Lecture 8: Syntax-Directed Translation

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
A Step-Back

source program \rightarrow \text{Lexical Analyzer} \rightarrow \text{tokens} \rightarrow \text{Parser} \rightarrow \text{syntax tree} \rightarrow \text{Intermediate Code Generator} \rightarrow \text{three-address code}

Symbol Table
Chapter 3

• Strings
• Regular expressions
• Tokens
• Transition diagrams
• Finite Automata
Chapter 4

• Grammars
• Derivations
• Parse-trees
• Top-down parsing (LL)
• Bottom-up paring (LR, SLR, LALR)
We Need Some Tools

• To help in semantic analysis
• To help in intermediate code generation
• Two such tools
  – Semantic rules (Syntax-Directed Definitions)
    \[
    \text{Production} \quad E \rightarrow E_1 + T \quad \text{Semantic Rule} \quad E.\text{code} = E_1.\text{code} \ || \ T.\text{code} \ || \ '+'
    \]
  – Semantic actions (Syntax Directed Translations)
    \[
    E \rightarrow E_1 + T \ \{ \ \text{print } '+' \ \}
    \]
Syntax-Directed Definitions

• Context-free grammar
• With attributes and rules to calculate the attributes

\[
\text{Production} \quad E \rightarrow E_1 + T \\
\text{Semantic Rule} \quad E.code = E_1.code \; || \; T.code \; || \; '+'
\]
Two Types of Attributes
Two Types of Attributes

Synthesized Attributes
Attribute of the node is defined in terms of:
• Attribute values at children of the node
• Attribute value at node itself

SDD involving only synthesized attributes is called **S-attributed**
Two Types of Attributes

Inherited Attributes

Attribute of the node is defined in terms of:
• Attribute values at parent of the node
• Attribute values at siblings
• Attribute value at node itself
A parse tree showing the values of its attributes is called *annotated parse tree*.
**Example**

Give the annotated parse tree of \((3+4) \times (5+6)n\)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) (L \rightarrow E \ n)</td>
<td>(L.\text{val} = E.\text{val})</td>
</tr>
<tr>
<td>2) (E \rightarrow E_1 + T)</td>
<td>(E.\text{val} = E_1.\text{val} + T.\text{val})</td>
</tr>
<tr>
<td>3) (E \rightarrow T)</td>
<td>(E.\text{val} = T.\text{val})</td>
</tr>
<tr>
<td>4) (T \rightarrow T_1 \ast F)</td>
<td>(T.\text{val} = T_1.\text{val} \times F.\text{val})</td>
</tr>
<tr>
<td>5) (T \rightarrow F)</td>
<td>(T.\text{val} = F.\text{val})</td>
</tr>
<tr>
<td>6) (F \rightarrow (E))</td>
<td>(F.\text{val} = E.\text{val})</td>
</tr>
<tr>
<td>7) (F \rightarrow \text{digit})</td>
<td>(F.\text{val} = \text{digit.\text{lexval}})</td>
</tr>
</tbody>
</table>
When Are Inherited Attributes Useful?

<table>
<thead>
<tr>
<th>PRODUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $T \rightarrow F T'$</td>
</tr>
<tr>
<td>2) $T' \rightarrow * F T_1'$</td>
</tr>
<tr>
<td>3) $T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>4) $F \rightarrow \text{digit}$</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $D \rightarrow T , L$</td>
<td>$L.inh = T.type$</td>
</tr>
<tr>
<td>2) $T \rightarrow \text{int}$</td>
<td>$T.type = \text{integer}$</td>
</tr>
<tr>
<td>3) $T \rightarrow \text{float}$</td>
<td>$T.type = \text{float}$</td>
</tr>
<tr>
<td>4) $L \rightarrow L_1 \ , \ id$</td>
<td>$L_1.inh = L.inh$</td>
</tr>
<tr>
<td></td>
<td>$\text{addType(id.entry, L.inh)}$</td>
</tr>
<tr>
<td>5) $L \rightarrow id$</td>
<td>$\text{addType(id.entry, L.inh)}$</td>
</tr>
</tbody>
</table>

Give annotated parse-trees for: int $a, b, c$
Evaluation Orders of SDDs

- Annotated parse tree shows attribute values
- Dependency graph helps us determine how those values are computed
Topological Order
S-Attributed Definitions

- Every attribute is synthesized
- We can evaluate its attribute in any bottom-up order of the nodes of the parse tree
  (e.g. postorder traversal -> LR parser).
L-Attributed Definitions

• Dependency graph edges can only go from left to right
  – i.e. use attributes from above or from the left
### Example

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $D \rightarrow TL$</td>
<td>$L.inh = T.type$</td>
</tr>
<tr>
<td>2) $T \rightarrow \text{int}$</td>
<td>$T.type = \text{integer}$</td>
</tr>
<tr>
<td>3) $T \rightarrow \text{float}$</td>
<td>$T.type = \text{float}$</td>
</tr>
<tr>
<td>4) $L \rightarrow L_1, \text{id}$</td>
<td>$L_1.inh = L.inh$</td>
</tr>
<tr>
<td></td>
<td>$\text{addType}(id.entry, L.inh)$</td>
</tr>
<tr>
<td>5) $L \rightarrow \text{id}$</td>
<td>$\text{addType}(id.entry, L.inh)$</td>
</tr>
</tbody>
</table>

![Diagram](image)
Syntax-Directed Translations

• Context-free grammar
• Can implement SDDs
• Program fragments embedded within production bodies
  – called semantic rules
  – Can appear anywhere within the production body
• Steps are usually as follows
  – Build parse tree
  – perform actions as you traverse left-to-right, depth-first (preorder)
$L \rightarrow E \textbf{n} \quad \{ \text{print}(E.\text{val}); \}$
$E \rightarrow E_1 + T \quad \{ E.\text{val} = E_1.\text{val} + T.\text{val}; \}$
$E \rightarrow T \quad \{ E.\text{val} = T.\text{val}; \}$
$T \rightarrow T_1 * F \quad \{ T.\text{val} = T_1.\text{val} \times F.\text{val}; \}$
$T \rightarrow F \quad \{ T.\text{val} = F.\text{val}; \}$
$F \rightarrow (E) \quad \{ F.\text{val} = E.\text{val}; \}$
$F \rightarrow \textbf{digit} \quad \{ F.\text{val} = \textbf{digit}.\text{lexval}; \}$
Implementing L-Attributed SDDs

• L-attributed definitions can be used in many translation applications

• Several methods of implementation
  – Build parse tree and annotate
  – Build parse tree, add actions, execute in preorder
  – Recursive descent
Recursive Descent

- Function $A$ for each nonterminal $A$
- Arguments of $A$ are inherited attributes of nonterminal $A$
- Return value of $A$ is the collection of synthesized attributes of $A$
Example

For that rule we want to generate labels:
L1: C
L2: S1

\[ S \rightarrow \textbf{while} \ (C) \ S_1 \]
\[ L_1 = \textit{new}(); \]
\[ L_2 = \textit{new}(); \]
\[ S_1\.next = L_1; \]
\[ C\.false = S\.next; \]
\[ C\.true = L_2; \]
\[ S\.code = \textbf{label} \ || \ L_1 \ || \ C\.code \ || \ \textbf{label} \ || \ L_2 \ || \ S_1\.code \]
$S \rightarrow \textbf{while} ( C ) S_1$

For that rule we want to generate labels:

L1: C
L2: S1

```c
string S(label next) {
    string Scode, Ccode; /* local variables holding code fragments */
    label L1, L2; /* the local labels */
    if ( current input == token \textbf{while} ) {
        advance input;
        check '(' is next on the input, and advance;
        $L1 = \text{new}()$;
        $L2 = \text{new}()$;
        $Ccode = C(next, L2)$;
        check ')' is next on the input, and advance;
        $Scode = S(L1)$;
        return("label" || $L1$ || $Ccode$ || "label" || $L2$ || $Scode$);
    }
    else /* other statement types */
}
```
Example

\[ S \rightarrow \textbf{while} \ (C) \ S_1 \]

For that rule we want to generate labels:
L1: C
L2: S1

```c
string S(label next) {
    string Scode, Ccode; /* local variables holding code fragment
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        check ')' is next on the input, and advance;
        Scode = S(L1);
        return("label" || L1 || Ccode || "label" || L2 || Scode
    }
    else /* other statement types */
}

void S(label next) {
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        check ')' is next on the input, and advance;
        print("label", L2);
        S(L1);
    }
    else /* other statement types */
}
Reading

• Skim: 5.3, 5.4.3, 5.4.4, 5.4.5, 5.5.3, and 5.5.4
• Read the rest