CSCI-GA.2130-001
Compiler Construction
Lecture 8:
Syntax-Directed Translation

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

Chapter 3

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

• Grammars
• Derivations
• Parse-trees
• Top-down parsing (LL)
• Bottom-up parsing (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)
    
    \[
    \text{PRODUCTION} \quad \text{SEMANTIC RULE} \\
    E \rightarrow E_1 + T \quad E.\text{code} = E_1.\text{code} \parallel T.\text{code} \parallel '+'
    \]
  
  – Semantic actions (Syntax Directed Translations)

    \[
    E \rightarrow E_1 + T \quad \{ \text{print } '+' \}
    \]
Syntax-Directed Definitions

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

**Production**

\[ E \rightarrow E_1 + T \]

**Semantic Rule**

\[ E.code = E_1.code \parallel T.code \parallel '+' \]
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 \textit{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 an *annotated parse tree*.
Give the annotated parse tree of \((3+4)\ast(5+6)n\)

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

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $T \rightarrow FT'$</td>
</tr>
<tr>
<td>2) $T' \rightarrow *FT'_1$</td>
</tr>
<tr>
<td>3) $T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>4) $F \rightarrow \text{digit}$</td>
</tr>
</tbody>
</table>
Example

Give annotated parse-trees for: int a, b, c

<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>
</tbody>
</table>
| 4) $L \rightarrow L_1 \ , \ id$ | $L_1.inh = L.inh$
| | $\text{addType}(id.entry, L.inh)$ |
| 5) $L \rightarrow \text{id}$ | $\text{addType}(id.entry, L.inh)$ |
Evaluation Orders of SDDs

• Annotated parse tree shows attribute values
• Dependency graph helps us determine how those values are computed
Topological Order
Cycles

• Arbitrary SDDs can have cycles.
• Cycles need to be avoided
  – Cannot proceed
  – Detecting cycles has exponential time-complexity.
• Two type of SDDs guarantee no-cycles:
  – $S$-attributed
  – $L$-attributed

\[
\begin{align*}
A.s &\rightarrow B \\
B.i &\rightarrow A.s + 1
\end{align*}
\]
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>$\text{addType}(\text{id}.entry, L.inh)$</td>
</tr>
<tr>
<td>5) $L \rightarrow \text{id}$</td>
<td>$\text{addType}(\text{id}.entry, L.inh)$</td>
</tr>
</tbody>
</table>
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)
Example

\[ L \rightarrow E \text{ n} \quad \{ \text{print}(E.\text{val}); \} \]
\[ E \rightarrow E_1 + T \quad \{ E.\text{val} = E_1.\text{val} + T.\text{val}; \} \]
\[ E \rightarrow T \quad \{ E.\text{val} = T.\text{val}; \} \]
\[ T \rightarrow T_1 * F \quad \{ T.\text{val} = T_1.\text{val} \times F.\text{val}; \} \]
\[ T \rightarrow F \quad \{ T.\text{val} = F.\text{val}; \} \]
\[ F \rightarrow ( E ) \quad \{ F.\text{val} = E.\text{val}; \} \]
\[ F \rightarrow \text{digit} \quad \{ F.\text{val} = \text{digit.\text{lexval}}; \} \]
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

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

For that rule we want to generate labels:

L1: \( C \)
L2: \( S_1 \)

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

\[
L1 = \text{new}(); \\
L2 = \text{new}(); \\
S_1.\text{next} = L1; \\
C.\text{false} = S.\text{next}; \\
C.\text{true} = L2; \\
S.\text{code} = \text{label} \parallel L1 \parallel C.\text{code} \parallel \text{label} \parallel L2 \parallel S_1.\text{code}
\]
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 fragments */
    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 */
}
```
Example

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

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 */
}
```

```c
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