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

```
rest → + term { print(“+”) } rest
```

Example SDT

```
expr → expr + term { print(“+”) }
expr → expr - term { print(“-”) }
expr → term
term → 0 { print(“0”) }
term → 1 { print(“1”) }
...
```

```
term → 9 { print(“9”) }
```
Example SDT (cont’ d)

Translates $9-5+2$ into postfix $95-2+$

Transforming SDD into SDT

- SDT are useful for implementing 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 left-to-right, pre-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
Summary of Algorithms to Be Presented

- 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

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 \rightarrow E , n )</td>
<td>\textit{print}(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{\texttt{digit}} )</td>
<td>( F.val = \texttt{digit.lexval} )</td>
</tr>
</tbody>
</table>
Postfix SDT

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

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

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

L \rightarrow E \ n \quad \{\text{print}(\text{stack}[\text{top} - 1].\text{val}); \ \text{top} = \text{top} - 1;\}
E \rightarrow E_1 + T \quad \{\text{stack}[\text{top} - 2].\text{val} = \text{stack}[\text{top} - 2].\text{val} + \text{stack}[\text{top}].\text{val}; \ \text{top} = \text{top} - 2;\}
E \rightarrow T
T \rightarrow T_1 * F \quad \{\text{stack}[\text{top} - 2].\text{val} = \text{stack}[\text{top} - 2].\text{val} * \text{stack}[\text{top}].\text{val}; \ \text{top} = \text{top} - 2;\}
T \rightarrow F
F \rightarrow (E) \quad \{\text{stack}[\text{top} - 2].\text{val} = \text{stack}[\text{top} - 1].\text{val}; \ \text{top} = \text{top} - 2;\}
F \rightarrow \text{digit}

---

Example: Postfix SDT

<table>
<thead>
<tr>
<th>Stack</th>
<th>value</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>_</td>
<td>3*5+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>$ 3</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce F \rightarrow digit</td>
</tr>
<tr>
<td>$ F</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce T \rightarrow F</td>
</tr>
<tr>
<td>$ T</td>
<td>3</td>
<td>*5+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>$ T * $</td>
<td>3 _ 5</td>
<td>5+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>$ T * 5</td>
<td>3 _ 5</td>
<td>+4n$</td>
<td>reduce F \rightarrow digit</td>
</tr>
<tr>
<td>$ T * F</td>
<td>3 _ 5</td>
<td>+4n$</td>
<td>reduce T \rightarrow T * F</td>
</tr>
<tr>
<td>$ T</td>
<td>15</td>
<td>+4n$</td>
<td>reduce E \rightarrow T</td>
</tr>
<tr>
<td>$ E</td>
<td>15</td>
<td>+4n$</td>
<td>shift</td>
</tr>
<tr>
<td>$ E + $</td>
<td>15 _</td>
<td>4n$</td>
<td>shift</td>
</tr>
<tr>
<td>$ E + 4</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce F \rightarrow digit</td>
</tr>
<tr>
<td>$ E + F</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce T \rightarrow F</td>
</tr>
<tr>
<td>$ E + T</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce E \rightarrow E + T</td>
</tr>
<tr>
<td>$ E</td>
<td>19</td>
<td>n$</td>
<td>shift</td>
</tr>
<tr>
<td>$ E n</td>
<td>19 _</td>
<td>$</td>
<td>reduce L \rightarrow E n</td>
</tr>
<tr>
<td>$ L</td>
<td>19</td>
<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 I: SDT for L-Attributed

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

Translation Scheme

- $D \rightarrow T \{ \ L.\text{in} = T.\text{type} \ \} \ L$
- $T \rightarrow \text{int} \ \{ \ T.\text{type} = \text{’integer’} \ \} $
- $T \rightarrow \text{real} \ \{ \ T.\text{type} = \text{’real’} \ \} $
- $L \rightarrow \{ \ L_1.\text{in} = L.\text{in} \ \} \ L_1, \text{id} \ \{ \ \text{addtype(id}.\text{entry}, L.\text{in}) \ \} $
- $L \rightarrow \text{id} \ \{ \ \text{addtype(id}.\text{entry}, L.\text{in}) \ \} $
Example II: SDT for L-Attributed

- Generate code for while statement
  \[ S \rightarrow \textbf{while} \ ( C ) \ S_1 \]
- \( S \) 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 II: 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 II: SDT for L-Attributed

- SDD for while statement

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $S \rightarrow \text{while ( C ) } S_1$ | $L1 = \text{new}();$
|                     | $L2 = \text{new}();$
|                     | $S_1\text{.next} = L1;$
|                     | $C\text{.false} = S\text{.next}.$
|                     | $C\text{.true} = L2;$
|                     | $S\text{.code} = \text{label } \| \, L1 \| \, C\text{.code}$
|                     | $\| \, \text{label } \| \, L2 \| \, S_1\text{.code};$ |
Example II: SDT for L-Attributed

$S \rightarrow \text{while } ( \{ \text{L1 = new(); L2 = new(); } \; \text{C.false = S.next; C.true = L2; } \} \; \text{C }) \{ S.next = L1; \} \; S_1 \{ S.code = \text{label } || \text{L1 } || C.code \; \text{label } || \text{L2 } || S_1.code ; \}$
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

\[ D \rightarrow T \ { L.in = T.type } L \]
\[ T \rightarrow \text{int} \ { T.type = \text{‘integer’} } \]
\[ T \rightarrow \text{real} \ { T.type = \text{‘real’} } \]
\[ L \rightarrow \{ L_1.in = L.in \} \ L_1, \ id \ { \text{addtype(id.entry, L.in)} } \]
\[ L \rightarrow id \ { \text{addtype(id.entry, L.in)} } \]

```
void D() {
    Type Type = T();
    Type Lin = Type;
    L(Lin);
}

Type T() {
    Type Type;
    if (current input == INT) {
        Type = TYPE_INT;
        advance input;
    } else if (current input == REAL) {
        Type = TYPE_REAL;
        advance input;
    } else error();
    return Type;
}

void L(Type Lin) { … }
```
Example: Descent Parsers

SDT

\[ S \rightarrow \text{while } ( \{ L1 = \text{new}(); \ L2 = \text{new}(); \ C\text{.false} = S\text{.next}; \ C\text{.true} = L2; \} \ C) \{ S_1\text{.next} = L1; \} \]

\[ S_1 \{ S\text{.code} = \text{label} \ || \ L1 \ || \ C\text{.code} \ || \ label \ || \ L2 \ || S_1\text{.code}; \} \]

Example: Descent Parsers

```
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" || L1 || Ccode
                || "label" || L2
                || 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 record 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 containing temporary values and actions for $A$, placed above $A$’s nonterminal record
  – Synthesized record for $A$, containing synthesized attributes and actions, placed below $A$’s nonterminal record

L-Attributed SDD and LL Parsers

• Consider CFG production $A \rightarrow B \ C$
  – Action record for $C$ 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

Rule: $A$

$B$

$C$

$A$ action

$A$ record

$A$ synth

$B$ record

$C$ record

$A$ synth

Attributes from $A$’s parent and $A$’s left siblings

Attributes from $B$ and $C$

Example

$S$

$next = x$

top

Example

while

Action

C
false = ?
true = ?

) Action

S
al1 = ?
next = ?

al2 = ?

stack[ top - 1 ]. next = al1;
print( "label", al2 );

L1 = new();
L2 = new();
stack[ top - 1 ]. false = snext;
stack[ top - 1 ]. true = L2;
stack[ top - 3 ]. al1 = L1;
stack[ top - 3 ]. al2 = L2;
print( "label", L1 );

Example

Action

snext = x
L1 = ?
L2 = ?

) Action

S
al1 = ?
next = ?

al2 = ?

stack[ top - 1 ]. next = al1;
print( "label", al2 );

L1 = new();
L2 = new();
stack[ top - 1 ]. false = snext;
stack[ top - 1 ]. true = L2;
stack[ top - 3 ]. al1 = L1;
stack[ top - 3 ]. al2 = L2;
print( "label", L1 );
### Example

```
stack[top].false = snext;
stack[top].true = L2;
stack[top - 3].al1 = L1;
stack[top - 3].al2 = L2;
print("label", L1);
```

```
L1 = new();
L2 = new();
stack[top - 1].false = snext;
stack[top - 1].true = L2;
stack[top - 3].al1 = L1;
stack[top - 3].al2 = L2;
print("label", L1);
```
Example

\[ \text{top} \]

<table>
<thead>
<tr>
<th>Action</th>
<th>( S_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a1 = y )</td>
<td>( \text{next} = y )</td>
</tr>
<tr>
<td>( a2 = z )</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{stack[\text{top} - 1].next = a1;} \]
\[ \text{print("label", a2);} \]

Example

\[ \text{top} \]

\[ S_1 \]

| \( \text{next} = y \) |
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 \( M_a \rightarrow \epsilon \{a'\} \), 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

```
stack :

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

val_s :

<table>
<thead>
<tr>
<th></th>
<th>Xs</th>
<th>Ys</th>
<th>Zs</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

val_i :

<table>
<thead>
<tr>
<th></th>
<th>Xi</th>
<th>Yi</th>
<th>Zi</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Example

SDT:

```
S → a A { C.i = f(A.s) } C
S → b A B { C.i = f(A.s) } C
C → c { C.s = g(C.i) }
```

Transformed SDT:

```
S → a A M_1 C
S → b A B M_2 C
M_1 → ε { val_s[top + 1] = f(val_s[top]); top = top + 1 }
M_2 → ε { val_s[top + 1] = f(val_s[top - 1]); top = top + 1 }
C → 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