7CS5A COMPILER CONSTRUCTION (Synthesized and inherited attributes)

Lecture 4: Oct. 04, 2018

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

4.1 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 labeled 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 4.1 *Syntax-directed definition of a simple desk calculator.*

Solution. The SDD in Table 4.1 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 4.1: Syntax-directed definition (SDD) 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**$.lexval$

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 4.1 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 4.2 *Construction of Annotated parse tree for* $3 * 5 + 4$ **n**.

Solution. Figure 4.1 shows an annotated parse tree for the input string $3 * 5 + 4$ **n**, constructed using the grammar and rules of Table 4.1. 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.

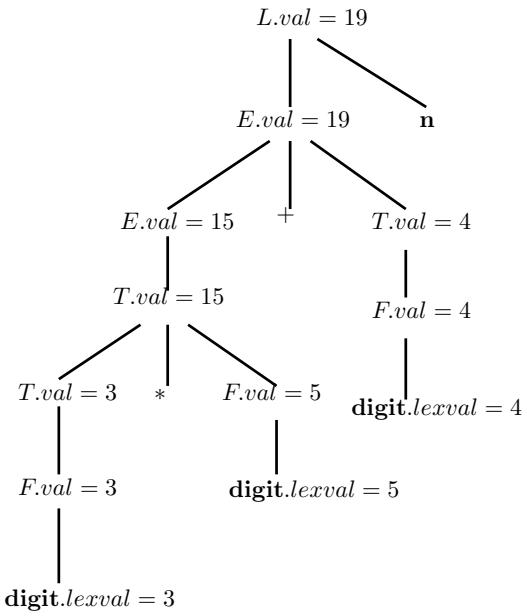

Figure 4.1: Annotated parse tree for $3 * 5 + 4$ **n**.

4.2 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 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; 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 nonterminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, respectively, along with the production and rules as follows:

$$\begin{array}{l} \textit{Production}: \\ \quad A \to B \end{array}$$

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

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 4.2.

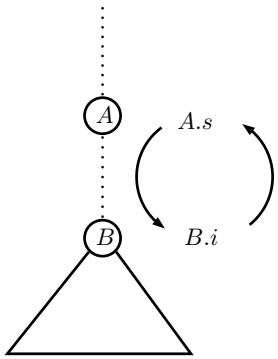

Figure 4.2: The circular dependency of $A.s$ and $B.i$ on one another.

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

Solution. The SDD in Table 4.2 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 4.2: An SDD based on a grammar suitable for top-down parsing.

Rule	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. 4.3. 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.

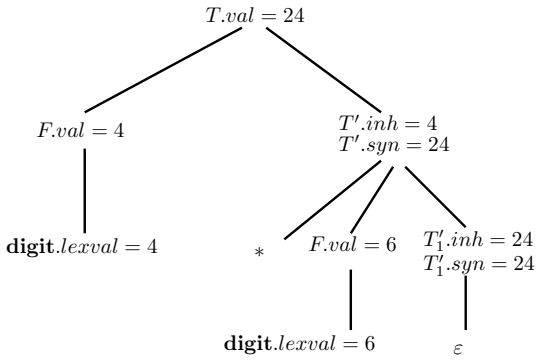

Figure 4.3: Annotated parse tree for $4 * 6$.

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 $T.val = 24$. □

4.3 Exercises

1. Using the SDD of Table 4.1, 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) $1*2*(3+4)\mathbf{n}$.

2. Extend the SDD of Table 4.2 to handle expressions as in Table 4.1.

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

4. 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?

5. 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.

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