

Syntax-Directed Translation

Compiler Construction, Lecture 11

K.R. Chowdhary, Professor of CS

September 25, 2019 | Fall 2019

Syntax-Directed Definitions and Translation Schemes

An **attribute** is any quantity attached to a construct: a type, the number of generated instructions, the location of the first instruction. Attributes are attached to grammar symbols.

Syntax-directed definition (SDD)

Attributes on symbols plus **semantic rules** on productions.

$E \rightarrow E_1 + T$

$E.code = E_1.code \parallel T.code \parallel +$

More readable, so better for specifications.

Translation scheme (SDT)

Semantic actions in braces inside production bodies.

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

The action's position fixes when it runs, so it is more efficient and better for implementation.

- **General approach:** build a parse or syntax tree, then compute attributes by visiting its nodes
- **L-attributed** (left to right) translations cover virtually all translations done during parsing
- **S-attributed** (synthesized) translations are easy to do with a bottom-up parse

Inductive definition

1. A variable or constant E is its own postfix form
2. $E_1 \text{ op } E_2$ has postfix form $E'_1 E'_2 \text{ op}$, where E'_i is the postfix form of E_i
3. (E_1) has the same postfix form as E_1

■ $(7 - 3) + 5 \Rightarrow 7 \ 3 \ - \ 5 \ +$

■ $7 - (3 + 2) \Rightarrow 7 \ 3 \ 2 \ + \ -$

Postfix needs no parentheses: operator position and arity allow only one decoding.

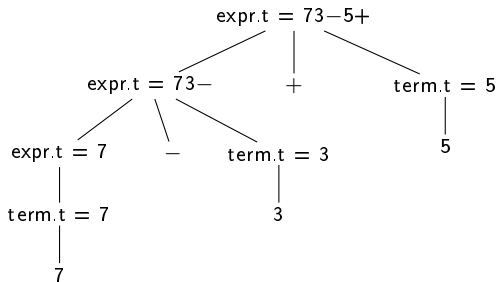
Evaluating $9 \ 5 \ 2 \ + \ - \ 3 \ *$

Scan from the left to the first operator, take its operands, replace them by the result, repeat:

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

Evaluation is a run-time job, not the compiler's.

SDD for Infix to Postfix and the Annotated Parse Tree



Annotated parse tree for $7 - 3 + 5$: the root gets $73 - 5 +$.

Production $\Rightarrow$ Semantic rule

$\text{expr} \rightarrow \text{expr}_1 + \text{term}$

$\text{expr.t} = \text{expr}_1.\text{t} \parallel \text{term.t} \parallel '+'$

$\text{expr} \rightarrow \text{expr}_1 - \text{term}$

$\text{expr.t} = \text{expr}_1.\text{t} \parallel \text{term.t} \parallel '-'$

$\text{expr} \rightarrow \text{term}$

$\text{expr.t} = \text{term.t}$

$\text{term} \rightarrow 0 \dots \text{term} \rightarrow 9$

$\text{term.t} = '0' \dots '9'$

A **synthesized** attribute at a node depends only on the node's children (and the node itself), so it can be computed in one bottom-up traversal.

Traversal orders

Preorder: the node, then the preorder of each child subtree, left to right.

Postorder: the postorder of each child subtree, left to right, then the node.

Procedure *visit(N)*

for each child *C* of *N*, left to right: *visit(C)*
evaluate the semantic rules at node *N*

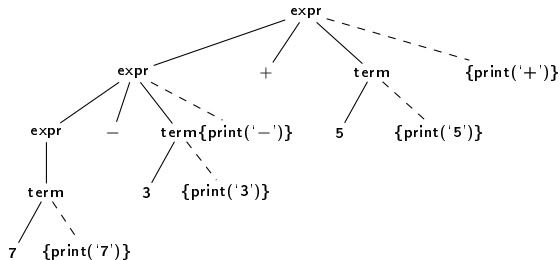
This depth-first traversal computes each node's translation after its children's.

- An SDD fixes no evaluation order: any order that computes an attribute after those it depends on is acceptable
- Synthesized attributes can be evaluated in *any* bottom-up traversal
- With inherited attributes as well, the evaluation order is much more complex

Simple SDD

The head's translation is the concatenation of the body non-terminals' translations, in the same order, with optional extra strings. It can be implemented by printing only the extra strings.

Translation Scheme: Actions and Parse-Tree Leaves



Actions as dashed extra leaves; output **7 3 - 5 +**.

Scheme (Table 11.2)

$\text{expr} \rightarrow \text{expr}_1 + \text{term} \{\text{print}('+')\}$

$\text{expr} \rightarrow \text{expr}_1 - \text{term} \{\text{print}('-')\}$

$\text{expr} \rightarrow \text{term}$

$\text{term} \rightarrow 0 \{\text{print}('0')\}$

...

$\text{term} \rightarrow 9 \{\text{print}('9')\}$

A scheme fixes evaluation order. Do a left-to-right depth-first traversal and run each action when its leaf is reached. Only operators and digits are printed; $\text{expr} \rightarrow \text{term}$ prints nothing.

Infix to prefix

Action *before* the operands:

```
expr → {print('+')} expr + term  
      | {print('-')} expr - term  
      | term
```

Similarly for * and /.

Review

1. When is an attribute synthesized?
2. Why no parentheses in postfix?
3. SDD vs. SDT? Parse tree vs. annotated tree?

Exercises

1. Convert $(3 + 5)/4 * 2$ to postfix, by the inductive definition and by an SDD with an annotated tree
2. Algorithms to convert an expression to postfix and to evaluate a postfix expression
3. Annotated parse trees for $3 * 5 + 7$, $3 * 2 - 8 + 5$, $5/2 + 7 - 6$, $8 * 3 + 6/2$
4. Schemes translating postfix to infix (inputs $7\ 3 - 5*$ and $7\ 3\ 5 * -$) and postfix to prefix
5. Schemes for integers to Roman numerals, and back
6. Synthesized vs. inherited attributes, with examples
7. Preorder and postorder traversals: examples, time and space complexity