بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان – دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۱۲)

کامپایلر

حسین فلسفین

مثالی از خصیصه‌های موروثی

Example 5.3: The SDD in Fig. 5.4 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; we used such a grammar as a running example to illustrate top-down parsing in Section 4.4.

	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)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Figure 5.4: 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 $T' \to * \; F \; T_1'$ 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 $3 * 5$ in Fig. 5.5. 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, $F \rightarrow \textbf{digit}$. The only semantic rule associated with this production defines $F.val = \textbf{digit}.lexval$, which equals 3.

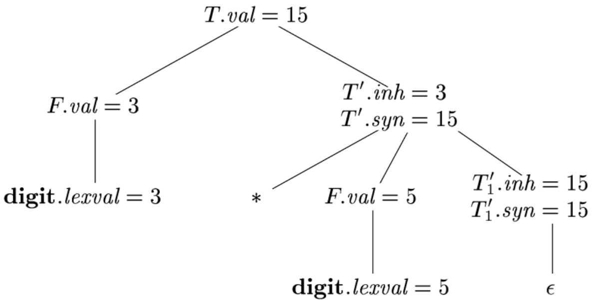


Figure 5.5: Annotated parse tree for $3 * 5$

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 $T' \rightarrow * F T'_1$. (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.

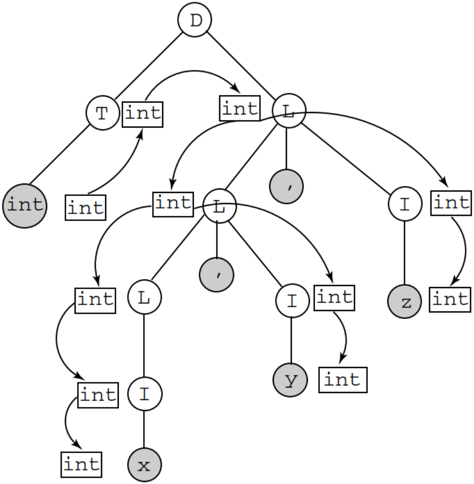
With $T'.inh = 3$ and $F.val = 5$, we get $T'_1.inh = 15$. At the lower node for T'_1, the production is $T' \rightarrow \epsilon$. The semantic rule $T'.syn = T'.inh$ defines $T'_1.syn = 15$. The *syn* attributes at the nodes for T' pass the value 15 up the tree to the node for T, where $T.val = 15$. $\square$

مثال دیگری از خصیصه‌های موروثی

Consider the following grammar for C language style variable declaration:

```
D : T L
L : L , I | I
T : int | float
I : x | y | z
```

Now consider a declaration `int x, y, z`*. The synthesized and inherited attribute derivation is shown in the following figure:*



Example of inherited attributes. One up-going arrow denotes a synthesized attribute, and all remaining arrows indicate inherited attributes

The corresponding attribute grammar is given below:

D	:	T L	L.in = T.type	(inherited)
T	:	int	T.type = int.int	(synthesized)
T	:	float	T.type = float.float	(synthesized)
L0	:	L1 , I	L1.in = L0.in	(inherited)
			I.in = L0.in	(inherited)
L	:	I	I.in = L.in	(inherited)
I	:	id	id.type = I.in	(inherited)

5.2 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. 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. (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.)*

☞ ***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.***

Example 5.4: Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \to E_1 + T$	$E.val = E_1.val + T.val$

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 Fig. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid. □

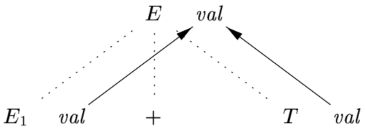


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $T.val$

Example 5.5: An example of a complete dependency graph appears in Fig. 5.7. The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 5.5.

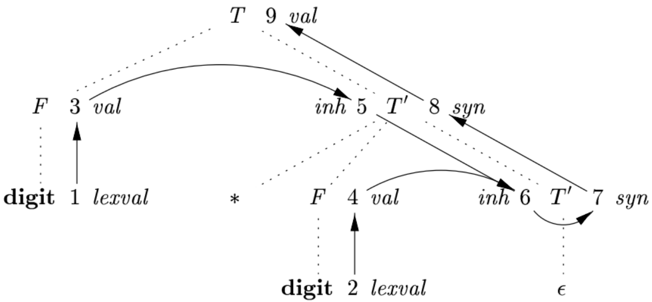


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

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**.*lexval*. 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 $F.val$ 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. 5.4. 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. □

Ordering the Evaluation of Attributes

The dependency graph characterizes 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 $N_1, N_2, \ldots, N_k$ such that if there is an edge of the dependency graph from N_i to N_j, then $i < j$. Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.

نباید در گراف وابستگی دوری وجود داشته باشد

If there is any cycle in the graph, then there are no topological sorts; 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 5.6:** The dependency graph of Fig. 5.7 has no cycles. One topological sort is the order in which the nodes have already been numbered:* $1, 2, \ldots, 9$. *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* $1, 3, 5, 2, 4, 6, 7, 8, 9$.

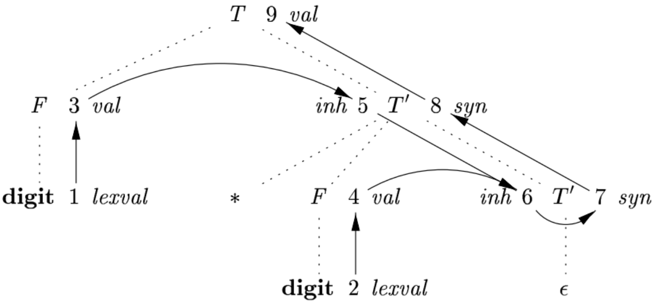


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

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

Example 5.7: The SDD of Fig. 5.1 is an example of an S-attributed definition. Each attribute, $L.val$, $E.val$, $T.val$, and $F.val$ 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 (see also the box "Preorder and Postorder Traversals" in Section 2.3.4):

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

ممکن است لزومی به ساخت صریح نودهای درخت نباشد

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. This fact will be used in Section 5.4.2 to evaluate synthesized attributes and store them on the stack during LR parsing, without creating the tree nodes explicitly.