CS 4300: Compiler Theory

Chapter 5
Syntax-Directed Translation

Dr. Xuejun Liang
Quick Review of Last Two Lectures

• Parser Generator Yacc and Bison
  – Yacc Specification
  – Writing a Grammar in Yacc
  – Dealing With Ambiguous Grammars
  – Resolve Parsing Action Conflicts
  – Combining Lex/Flex with Yacc/Bison
  – Error Recovery in Yacc

• Programming project III

• Review of homework assignments #6 and #7
Outlines (Sections)

1. Syntax-Directed Definitions
2. Evaluation Orders for SDD's
3. Applications of Syntax-Directed Definition
4. Syntax-Directed Translation Schemes
5. Implementing L-Attributed SDD's
1. Syntax-directed Definition

- A syntax-directed definition (SDD) specifies the values of attributes by associating semantic rules with the grammar productions

\[
\text{Production} \quad E \rightarrow E_1 + T \\
\text{Semantic Rule} \quad E\text{.code} = E_1\text{.code} \parallel T\text{.code} \parallel '+'
\]

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

\[
E \rightarrow E_1 + T \quad \{ \text{print '+'} \}
\]

- 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
Attributes

• A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
• An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.
• Attribute values typically represent
  – Numbers (literal constants)
  – Strings (literal constants)
  – Memory locations, such as a frame index of a local variable or function argument
  – A data type for type checking of expressions
  – Scoping information for local declarations
  – Intermediate program representations
Example Syntax-directed Definition

A simple desk calculator

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E \mathbf{n}$</td>
<td>$L.val = E.val$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.val = E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val = T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$T.val = T_1.val * F.val$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.val = F.val$</td>
</tr>
<tr>
<td>$F \rightarrow ( E )$</td>
<td>$F.val = E.val$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F.val = \text{digit}$.lexval</td>
</tr>
</tbody>
</table>

Note: all attributes in this example are of the synthesized type

An SDD with only synthesized attributes is called **S-attributed**.

An SDD without side effects is called an **attribute grammar**.
Annotated Parse Tree for $3 \times 5 + 4$ n

A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.

$L \rightarrow E \ n$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 \times F$

$T \rightarrow F$

$F \rightarrow ( E )$

$F \rightarrow \text{digit}$
Annotating a Parse Tree With Depth-First Traversals

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

```pascal
procedure visit(n : node);
begin
  for each child m of n, from left to right do
    visit(m);
  evaluate semantic rules at node n
end
```
An SDD Based on a Grammar Suitable for Top-down Parsing

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| 1) $T \rightarrow FT'$ | $T'.inh = F.val$  
                              $T.val = T'.syn$  |
| 2) $T' \rightarrow *FT_1'$ | $T'_1.inh = T'.inh \times F.val$  
                              $T'.syn = T'_1.syn$  |
| 3) $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$  |
| 4) $F \rightarrow \text{digit}$ | $F.val = \text{digit.lexval}$  |

An inherited attribute for nonterminal $T'$ is used to pass the operand to the operator:

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow \text{digit}$
Annotated Parse Tree for $3 \times 5$

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
</table>
| $T \rightarrow FT'$ | $T'.inh = F.val$
|               | $T.val = T'.syn$               |
| $T' \rightarrow * FT'_1$ | $T'_1.inh = T'.inh \times F.val$
|               | $T'.syn = T'_1.syn$           |
| $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$            |
| $F \rightarrow \text{digit}$ | $F.val = \text{digit. lexval}$ |
Example Attribute Grammar with Synthesized & Inherited Attributes

### Simple Type Declaration

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow TL$</td>
<td>$L\text{.inh} = T\text{.type}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$T\text{.type} = 'integer'$</td>
</tr>
<tr>
<td>$T \rightarrow \text{float}$</td>
<td>$T\text{.type} = 'float'$</td>
</tr>
<tr>
<td>$L \rightarrow L_1, \text{id}$</td>
<td>$L_1\text{.inh} = L\text{.inh};$ addtype(id.entry, L.inh)</td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>addtype(id.entry, L.inh)</td>
</tr>
</tbody>
</table>

Synthesized: $T\text{.type}$, id.entry  
Inherited: $L\text{.inh}$

Treated as dummy synthesized attribute with the head
2. Evaluation Orders for SDD's

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree.

\[ A \rightarrow X Y \]

\[ A.a = f(X.x, Y.y) \]

\[ X.x = f(A.a, Y.y) \]

\[ Y.y = f(A.a, X.x) \]
Evaluation Orders for SDD 's (Cont.)

• Edges in the dependency graph determine the evaluation order for attribute values
  – Dependency graphs cannot be cyclic
• So, dependency graph is a directed acyclic graph (DAG)

\[
\begin{align*}
A & \rightarrow X Y \\
X.x & := f(Y.y) \\
Y.y & := f(A.a)
\end{align*}
\]

Error: cyclic dependence
Annotated Parse Tree for 3*5 with Dependency Graph
Annotated Parse Tree for 3*5 with Dependency Graph
Annotated Parse Tree

float $id_1$, $id_2$, $id_3$

\[
D 
\xrightarrow{D \rightarrow TL}
T \xrightarrow{T \rightarrow int}
T \xrightarrow{T \rightarrow float}
L \xrightarrow{L \rightarrow L_1, id}
L \xrightarrow{L \rightarrow id}
\]

$L.\text{inh} = T.\text{type}$
$T.\text{type} = \text{'integer'}$
$T.\text{type} = \text{'float'}$
$L_1.\text{inh} = L.\text{inh};$
\text{addtype}(id.\text{entry}, L.\text{inh})$
\text{addtype}(id.\text{entry}, L.\text{inh})$
Annotated Parse Tree with Dependency Graph

float id₁, id₂, id₃

D → TL
T → int
T → float
L → L₁, id
L → id

L.inh = T.type
T.type = 'integer'
T.type = 'float'
L₁.inh = L.inh;
addtype(id.entry, L.inh)
addtype(id.entry, L.inh)
Evaluation Order

• A **topological sort** of a directed acyclic graph (DAG) is any ordering \( m_1, m_2, \ldots, m_n \) of the nodes of the graph, such that if \( m_i \rightarrow m_j \) is an edge, then \( m_i \) appears before \( m_j \).

• Any topological sort of a dependency graph gives a valid evaluation order of the semantic rules.

• Example: Topological orders of DAG on slide 15
  – 1, 2, 3, 4, 5, 6, 7, 8, 9.
  – 1, 3, 5, 2, 4, 6, 7, 8, 9.
• Example: Topological orders of the following DAG
  - 1, 2, 3, 4, 5, 6, 7, 8, 9.
  - 1, 3, 5, 2, 4, 6, 7, 8, 9.
Example Parse Tree with Topologically Sorted Actions

Topological sort:
1. Get \text{id}_1.entry
2. Get \text{id}_2.entry
3. Get \text{id}_3.entry
4. \text{T}_1.type = 'float'
5. \text{L}_1.inh = 'float'
6. \text{addtype(\text{id}_3.entry, L}_1.inh)
7. \text{L}_2.inh = \text{L}_1.inh
8. \text{addtype(\text{id}_2.entry, L}_2.inh)
9. \text{L}_3.inh = \text{L}_2.inh
10. \text{addtype(\text{id}_1.entry, L}_3.inh)

float \text{id}_1, \text{id}_2, \text{id}_3
L-Attributed Definitions

- A syntax-directed definition is **L-attributed** if each inherited attribute of $X_j$ on the right side of production $A \rightarrow X_1 X_2 \ldots X_n$ depends only on

  1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of $A$

  Shown: dependences of inherited attributes

```
  A.a
  X_1.x  X_2.x
```

- L-attributed definitions allow for a natural order of evaluating attributes: **depth-first and left to right**

- Note: every $S$-attributed syntax-directed definition is also L-attributed
## 3. Applications of SDD

**Construction of Syntax Trees**

### S-attributed Definition for Simple Expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E\text{.node} = \text{new Node}(\text{'+', } E_1\text{.node}, T\text{.node})$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 - T$</td>
<td>$E\text{.node} = \text{new Node}(\text{'-', } E_1\text{.node}, T\text{.node})$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E\text{.node} = T\text{.node}$</td>
</tr>
<tr>
<td>$T \rightarrow (E)$</td>
<td>$T\text{.node} = E\text{.node}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T\text{.node} = \text{new Leaf}(\text{id, id.entry})$</td>
</tr>
<tr>
<td>$T \rightarrow \text{num}$</td>
<td>$T\text{.node} = \text{new Leaf}(\text{num, num.val})$</td>
</tr>
</tbody>
</table>

**Note:** This is a S-attributed definition, then can be done during bottom-up parsing.
Example: Syntax Tree for $a - 4 + c$

Steps in the construction of the syntax tree

1) $p_1 = \text{new Leaf(id, entry-a)}$;
2) $p_2 = \text{new Leaf(num, 4)}$;
3) $p_3 = \text{new Node(’,’, } p_1, p_2)$;
4) $p_4 = \text{new Leaf(id, entry-c)}$;
5) $p_5 = \text{new Node(’+’, } p_3, p_4)$;

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $E \rightarrow E_1 + T$</td>
<td>$E$.node = new Node(’+’, $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>2) $E \rightarrow E_1 - T$</td>
<td>$E$.node = new Node(’-’, $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>3) $E \rightarrow T$</td>
<td>$E$.node = $T$.node</td>
</tr>
<tr>
<td>4) $T \rightarrow (E)$</td>
<td>$T$.node = $E$.node</td>
</tr>
<tr>
<td>5) $T \rightarrow \text{id}$</td>
<td>$T$.node = new Leaf(id, id.entry)</td>
</tr>
<tr>
<td>6) $T \rightarrow \text{num}$</td>
<td>$T$.node = new Leaf(num, num.val)</td>
</tr>
</tbody>
</table>
Constructing Syntax Tree During Top-Down Parsing

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
</table>
| 1) $E \rightarrow T E'$ | $E$.node = $E'$.syn  
                        | $E'$.inh = $T$.node |
| 2) $E' \rightarrow + T E'_1$ | $E'_1$.inh = new Node('+', $E'$.inh, $T$.node)  
                                | $E'$.syn = $E'_1$.syn |
| 3) $E' \rightarrow - T E'_1$ | $E'_1$.inh = new Node('-', $E'$.inh, $T$.node)  
                                | $E'$.syn = $E'_1$.syn |
| 4) $E' \rightarrow \epsilon$ | $E'$.syn = $E'$.inh |
| 5) $T \rightarrow ( E )$ | $T$.node = $E$.node |
| 6) $T \rightarrow id$ | $T$.node = new Leaf(id, id.entry) |
| 7) $T \rightarrow num$ | $T$.node = new Leaf(num, num.val) |