Syntax-Directed Translation

Chapter 5
Bad Translation

• CS581 is a very boring class
  – Use babelfish translator
  – English to Chinese, then Chinese to English

• CS581 is the extremely tasteless kind
Motivation: parser as a translator

Syntax-directed translation

- Stream of tokens
- Parser
- Syntax + translation rules (typically hardcoded in the parser)
- ASTs, or assembly code
Syntax-Directed Definitions

• A syntax-directed definition (SDD) is a context free grammar together with attributes and rules
  – Attributes are associated with grammar symbols
  – Rules are associated with productions
• A synthesized attribute for A at a parse tree node N is defined only in terms of attributes at the children of N and at N itself
• An inherited attribute for B at a parse tree node N is defined only in terms of attributes at the N’s parent, N itself, and N’s siblings
SDD Example 1

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + T</td>
<td>( E_1.val = E_2.val + T.val )</td>
</tr>
<tr>
<td>E → T</td>
<td>( E.val = T.val )</td>
</tr>
<tr>
<td>T → T * F</td>
<td>( T_1.val = T_2.val * F.val )</td>
</tr>
<tr>
<td>T → F</td>
<td>( T.val = F.val )</td>
</tr>
<tr>
<td>F → int</td>
<td>( F.val = \text{int.lexval} )</td>
</tr>
<tr>
<td>F → ( E )</td>
<td>( F.val = E.val )</td>
</tr>
</tbody>
</table>

- An SDD that involves only synthesized attributes is called S-attributed
SDD Example 2

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T \rightarrow F \ T'$</td>
<td>$T'.inh = F.val$</td>
</tr>
<tr>
<td></td>
<td>$T.val = T'.syn$</td>
</tr>
<tr>
<td>$T' \rightarrow *FT'_1$</td>
<td>$T'_1.inh = T'.inh \times F.val$</td>
</tr>
<tr>
<td></td>
<td>$T'.syn = T'_1.syn$</td>
</tr>
<tr>
<td>$T' \rightarrow \epsilon$</td>
<td>$T'.syn = T'.inh$</td>
</tr>
<tr>
<td>$F \rightarrow \text{int}$</td>
<td>$F.val = \text{int.lexval}$</td>
</tr>
</tbody>
</table>

- Annotated parse tree for $3*5$
TEST YOURSELF #1

• A CFG for the language of binary numbers:
  \[ B \rightarrow 0 \]
  \[ \rightarrow 1 \]
  \[ \rightarrow B \ 0 \]
  \[ \rightarrow B \ 1 \]

• Define a syntax-directed translation so that the translation of a binary number is its base-10 value.

• Draw the parse tree for 1001 and annotate each nonterminal with its translation.
Evaluation Orders for SDDs

• S-Attributed Definitions

```
postorder(N) {
    for (each child C of N, from the left)
        postorder (C);
    evaluate the attributes associated with N;
}
```
L-Attributed Definitions

• Each attribute must be either
  – Synthesized, or
  – Inherited with limited rules. For \(A \rightarrow X_1X_2...X_n\), and there is an inherited attribute \(X_i.a\) computed by a rule. The rule must use only
    • Inherited attributes of head \(A\)
    • Either inherited or synthesized attributes of \(X_1,X_2,...X_{i-1}\)
    • Either inherited or synthesized attributes of \(X_i\), but only if there are no cycles

• \(A \rightarrow BC\) with semantic rules \(A.s=B.b\); \(B.i=f(C.c, A.s)\) is not L-attributed
AST vs Parse Tree

• AST is a better structure for later compiler stages
  – omits details having to do with the source language,
  – only contains information about the essential structure of the program.
    • e.g., parentheses, commas, semi-colons

• AST is condensed form of a parse tree
  – operators appear at internal nodes, not at leaves.
  – "Chains" of single productions are collapsed.
  – Lists are "flattened".
  – Syntactic details are ommitted
Example: 2 * (4 + 5): parse tree vs AST
Abstract Syntax Tree (AST)

- Derivation = sequence of applied productions
  - $S \rightarrow E + S \rightarrow 1 + S \rightarrow 1 + E \rightarrow 1 + 2$

- Parse tree = graph representation of a derivation
  - Doesn’t capture the order of applying the productions

- AST discards unnecessary information from the parse tree
AST Construction

• We want to explicitly construct the AST during the parsing phase
Constructing Syntax Tree

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E + T )</td>
<td>( E_1).node = new Node(‘+’, ( E_2 ).node, ( T ).node)</td>
</tr>
<tr>
<td>( E \rightarrow T )</td>
<td>( E ).node = ( T ).node</td>
</tr>
<tr>
<td>( T \rightarrow T * F )</td>
<td>( T_1).node = new Node(‘*’, ( T_2 ).node, ( F ).node)</td>
</tr>
<tr>
<td>( T \rightarrow F )</td>
<td>( T ).node = ( F ).node</td>
</tr>
<tr>
<td>( F \rightarrow \text{int} )</td>
<td>( F ).node = new leaf(int, int.val)</td>
</tr>
<tr>
<td>( F \rightarrow (\ E \ ) )</td>
<td>( F ).node = ( E ).node</td>
</tr>
</tbody>
</table>
TEST YOURSELF #2

• Illustrate the syntax-directed translation defined above by
  – drawing the parse tree for $2 + 3 \times 4$, and
  – annotating the parse tree with its translation
    • i.e., each nonterminal $X$ in the parse tree will have a pointer to the root of the AST subtree that is the translation of $X$. 
Constructing Syntax Tree

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T \rightarrow F \ T'$</td>
<td>$T$.node = $T'$.syn</td>
</tr>
<tr>
<td></td>
<td>$T'$.inh = F.node</td>
</tr>
<tr>
<td>$T' \rightarrow *FT'_1$</td>
<td>$T'_{1}.inh = \text{new node}(\text{&quot;*&quot;}, \ T'.inh, F.node)</td>
</tr>
<tr>
<td></td>
<td>$T'$.syn = $T'_{1}.syn</td>
</tr>
<tr>
<td>$T' \rightarrow \epsilon$</td>
<td>$T'$.syn = $T'$.inh</td>
</tr>
<tr>
<td>$F \rightarrow \text{int}$</td>
<td>$F$.node = \text{new leaf(int, int.val)}</td>
</tr>
</tbody>
</table>

- Annotated parse tree for 3*5
Example 2: Compute the type of an expression

\[
\begin{align*}
E \rightarrow E + E & \quad \text{if } ((E_2.\text{trans} == \text{INT}) \text{ and } (E_3.\text{trans} == \text{INT}) \\
& \quad \text{then } E_1.\text{trans} = \text{INT} \\
& \quad \text{else } E_1.\text{trans} = \text{ERROR} \\
E \rightarrow E \text{ and } E & \quad \text{if } ((E_2.\text{trans} == \text{BOOL}) \text{ and } (E_3.\text{trans} == \text{BOOL}) \\
& \quad \text{then } E_1.\text{trans} = \text{BOOL} \\
& \quad \text{else } E_1.\text{trans} = \text{ERROR} \\
E \rightarrow E == E & \quad \text{if } ((E_2.\text{trans} == E_3.\text{trans}) \text{ and } (E_2.\text{trans} \neq \text{ERROR})) \\
& \quad \text{then } E_1.\text{trans} = \text{BOOL} \\
& \quad \text{else } E_1.\text{trans} = \text{ERROR} \\
E \rightarrow \text{true} & \quad \text{E.\text{trans} = \text{BOOL}} \\
E \rightarrow \text{false} & \quad \text{E.\text{trans} = \text{BOOL}} \\
E \rightarrow \text{int} & \quad \text{E.\text{trans} = \text{INT}} \\
E \rightarrow (E) & \quad \text{E}_1.\text{trans} = E_2.\text{trans}
\end{align*}
\]
Syntax-Directed Translation Schemes (SDT)

• SDT is a CFG with program fragments embedded within production bodies
  – Any SDT can be implemented by first building a parse tree, but not efficient
  – Typically SDT’s are implemented during parsing
    • LR-grammar, and the SDD is S-attributed
    • LL-grammar, and the SDD is L-attributed
LL parsing: extend procedures for non-terminals

void parse_S() {
    switch (token) {
        case num: case '(': 
            parse_E();
            parse_S'();
            return;
        default: 
            ParseError();
    }
}

Chapter 5: Syntax-directed Translation
AST Construction - LR

• We again need to add code for explicit AST construction

• AST construction mechanism
  – Store parts of the tree on the stack
  – For each nonterminal symbol X on stack, also store the sub-tree rooted at X on stack
  – Whenever the parser performs a reduce operation for a production $X \rightarrow \gamma$, create an AST node for X
Implementing AST

<table>
<thead>
<tr>
<th>Production</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + T</td>
<td>stack[top-2]= new Node(‘+’,stack[top-2].node,stack[top].node);</td>
</tr>
<tr>
<td></td>
<td>top = top – 2;</td>
</tr>
<tr>
<td>E → T</td>
<td>stack[top-2]= new Node(‘+’,stack[top-2].node,stack[top].node);</td>
</tr>
<tr>
<td>T → T * F</td>
<td>top = top – 2;</td>
</tr>
<tr>
<td>T → F</td>
<td>stack[top-2]=stack[top].node;</td>
</tr>
<tr>
<td>F → int</td>
<td>top = top – 2;</td>
</tr>
<tr>
<td>F → ( E )</td>
<td>top = top – 2;</td>
</tr>
</tbody>
</table>
AST Construction for LR - Example

input string: “1 + 2 + 3”

Before reduction: $S \rightarrow E + S$

After reduction: $S \rightarrow E + S$