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

Lecture 22: Sept. 26, 2019

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

22.1 Introduction

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar. For example, consider an expression *expr* generated by the production,

$$expr \rightarrow expr_1 + term.$$

The subscript in $expr_1$, is used only to distinguish the instance of *expr* in the production body from the head of the production. We can translate *expr* by exploiting its structure, as in the following pseudo-code:

$$\begin{aligned}&translate\ expr_1;\\&translate\ term;\\&handle\ the\ +;\end{aligned} \tag{22.1}$$

In addition to program fragments, we associate information with a language construct by attaching attributes to the grammar symbol(s) representing the construct. A syntax-directed definition (SDD) specifies the values of attributes by embedding semantic rules in the grammar productions, for example, an infix-to-postfix translator might have a production and semantic rule:

$$Production : E \rightarrow E_1 + T$$

$$\begin{aligned}&Corresponding\ Semantic\ Rule :\\&E.code = E_1.code \parallel T.code \parallel {}'+' \end{aligned} \tag{22.2}$$

This production has two nonterminals, 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[1]. The semantic rule specifies that the string

[1]The word *code* here is for identification, and not for program code. This attribute code supports concatenation.

$E.code$ is formed by concatenating ($\|$ is symbol for concatenation) $E_1.code$, $T.code$, 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, for example, we may need to do type change to perform arithmetic.

A syntax-directed translation (SDT) scheme embeds program fragments called *semantic actions*. The following production shows semantic action as "print'+'".

$$E \to E_1 + T \quad \{print'+'\} \tag{22.3}$$

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 (e.g., in preorder, post order, and inorder). In production (22.3), the action occurs at the end of the production body. In general, a semantic actions may occur at any position in a production body.

Between the two notations discussed above, syntax-directed definitions (e.g., with print '+') can be more readable, and hence more useful for specifications. However, translation schemes, like, infix to postfix format (equation 22.2) can be more efficient, and hence more useful for implementations.

22.2 Syntax Directed Translation

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 other 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 covers 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.

There are two important concepts related to *syntax directed translation*: 1. Attribute, and 2. syntax-directed translation schema.

Attributes. An attribute is any quantity associated with a programming construct, for example, data types in the expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code, etc. In the programming constructs, we use grammar symbols (nonterminals and terminals), and extend the notion of attributes from constructs to the symbols to represent them.

Syntax-directed translation schema. A translation scheme is a notation for attaching program fragments to the productions of a grammar. These program fragments are executed when the production rule is applied during *syntax analysis*. The combined result of all these fragment executions, in the order of the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

22.2.1 Translate to postfix format

We will use Syntax-directed translations to translate infix expressions into postfix notation, to evaluate expressions, and to build syntax trees for programming constructs.

We will discuss translation of expressions into postfix notation. A postfix notation for an expression E can be defined *inductively* as in the following.

Definition 22.1 Conversion from infix to postfix fix notation.

1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form E_1 *op* E_2, where *op* is an arbitrary operator, then postfix notation for E is "$E'_1\ E'_2\ \ op$", where E'_1 and E'_2 are the postfix notations for E_1 and E_2, respectively. The E_1 and E'_2, will not be same unless they are constants or variables. Similar is case with E_2 vs E'_2.
3. If E is a parenthesized expression of the form (E_1), then the postfix notation for E is the same as the postfix notation for E_1. □

Note that there is no parentheses in the translated postfix expression.

Example 22.2 *Translating into posfix notation.*

The postfix notation for $(7-3)+5$ is $7\ 3-5+$. The translation of 7, 3, 5 are constants, by rule (1). Then the translation of $7-3$ is $7\ 3-$ by rule (2). The translation of $(7-3)$ is same by rule (3) above.

Having translated the parenthesized subexpression, we apply the rule (2), to the entire expression, with $(7-3)$ in the role of E_1 and 5 in the role of E_2, to get result $7\ 3-5+$.

As another example, the postfix notation for $7-(3+2)$ is $7\ 3\ 2+-$. That is, $3+2$ is first translated into $3\ 2+$, which become the second argument of minus sign. □

No parentheses are needed in postfix notation, because the position and arity (number of arguments) of the operators permits only one possible decoding of a postfix expression. The "trick" to compute postfix expression is to repeatedly scan the postfix string from the left most, until you find an operator. Then, look to the left for the proper number of operands, and group this operator with its operands. Then evaluate the operator on the operands, and replace them by the result. Then repeat the process, continuing to the right and searching for another operator.

Consider the postfix expression $952+-3*$, we find that it evaluates as follows. However, note that, evaluation of an expression is not the job of a compiler, and it is carried out at run time.

$$\begin{aligned} 9\,5\,2\,+\ \ -\ \ 3\,* &\Rightarrow 9\,7\ \ -\ \ 3\,* \\ &\Rightarrow 2\,3\,* \\ &\Rightarrow 6. \end{aligned}$$

22.2.2 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.

The procedure $visit(N)$ in Algorithm 1 is a depth first traversal, that when called, visits the children of a given node N in left-to-right order. In this traversal, we have included the action of evaluating translations at each node, just before we finish with the node N, i.e., after translations at the children have surely been computed, we compute the translated node N. In general, the actions associated with a traversal can be whatever we choose, or nothing at all.

Algorithm 1 visit(N)

```
1: Procedure visit(node N){
2: for (each child C of parent N, scan from left to right) do
3:     visit(C);
4: end for
5: evaluate semantic rules at node N;
6: }
```

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, after considering the both *synthesized* and *inherited* attributes, the matter of evaluation order is quite complex.

Example 22.3 *Syntax-directed definition for infix to postfix translation.*

The annotated parse tree in Fig. 22.1 is based on the syntax-directed definition in Table 22.1 for translating expressions consisting of digits separated by plus or minus signs into postfix notation. The attribute t indicates the attribute as postfix of *term* and *expr*. Each non-terminal has a string-valued attribute t that represents the postfix notation for the expression generated by that non-terminal in a parse tree. Each parent node computes its attribute by synthesising the attributes of its children nodes. The symbol $\|$ in the semantic rule is the operator for string concatenation.

The postfix form of a digit is the digit itself; e.g., the semantic rule associated with the production $term \rightarrow 7$ defines $term.t$ to be 7 itself whenever this production is used at a node in a parse tree, and similar action is taken for other digits. As another example, when the production $expr \rightarrow term$ is applied, the value of $term.t$ becomes what was the value of $expr.t$ (i.e., $term.t$'s value moves up into value of $expr.t$), in the left most leg of the tree in Fig. 22.1.

The production $expr \rightarrow expr_1 + term$ derives an expression containing a plus operator. The

Table 22.1: Syntax-directed definition for infix to postfix translation.

Production	Semantic Rules
$expr \to expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel '+'$
$expr \to expr_1 - term$	$expr.t = expr_1.t \parallel term.t \parallel '-'$
$expr \to term$	$expr.t = term.t$
$term \to 0$	$term.t =' 0'$
$term \to 1$	$term.t =' 1'$
...	...
$term \to 9$	$term.t =' 9'$

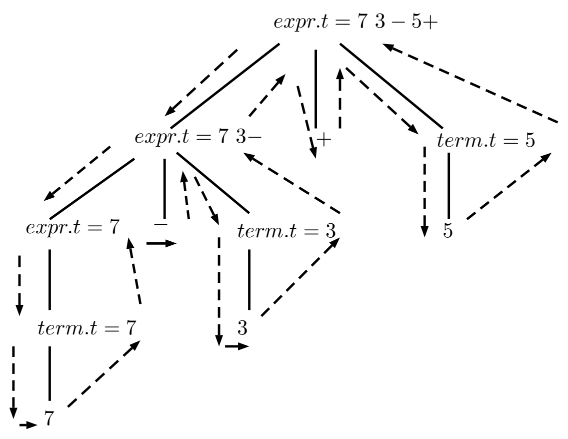

Figure 22.1: Annotated Parse Tree for $7 - 3 + 5$.

left operand of the plus operator is given by $expr_1$, and the right operand by *term*. The semantic rule

$$exper.t = expr_1.t \parallel term.t \parallel '+'$$

associated with this production constructs the value of attribute *expr.t* by concatenating the postfix forms $expr_1.t$ and *term.t* of the left and right operands, respectively, and then appending the plus sign. This rule is a formalization of the definition of "postfix expression." □

Definition 22.4 ***Annotated parse tree.*** *A parse tree showing the attribute values at each node is called an* annotated parse tree*.* □

For example, Fig. 22.1 shows an annotated parse tree for expression $7 - 3 + 5$ with an attribute t associated with the non-terminals *expr* and *term*. The value $73 - 5+$ of the attribute at the root node, which is the postfix notation for $7 - 3 + 5$. We shall see shortly how these expressions are computed.

Definition 22.5 Simple SDT. *The syntax-directed definition in Example 22.3 has the following important property: the string representing the translation of the nonterminal at the head of each production is the concatenation of the translations of the non-terminals in the production body, in the same order as in the production, with some optional additional strings interleaved.* □

Simple syntax-directed definition can be implemented by printing only the additional strings, in the order they appear in the definition.

22.3 Translation Schemes by adding semantic actions

The syntax-directed definition we discussed earlier (for infix to postfix translation) 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 increment ally, 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, as in

$$rest \to + \ \ term \ \ \{print('+')\} \ rest_1 \tag{22.4}$$

We shall see such rules when we consider an alternative form of grammar for expressions, where the non-terminal *rest* (equation 22.4) represents "everything but the first term of an expression." Again, the subscript in $rest_1$ distinguishes this instance of non-terminal *rest* in the production body from the instance of *rest* at the head of the production.

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. For example, the portion of the parse tree for the above production and action is shown in Fig. 22.2. The node for a semantic action (i.e., print '+' node) has no children, so the action is performed when this node is first seen.

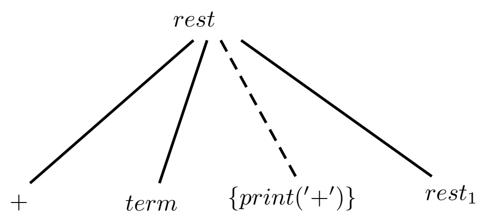

Figure 22.2: An extra leaf ($\{print('+')\}$) is constructed for a semantic action.

Definition 22.6 Preorder. *The Preorder list 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.*

Definition 22.7 Postorder. *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, then followed by N itself.*

Example 22.8 *Actions translating* $7 - 3 + 5$ *into* $7\ 3 - 5\ +$.

The parse tree in Fig. 22.3 has print statements at extra leaves, which are attached by dashed lines to interior nodes of the parse tree. The translation scheme appears in Table 22.2. 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.

Table 22.2: Actions for translating into postfix notation

$expr$	$\rightarrow$	$expr_1 + term$	$\{print('+')\}$
$expr$	$\rightarrow$	$expr_1 - term$	$\{print('-')\}$
$expr$	$\rightarrow$	$term$	
$term$	$\rightarrow$	$0\ \{print('0')\}$	
$term$	$\rightarrow$	$1\ \{print('1')\}$	
...	...	...	
$term$	$\rightarrow$	$9\ \{print('9')\}$	

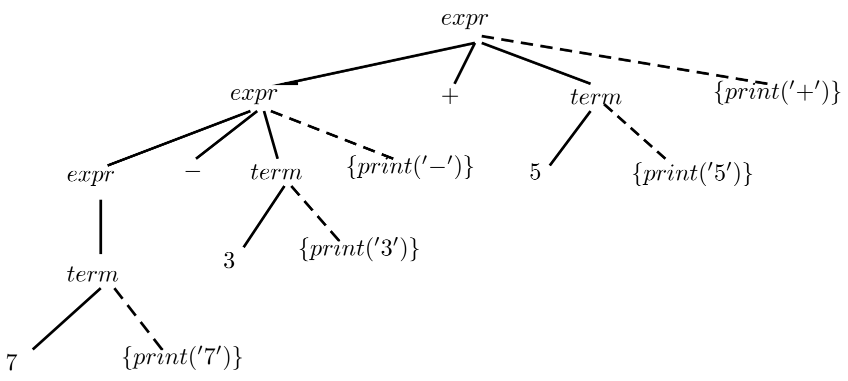

Figure 22.3: Actions translating $7 - 3 + 5$ into $7\ 3 - 5\ +$.

The root of Fig. 22.3 represents the first production in Tables. 22.2. In a postorder traversal, we first perform all the actions in the leftmost 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 {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. 22.3 print $7\ 3 - 5\ +$. □

22.3.1 Solved Problems

1. Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g., $-xy$ is the prefix notation for $x - y$. Give annotated parse trees for the inputs: a) $7 - 3 + 5$ and b) $7 - 3 * 5$.

Ans. Given Productions:

$$\begin{aligned} expr &\rightarrow expr + term \\ &\mid expr - term \\ &\mid term \end{aligned}$$

Translations schemes from infix notation into prefix are:

$$\begin{aligned} expr &\rightarrow \{print(\text{``}+\text{''})\}\ expr + term \\ &\mid \{print(\text{``}-\text{''})\}\ expr - term \\ &\mid term \end{aligned}$$

Ans. Given Productions:

$$\begin{aligned} expr &\rightarrow expr * term \\ &\mid expr/term \\ &\mid term \end{aligned}$$

Translations schemes from infix notation into prefix are:

$$\begin{aligned} expr &\rightarrow \{print(\text{``}*\text{''})\}\ expr * term \\ &\mid \{print(\text{``}/\text{''})\}\ expr/term \\ &\mid factor \\ factor &\rightarrow digit\ \{print(digit)\} \\ &\mid (expr) \end{aligned}$$

The complete answer is Ans_1 plus Ans_2.

2. Construct a syntax-directed translation scheme that translates postfix arithmetic expressions into equivalent prefix arithmetic expressions.

Ans. Given production is:

$$expr \rightarrow expr\ expr\ op \mid digit$$

Translation scheme:

$$expr \rightarrow \{print(op)\}\ expr\ expr\ op \mid digit\ \{print(digit)\}$$

22.4 Review Questions

1. When an attribute is called synthesized attribute?
2. What is special property of a synthesized attribute?
3. What can be the semantic actions, other than *print*? Give examples.
4. Why parentheses are not required in the postfix expression? Justify.
5. Define following:
   (a) Syntax Directed Definition (SDD)
   (b) Syntax Directed Translation (SDT)
   (c) Semantic Actions
   (d) Annotated Parse Tree
6. What is synthesized attribute?
7. What is attribute in syntax directed definition?
8. For computing from a postfix expression:
   (a) Original string is scanned number of times
   (b) Partly computed string is scanned many times
   (c) Original string is scanned once only
   (d) Original string is scanned in parallel
9. What is difference between parse-tree and annotated parse-tree?

22.5 Exercises

1. Describe in your own words, how you will evaluate a postfix expression?
2. Write an algorithm in your own words for evaluating postfix expression.
3. Convert the expression $(3+5)/4*2$ into postfix notation. Write all the necessary steps for this.
4. Give an inductive definition for obtaining postfix expression. Why the definition given in this chapter is not called as recursive definition.
5. Write an algorithm in your own language to convert any general expression into postfix notation.
6. Write an algorithm in your own language to evaluate any given expression of postfix expression, using inductive definition for the expression.
7. Convert the expression $(3 + 5)/4 * 2$ into postfix notation:
   (a) Using inductive definition
   (b) SDD and draw the annotated tree
8. Demonstrate computing a postfix expression obtained from $(3 + 5)/4 * 2$.

9. Construct an annotated parse tree for $3 * 5 + 7$.

10. Construct annotated parse trees for each of the following.

    (a) $3 * 2 - 8 + 5$

    (b) $5/2 + 7 - 6$

    (c) $8 * 3 + 6/2$

11. Construct a syntax-directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give annotated parse trees for the inputs $7\ 3 - 5\ *$ and $7\ 3\ 5\ * -$.

12. Construct a syntax-directed translation scheme that translates integers into Roman numerals.

13. Construct a syntax-directed translation scheme that translates Roman numerals into integers.

14. Explain the difference between *synthesized* and *inherited* attributed attributes. Give examples for each.

15. Define preorder and postorder traversals of trees. Give one example of each. Find out the time and space complexities of standard free and post order travels.

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`.