Syntax-Directed Translation
Part II
Chapter 5

Slides adapted from:
© Robert van Engelen, Florida State University
Alessandro Artale, Free University of Bolzano

Syntax-Directed Translation Schemes

- Syntax-directed translation schemes are more implementation oriented than SDD
- Syntax-directed translation schemes indicate the order in which semantic rules and attributes are to be evaluated
- Yacc/Bison use syntax-directed translation schemes internally
Syntax-Directed Translation Schemes

- A syntax-directed translation scheme (SDT) is an attribute grammar embedded with semantic actions

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \text{ + term} \{ \text{print(“+”) } \} \\
\text{expr} & \rightarrow \text{expr} \text{ - term} \{ \text{print(“-”) } \} \\
\text{expr} & \rightarrow \text{term} \\
\text{term} & \rightarrow 0 \{ \text{print(“0”) } \} \\
\text{term} & \rightarrow 1 \{ \text{print(“1”) } \} \\
\ldots & \ldots \\
\text{term} & \rightarrow 9 \{ \text{print(“9”) } \}
\end{align*}
\]
Example SDT (cont’d)

Transforming SDD into SDT

- SDT are useful to implement S-attributed / L-attributed SDD, even if SDT are a more general mechanisms
- We (later) provide several algorithms for the transformation from SDD to SDT
SDT Implementation

• SDT can be implemented by
  – Building a parse tree
  – Visit the tree in pre-order, i.e., left-to-right, depth-first order, executing actions as soon as they are encountered

• SDT can also be implemented *without* building the parse tree if
  – The underlying grammar is LL / LR and the SDD is S-attributed / L-attributed
• We (later) provide several algorithms for the implementation of SDT
Postfix SDT

- SDD is S-attributed and underlying grammar is LR

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

Postfix SDT

- Add all semantic actions at the end of the production right-hand side

\[
\begin{align*}
L \to E\ n & \quad \{\text{print}(E.\text{val})\} \\
E \to E_1 + T & \quad \{E.\text{val} = E_1.\text{val} + T.\text{val}\} \\
E \to T & \quad \{E.\text{val} = T.\text{val}\} \\
T \to T_1 * F & \quad \{T.\text{val} = T_1.\text{val} * F.\text{val}\} \\
T \to F & \quad \{T.\text{val} = F.\text{val}\} \\
F \to (\ E\ ) & \quad \{F.\text{val} = E.\text{val}\} \\
F \to \text{digit} & \quad \{F.\text{val} = \text{digit}.\text{lexval}\}
\]
Postfix SDT

- Postfix SDT can be implemented during LR parsing by executing actions at reduction time
- Extra fields are added to the stack to hold the values of synthesized attributes

Stack:

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>X.s</td>
<td>Y.s</td>
<td>Z.s</td>
</tr>
</tbody>
</table>

Example: Postfix SDT

\[
\begin{align*}
L &\rightarrow E \text{ n} & \{ \text{print}(stack[top - 1].val); \ top = top - 1; \} \\
E &\rightarrow E_1 + T & \{ stack[top - 2].val = stack[top - 2].val + stack[top].val; \ top = top - 2; \} \\
E &\rightarrow T \\
T &\rightarrow T_1 \ast F & \{ stack[top - 2].val = stack[top - 2].val \ast stack[top].val; \ top = top - 2; \} \\
T &\rightarrow F \\
F &\rightarrow (E) & \{ stack[top - 2].val = stack[top - 1].val; \ top = top - 2; \} \\
F &\rightarrow \text{digit} \\
\end{align*}
\]
Example: Postfix SDT

<table>
<thead>
<tr>
<th>Stack</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$ 3</td>
<td>3</td>
</tr>
<tr>
<td>$ F</td>
<td>3</td>
</tr>
<tr>
<td>$ T</td>
<td>3</td>
</tr>
<tr>
<td>$ T *</td>
<td>3 _</td>
</tr>
<tr>
<td>$ T * 5</td>
<td>3 _ 5</td>
</tr>
<tr>
<td>$ T * F</td>
<td>3 _ 5</td>
</tr>
<tr>
<td>$ T</td>
<td>15</td>
</tr>
<tr>
<td>$ E</td>
<td>15</td>
</tr>
<tr>
<td>$ E +</td>
<td>15 _</td>
</tr>
<tr>
<td>$ E + 4</td>
<td>15 _ 4</td>
</tr>
<tr>
<td>$ E + F</td>
<td>15 _ 4</td>
</tr>
<tr>
<td>$ E + T</td>
<td>15 _ 4</td>
</tr>
<tr>
<td>$ E</td>
<td>19</td>
</tr>
<tr>
<td>$ E n</td>
<td>19 _</td>
</tr>
<tr>
<td>$ L</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>3*5+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>*5+4n$</td>
<td>reduce $F \rightarrow \text{digit}$</td>
</tr>
<tr>
<td>*5+4n$</td>
<td>reduce $T \rightarrow F$</td>
</tr>
<tr>
<td>*5+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>5+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>+4n$</td>
<td>reduce $F \rightarrow \text{digit}$</td>
</tr>
<tr>
<td>+4n$</td>
<td>reduce $T \rightarrow T * F$</td>
</tr>
<tr>
<td>+4n$</td>
<td>reduce $E \rightarrow T$</td>
</tr>
<tr>
<td>+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>4n$</td>
<td>shift</td>
</tr>
<tr>
<td>n$</td>
<td>reduce $F \rightarrow \text{digit}$</td>
</tr>
<tr>
<td>n$</td>
<td>reduce $T \rightarrow F$</td>
</tr>
<tr>
<td>n$</td>
<td>reduce $E \rightarrow E + T$</td>
</tr>
<tr>
<td>n$</td>
<td>shift</td>
</tr>
<tr>
<td>$</td>
<td>reduce $L \rightarrow E n$</td>
</tr>
<tr>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

SDT for L-Attributed SDD

- SDD is L-attributed and underlying grammar is LL
- SDD to SDT transformation for rule
  
  \[ A \rightarrow \alpha B \beta \]

  - Embed action for computation of inherited attributes for $B$ immediately before $B$
  - Place action for computation of synthesized attributes for $A$ at the end of $\alpha B \beta$
Example: SDT for L-Attributed

Production | Semantic Rule
---|---
$D \rightarrow TL$ | $L.in = T.type$
$T \rightarrow \text{int}$ | $T.type = 'integer'$
$T \rightarrow \text{real}$ | $T.type = 'real'$
$L \rightarrow L_1, \text{id}$ | $L_1.in = L.in; \ \text{addtype(id.entry, L.in)}$
$L \rightarrow \text{id}$ | $\text{addtype(id.entry, L.in)}$

Translation Scheme

$D \rightarrow T \{ \ L.in = T.type \ \} L$
$T \rightarrow \text{int} \{ \ T.type = 'integer' \ \} $
$T \rightarrow \text{real} \{ \ T.type = 'real' \ \}$
$L \rightarrow \{ \ L_1.in = L.in \ \} L_1, \text{id} \{ \ \text{addtype(id.entry, L.in)} \ \}$
$L \rightarrow \text{id} \{ \ \text{addtype(id.entry, L.in)} \ \}$

Example: SDT for L-Attributed

- Generate code for while statement
  
  $S \rightarrow \text{while} ( \ C \ ) S_1$

- $S$ generates statements (while and others)
- $C$ generates Boolean expression
- We assume code for $C$ and $S_1$ is synthesized by some other productions not shown
Example: SDT for L-Attributed

• Use the following attributes:
  – Inherited $S.next$: label of code that follows $S$
  – Synthesized $S.code$: intermediate code for $S$, ending with a jump to $S.next$
  – Inherited $C.true$: label of code that must be executed if $C$ is true
  – Inherited $C.false$: label of code that must be executed if $C$ is false
  – Synthesized $C.code$: intermediate code for $C$, ending with a jump to either $C.true$ or $C.false$

Example: SDT for L-Attributed

• SDD for while statement

\[
S \rightarrow \textbf{while } (C) \, S_1 \quad L1 = \textit{new}(); \\
L2 = \textit{new}(); \\
S_1.next = L1; \\
C.false = S.next; \\
C.true = L2; \\
S.code = \textbf{label} \parallel L1 \parallel C.code \\
\parallel \textbf{label} \parallel L2 \parallel S_1.code ;
\]
Example: SDT for L-Attributed

- SDT for while statement

\[ S \rightarrow \textbf{while} \quad (\quad \{ \quad L_1 = \text{new}(); \quad L_2 = \text{new}(); \quad C.false = S.next; \quad C.true = L_2; \quad \} \quad C \quad ) \quad \{ \quad S.next = L_1; \quad \} \quad S_1 \quad \{ \quad S.code = \textbf{label} \| L_1 \| C.code \| \textbf{label} \| L_2 \| S_1.code; \quad \} \]
Implementation of L-Attributed SDD / SDT

• Assume underlying grammar is LL, we have three methods
• Recursive descent parser augmented with argument passing
• SDT in conjunction with LL parser
• SDT in conjunction with LR parser

L-Attributed SDD and Recursive Descent Parsers

• Parser has a function $A()$ for each nonterminal $A$
• The arguments of $A()$ are the inherited attributes for nonterminal $A$
• The return value of $A()$ is the collection of the synthesized attributes for nonterminal $A$
Example: Descent Parsers

SDT

\[
D \rightarrow T \{ L.in = T.type \} L \\
T \rightarrow \text{int} \{ T.type = \text{`integer'} \} \\
T \rightarrow \text{real} \{ T.type = \text{`real'} \} \\
L \rightarrow \{ L_i.in = L.in \} L_i, \text{id} \{ addtype(id.entry, L.in) \} \\
L \rightarrow \text{id} \{ addtype(id.entry, L.in) \}
\]

\[
\begin{align*}
\text{void } D() & \{ \\
\text{Type } & Ttype = T(); \\
\text{Type } & Lin = Ttype; \\
& L(Lin); \\
\text{Type } & T() \{ \\
\text{Type } & Ttype; \\
\text{if} (\text{current input} == \text{INT}) \{ \\
& Ttype = \text{TYPE_INT}; \\
& \text{advance input}; \} \\
\text{else if} (\text{current input} == \text{REAL}) \{ \\
& Ttype = \text{TYPE_REAL}; \\
& \text{advance input}; \} \\
& \text{else error();} \\
& \text{return } Ttype; \\
& \text{void } L(Type Lin) \{ \ldots \}
\end{align*}
\]

Output: synthesized attribute

Input: inherited attribute
Example: Descent Parsers

SDT

\[ S \rightarrow \text{while} \ ( \]
\[ \{ \ L1 = \text{new}(); \ L2 = \text{new}(); \]
\[ C.\text{false} = S.\text{next}; \ C.\text{true} = L2; \} \]
\[ C \) \]
\[ \{ S_{1}.\text{next} = L1; \} \]
\[ S_{1} \]
\[ \{ S.\text{code} = \text{label} \parallel L1 \parallel C.\text{code} \]
\[ \parallel \text{label} \parallel L2 \parallel S_{1}.\text{code} ; \} \]

Example: Descent Parsers

```c
string S(label next) {
string Scode, Ccode;    // local vars
label L1, L2;          // local labels
if (current input == "while") {
    advance input;
    check '(' and advance
    L1 = new();
    L2 = new();
    Ccode = C(next, L2);
    check ')' and advance
    Scode = S(L1);
    return ("label" \parallel L1 \parallel Ccode
             \parallel "label" \parallel L2
             \parallel Scode); }
else /* other statement types */
```
L-Attributed SDD and On-The-Fly Code Generation

- Storing generated code in synthesized attributes is inefficient, copying long strings is time consuming
- More efficient to incrementally generate code into some output buffer

L-Attributed SDD and On-The-Fly Code Generation

- Assumptions
  - Code generated in a string-valued, synthesized attribute
  - Code produced by concatenation of code at children *preserving the order*, with possibly additional symbols
- We can incrementally emit the string components that are not main attributes
Example

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

L-Attributed SDD and LL Parsers

- When nonterminal A is at the top of the stack, its parent and left siblings are no longer there
  - Impossible to transfer inherited attributes to A
- When nonterminal A is expanded, it disappears from the stack
  - Impossible to collect synthesized attributes for A
L-Attributed SDD and LL Parsers

• Extend LL stack to include
  – Nonterminal record for $A$ along with inherited attributes for $A$
  – Action record representing actions for $A$, placed above $A$’s nonterminal record
  – Synthesized record for $A$, representing synthesized attributes and copy actions, placed below $A$’s nonterminal record

L-Attributed SDD and LL Parsers

• Consider CFG production $A \rightarrow B \ C$
  – Action record for $C$ also used to store copies of inherited attributes, since $A$, $B$ are no longer in the stack when $C$ is processed
  – Synthesized record for $A$ is placed below $A$, and survives when $A$ (and $B$ and $C$) are removed from the stack
L-Attributed SDD and LL Parsers

Example

\[
S \\
next = x
\]
L-Attributed SDD and LR Parsers

- Assume L-attributed SDD over LL grammar
- We adapt the grammar to compute the translation during LR parsing
  - Convert SDD into SDT
  - Replace each embedded action \( \{a\} \) with marker nonterminal \( M_a \)
  - Add new production \( M_a \rightarrow \varepsilon \{a'\} \), with action \( a' \) adapted from \( a \)
L-Attributed SDD
and LR Parsers

• For rules $A \rightarrow \alpha M_a \beta$ and $M_a \rightarrow \epsilon \{a'\}$, modified action $a'$ defined as
  – Attributes of $A$ or attributes of symbols in $\alpha$ that are used by $a$ are copied as inherited attributes of $M_a$
  – Action $a'$ defined as $a$, but newly computed attributes become synthesized attributes of $M_a$

L-Attributed SDD
and LR Parsers

• Extra fields are added to the stack to hold the values of synthesized / inherited attributes

<table>
<thead>
<tr>
<th>stack</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>val_s</td>
<td></td>
<td>X.s</td>
<td>Y.s</td>
</tr>
<tr>
<td>val_i</td>
<td></td>
<td>X.i</td>
<td>Y.i</td>
</tr>
</tbody>
</table>
Example

SDT:

\[ S \rightarrow a A \{ C.i = f(A.s) \} C \]
\[ S \rightarrow b A B \{ C.i = f(A.s) \} C \]
\[ C \rightarrow c \{ C.s = g(C.i) \} \]

Transformed SDT:

\[ S \rightarrow a A M_1 C \]
\[ S \rightarrow b A B M_2 C \]
\[ M_1 \rightarrow \epsilon \{ val_s[top + 1] = f(val_s[top]); top = top + 1 \} \]
\[ M_2 \rightarrow \epsilon \{ val_s[top + 1] = f(val_s[top - 1]); top = top + 1 \} \]
\[ C \rightarrow c \{ val_i[top] = g(val_i[top - 1]) \} \]

L-Attributed SDD
and LR Parsers

- If underlying grammars is strictly LR, we cannot do L-attributed SDD
- Consider production \( A \rightarrow B C \)
  - Inherited attributes of \( B \) depend on inherited attributes of \( A \)
  - But in LR parsing, when \( B \) is processed \( A \) is not known
Summary of Presented Algorithms

• SDD > SDT transformation
  – S-attributed SDD > Postfix SDT
  – L-attributed SDD > SDT

• SDT implementation
  – Postfix SDT via LR
  – L-attributed, LL SDT via recursive descent
  – L-attributed, LL SDT via LL parsing
  – L-attributed, LL SDT via LR parsing