *c*, respectively.

1) p_1 = *mkleaf*(**id**, entry-a);
2) p_2 = *mkleaf* (**num**, 4);
3) p_3 = *mknode*('-', p_1, p_2);
4) p_4 = *mkleaf*(**id**, entry-c);
5) p_5 = *mknode*('+', p_3, p_4);

The tree is constructed bottom up. The function calls *mkleaf*(**id**, *entry-a*) and *mkleaf*(**num**, 4) construct the leaves for *a* and 4; the pointers to these nodes are saved using p_1, and p_2. The call *mknode*('-', p_1, p_2) then constructs the interior node with the leaves for *a* and 4 as children. After two more step, p_5 is left pointing to the root.

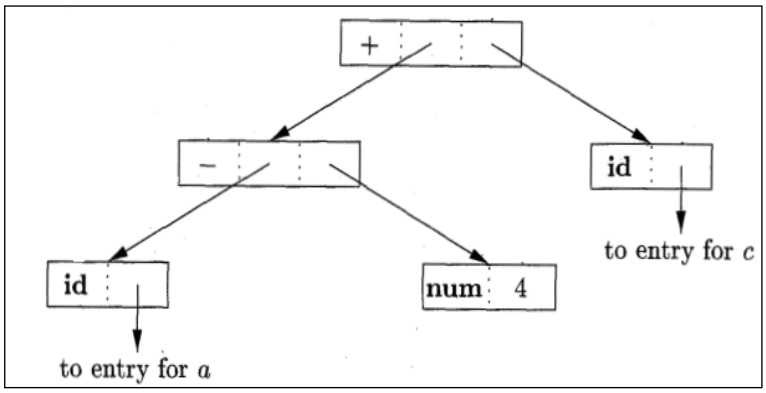


Figure 4.9: Syntax tree for a – 4 + c

❖ **A Syntax-Directed definition for Constructing Syntax Trees**

The following table contains an S-attributed definition for constructing a syntax tree for an expression containing the operators + and - . It uses the underlying productions of the grammar to schedule the calls of the functions *mknode* and *mkleaf* to construct the tree. The synthesized attribute *node* for *E* and *T* keeps track of the pointers returned by the function calls.

Production	Semantic rule
1) E → E_1 + T	*E.node = mknode('+', E_1.node, T.node)*
2) E → E_1 - T	*E.node = mknode('-', E_1.node, T.node)*
3) E → T	*E.node = T.node*
4) T → (E)	*T.node = E.node*
5) T → **id**	*T.node = mkleaf (**id**, **id**.entry)*
6) T → **num**	*T.node = mkleaf (**num**, **num**.val)*

Table 4.1: Syntax-directed definition for constructing a syntax tree for an expression.

Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with '+' for *op* and two children, E_1*.node* and *T.node*, for the subexpressions. The second production has a similar rule. For production 3, $E \rightarrow T$, no node is created, since *E.node* is the same as *T.node*. Similarly, no node is created for production 4, $T \rightarrow (E)$. The value of *T.node* is the same as *E.node*, since parentheses are used only for grouping; they influence the structure of the parse tree and the syntax tree, but once their job is done, there is no further need to retain them in the syntax tree. The last two T-productions have a single terminal on the right. We use the constructor *mkleaf* to create a suitable node, which becomes the value of *T.node*.

The following figure (Figure 4.10) shows the construction of a syntax tree for the input a – 4 + c. The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of *E.node* and *T.node*; each line points to the appropriate syntax tree node.

At the bottom we see leaves for *a*, *4* and *c*, constructed by *mkleaf*. We suppose that the lexical value *id.entry* points into the symbol table, and the lexical value *num.val* is the numerical value of a constant. These leaves, or pointers to them, become the value of *T.node* at the three parse-tree nodes labeled *T*, according to rules 5 and 6. Note that by rule 3, the pointer to the leaf for *a* is also the value of *E.node* for the leftmost *E* in the parse tree.

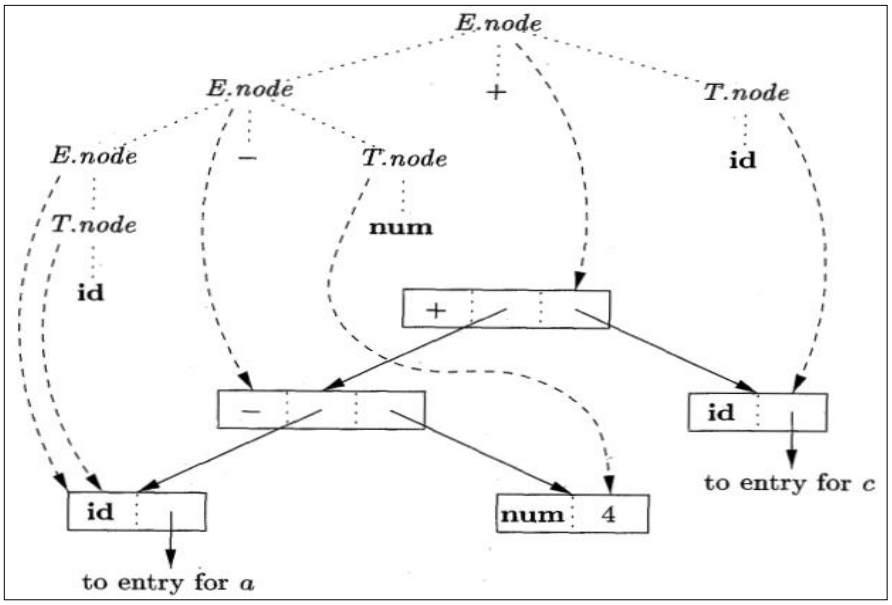


Figure 4.10: Syntax tree for a - 4 + c

Rule 2 causes us to create a node with op equal to the minus sign and pointers to the first two leaves. Then, rule 1 produces the root node of the syntax tree by combining the node for - with the third leaf.

4.4. Bottom-Up Evaluation of S-attributed definitions

Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed. The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack. Whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.

An SDD is S-attributed if every attribute is synthesized. 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.

```
Postorder (N)
  {
    for (each child C of N, from the left)
          Postorder(C);
    evaluate the attributes associated with node N;
  }
```

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.

- Preorder and postorder traversals are two important special cases of depth-first traversals in which we visit the children of each node from left to right. Often, we traverse a tree to perform some particular action at each node. If the action is done when we first visit a node, then we may refer to the traversal as a preorder traversal. Similarly, if the action is done just before we leave a node for the last time, then we say it is a postorder traversal of the tree.

4.5. L-attributed definitions

When translation takes place during parsing, the order of evaluation of attributes is linked to the order in which nodes of a parse tree are "created" by the parsing method. A natural order that characterizes many top-down and bottom-up translation methods is the one obtained by applying the procedure *dfvisit* in Fig. 4.11 to the root of a parse tree. We call this evaluation order the *depth-first* order. Even if the parse tree is not actually constructed, it is useful to study translation during parsing by considering depth-first evaluation of attributes at the nodes of a parse tree.

```
procedure dfvisit (n : node);
begin
    for each child m of n, from left to right do begin
        evaluate inherited attributes of m;
        dfvisit (m)
    end;
evaluate synthesized attributes of n
```

Fig. 4.11. Depth-first evaluation order for attributes in a parse tree.

We now introduce a class of syntax-directed definitions, called L-attributed definitions, whose attributes can always be evaluated in depth-first order. (The L is for "left", because attribute information appears to flow from left to right).

L-attributed definition is the second class of SDD. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). 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 $A \rightarrow X_1X_2 \ldots X_n$, 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 $X_1, X_2, ..., X_{i-1}$ located 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.

For example, the SDD in Fig. 4.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

Production	**Semantic Rule**
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \; X \; F.val$

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.val*, 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.

Any SDD containing the following production and rules cannot be L-attributed (**s** is synthesized and **i** is for inherited):

Production	**Semantic Rule**
$A \rightarrow BC$	$A.s = B.b;$
	$B.i = f(C.c, A.s);$

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 at a child (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.

The syntax-directed definition in the following table (Table: 4.2) is not L-attributed because the inherited attribute *Q.i* of the grammar symbol *Q* depends on the attribute *R.s* of the grammar symbol to its right.

Production	**Semantic Rule**
$A \rightarrow LM$	$L.i = l(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
A → QR	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

Table 4.2: A non- L-attributed syntax directed definition.