بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

تحلیل معنایی و ساخت کد میانی می‌تواند مبتنی بر درخت تجزیه صورت پذیرد

☞ *Informally, syntax concerns the form of a valid program, while semantics concerns its meaning.*

☞ *Both semantic analysis and intermediate code generation can be described in terms of annotation, or decoration of a parse tree or syntax tree. The annotations themselves are known as attributes. (The process of evaluating attributes is called annotation or decoration of the parse tree.)*

☞ *Attribute grammars provide a formal framework for the decoration of a tree. This framework is a useful conceptual tool even in compilers that do not build a parse tree or syntax tree as an explicit data structure.*

☞ *In practice, most compilers require decoration of the parse tree (or the evaluation of attributes that would reside in a parse tree if there were one) to occur in the process of an LL or LR parse.*

جلسهٔ پیش، ***Syntax-Directed Definitions*** را معرفی کردیم. الان به معرفی ***Syntax-Directed Translation Schemes*** می‌پردازیم.

2.3.4 Tree Traversals

Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme. A traversal of a tree starts at the root and visits each node of the tree in some order.

A depth-first traversal starts at the root and recursively visits the children of each node in any order, not necessarily from left to right. It is called "depth-first" because it visits an unvisited child of a node whenever it can, so it visits nodes as far away from the root (as "deep") as quickly as it can.

```
procedure visit(node N) {
        for ( each child C of N, from left to right ) {
                visit(C);
        }
        evaluate semantic rules at node N;
}
```

Figure 2.11: A depth-first traversal of a tree

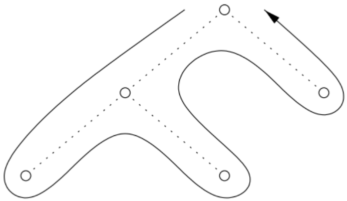


Figure 2.12: Example of a depth-first traversal of a tree

The procedure $visit(N)$ in Fig. 2.11 is a depth first traversal that visits the children of a node in left-to-right order, as shown in Fig. 2.12. In this traversal, we have included the action of evaluating translations at each node, just before we finish with the node (that is, after translations at the children have surely been computed). In general, the actions associated with a traversal can be whatever we choose, or nothing at all.

A syntax-directed definition does not impose any specific order for the evaluation of attributes on a parse tree; any evaluation order that computes an attribute a after all the other attributes that a depends on is acceptable. Synthesized attributes can be evaluated during any bottom-up traversal, that is, a traversal that evaluates attributes at a node after having evaluated attributes at its children. In general, with both synthesized and inherited attributes, the matter of evaluation order is quite complex; see Section 5.2.

Preorder and Postorder Traversals

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. The procedure $visit(N)$ in Fig. 2.11 is an example of a postorder traversal.

Preorder and postorder traversals define corresponding orderings on nodes, based on when the action at a node would be performed. The *preorder* of a (sub)tree rooted at node N consists of N, followed by the preorders of the subtrees of each of its children, if any, from the left. The *postorder* of a (sub)tree rooted at N consists of the postorders of each of the subtrees for the children of N, if any, from the left, followed by N itself.

2.3.5 Translation Schemes

The syntax-directed definition in Fig. 2.10 builds up a translation by attaching strings as attributes to the nodes in the parse tree. We now consider an alternative approach that does not need to manipulate strings; it produces the same translation incrementally, by executing program fragments.

A syntax-directed translation scheme is a notation for specifying a translation by attaching program fragments to productions in a grammar. A translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly specified.

Program fragments embedded within production bodies are called semantic actions. The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body.

When drawing a parse tree for a translation scheme, we indicate an action by constructing an extra child for it, connected by a dashed line to the node that corresponds to the head of the production. The node for a semantic action has no children, so the action is performed when that node is first seen.

A syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$$E \to E_1 + T \ \{\ \mathrm{print}'+'\ \}$$

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 the above, the action occurs at the end, after all the grammar symbols; in general, semantic actions may occur at any position in a production body.

Example 2.12: The parse tree in Fig. 2.14 has print statements at extra leaves, which are attached by dashed lines to interior nodes of the parse tree. The translation scheme appears in Fig. 2.15. The underlying grammar generates expressions consisting of digits separated by plus and minus signs. The actions embedded in the production bodies translate such expressions into postfix notation, provided we perform a left-to-right depth-first traversal of the tree and execute each print statement when we visit its leaf.

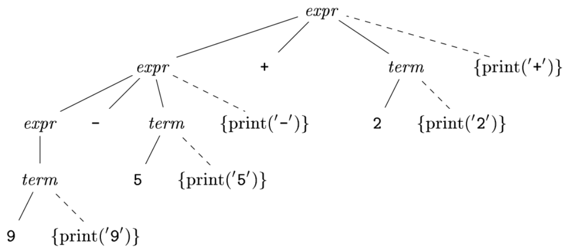


Figure 2.14: Actions translating 9-5+2 into 95-2+

$$\begin{array}{lcll}
expr & \rightarrow & expr_1 \texttt{ + } term & \{\text{print}('\texttt{+}')\} \\
expr & \rightarrow & expr_1 \texttt{ - } term & \{\text{print}('\texttt{-}')\} \\
expr & \rightarrow & term & \\
term & \rightarrow & \texttt{0} & \{\text{print}('\texttt{0}')\} \\
term & \rightarrow & \texttt{1} & \{\text{print}('\texttt{1}')\} \\
 & & \cdots & \\
term & \rightarrow & \texttt{9} & \{\text{print}('\texttt{9}')\}
\end{array}$$

Figure 2.15: Actions for translating into postfix notation

☞ *The root of Fig. 2.14 represents the first production in Fig. 2.15. In a postorder traversal, we first perform all the actions in the left-most subtree of the root, for the left operand, also labeled $expr$ like the root. We then visit the leaf $+$ at which there is no action. We next perform the actions in the subtree for the right operand $term$ and, finally, the semantic action $\{\text{print}('+')\}$ at the extra node.*

☞ *Since the productions for $term$ have only a digit on the right side, that digit is printed by the actions for the productions. No output is necessary for the production $expr \rightarrow term$, and only the operator needs to be printed in the action for each of the first two productions. When executed during a postorder traversal of the parse tree, the actions in Fig. 2.14 print* `95-2+`.

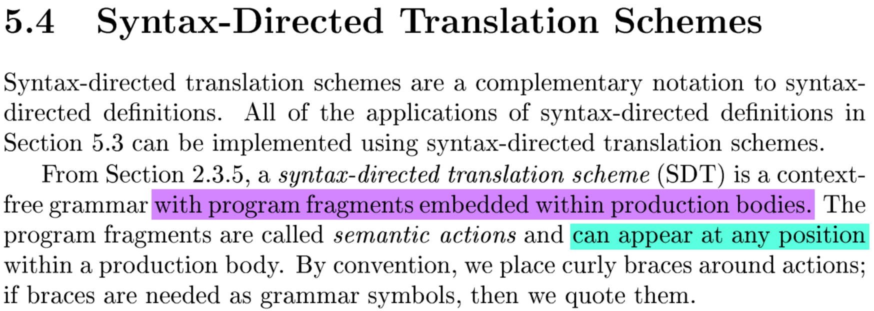
☞ *Note that although the schemes in Fig. 2.10 and Fig. 2.15 produce the same translation, they construct it differently; Fig. 2.10 attaches strings as attributes to the nodes in the parse tree, while the scheme in Fig. 2.15 prints the translation incrementally, through semantic actions.*

☞ *The semantic actions in the parse tree in Fig. 2.14 translate the infix expression* 9−5+2 *into* 95−2+ *by printing each character in* 9−5+2 *exactly once, without using any storage for the translation of subexpressions. When the output is created incrementally in this fashion, the order in which the characters are printed is significant.*

The implementation of a translation scheme must ensure that semantic actions are performed in the order they would appear during a postorder traversal of a parse tree. The implementation need not actually construct a parse tree (often it does not), as long as it ensures that the semantic actions are performed as if we constructed a parse tree and then executed the actions during a postorder traversal.

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.

فصل ۵.۴ از کتاب به این موضوع اختصاص دارد



5.4 Syntax-Directed Translation Schemes

Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. All of the applications of syntax-directed definitions in Section 5.3 can be implemented using syntax-directed translation schemes.

From Section 2.3.5, a *syntax-directed translation scheme* (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.

بحث مفصل‌تر دربارهٔ *SDD*ها، انواع خصیصه‌ها، و محاسبهٔ آنها

Inherited and Synthesized Attributes

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.

☞ *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.*

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.

S-attributed SDDs

An SDD that involves only synthesized attributes is called S-attributed; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the non-terminal at the head of a production from attributes taken from the body of the production. An S-attributed SDD can be implemented naturally in conjunction with an LR parser.

مثالی از یک *S-attributed SDD*

Example 5.1 : The SDD in Fig. 5.1 is based on our 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.

	PRODUCTION	SEMANTIC RULES
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
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)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\mathrm{lexval}$

Figure 5.1: Syntax-directed definition of a simple desk calculator

The rule for production 1, $L \to E\ \mathbf{n}$, sets $L.val$ to $E.val$, which we shall see is the numerical value of the entire expression.

Production 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 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 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. □

☞ *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.*

☞ *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 5.1, 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 in Section 5.2.3.*

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

PRODUCTION	SEMANTIC RULES
$A \to B$	$A.s = B.i;$ $B.i = A.s + 1$

These rules are circular; 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 suggested by Fig. 5.2.

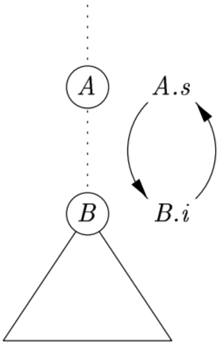


Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

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. (Without going into details, while the problem is decidable, it cannot be solved by a polynomial-time algorithm, even if $P = NP$, since it has exponential time complexity.)

Fortunately, there are useful subclasses of SDD's that are sufficient to guarantee that an order of evaluation exists, as we shall see in Section 5.2.

***Example 5.2:** Figure 5.3 shows an annotated parse tree for the input string $3 * 5 + 4\,\mathbf{n}$, constructed using the grammar and rules of Fig. 5.1. 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.*

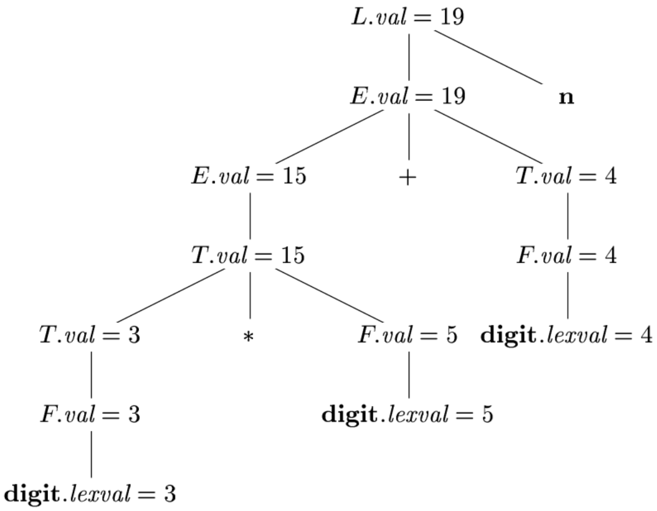


Figure 5.3: Annotated parse tree for $3 * 5 + 4\ \mathbf{n}$