Syntax-Directed Translation

Technique used to build semantic information for large structures, based on its syntax.

In a compiler, *Syntax-Directed Translation* is used for
- Constructing Abstract Syntax Tree
- Type checking
- Intermediate code generation

The Essence of Syntax-Directed Translation

The semantics (i.e., *meaning*) of the various constructs in the language is viewed as *attributes* of the corresponding grammar symbols.

Example:

```
sequence of characters 495
→ grammar symbol TOK_INT
→ meaning ≡ integer 495
→ is an attribute of TOK_INT (yyval.int_val).
```

Attributes are associated with **Terminal** as well as **Nonterminal** symbols.
An Example of Syntax-Directed Translation

\[
\begin{align*}
E & \rightarrow E \ast E \\
E & \rightarrow E + E \\
E & \rightarrow \text{id}
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow E_1 \ast E_2 \quad \{ E.\text{val} := E_1.\text{val} \ast E_2.\text{val} \} \\
E & \rightarrow E_1 + E_2 \quad \{ E.\text{val} := E_1.\text{val} + E_2.\text{val} \} \\
E & \rightarrow \text{int} \quad \{ E.\text{val} := \text{int}.\text{val} \}
\end{align*}
\]

Another Example of Syntax-Directed Translation

\[
\begin{align*}
\text{Decl} & \rightarrow \text{Type VarList} \\
\text{Type} & \rightarrow \text{integer} \\
\text{Type} & \rightarrow \text{float} \\
\text{VarList} & \rightarrow \text{id} , \text{VarList} \\
\text{VarList} & \rightarrow \text{id}
\end{align*}
\]

\[
\begin{align*}
\text{Decl} & \rightarrow \text{Type VarList} \quad \{ \text{VarList}.\text{type} := \text{Type}.\text{type} \} \\
\text{Type} & \rightarrow \text{integer} \quad \{ \text{Type}.\text{type} := \text{int} \} \\
\text{Type} & \rightarrow \text{float} \quad \{ \text{Type}.\text{type} := \text{float} \} \\
\text{VarList} & \rightarrow \text{id} , \text{VarList}_1 \quad \{ \text{VarList}_1.\text{type} := \text{VarList}.\text{type}; \\
& \quad \quad \quad \text{id}.\text{type} := \text{VarList}.\text{type} \} \\
\text{VarList} & \rightarrow \text{id} \quad \{ \text{id}.\text{type} := \text{VarList}.\text{type} \}
\end{align*}
\]
Attributes

- **Synthesized**: Attribute of LHS symbol of a grammar rule, whose value depends on attributes of RHS symbols of the grammar rule.
  - Value flows from child to parent in the parse tree.
  - Example: \textit{val} in Expression grammar

- **Inherited**: Attribute of an RHS symbol of a grammar rule, whose value depends on attributes of the LHS symbol and the other RHS symbols of the grammar rule.
  - Value flows into a node in the parse tree from parents and/or siblings.
  - Example: \textit{type} of \textit{VarList} in declaration grammar

Syntax-Directed Definition

*Actions* associated with each production in a grammar. For a production $A \rightarrow X \ Y$, actions may be of the form:

- $A.attr := f(X.attr', Y.attr'')$ for synthesized attributes
- $Y.attr := f(A.attr', X.attr'')$ for inherited attributes

If the function $f$ does not have side effects, syntax directed definitions is also called as *attribute grammars*. 
**Attributes and Definitions**

- **S-Attributed Definitions:** Where all attributes are *synthesized*.
- **L-Attributed Definitions:** Where all *inherited* attributes are such that their values depend only on
  - inherited attributes of the parent, and
  - attributes of left siblings

---

**Synthesized Attributes: An Example**

```
E    -->  E * E
E    -->  E + E
E    -->  int

E    -->  E1 * E2  {E.val := E1.val * E2.val}
E    -->  E1 + E2  {E.val := E1.val + E2.val}
E    -->  int      {E.val := int.val}
```
Information Flow for “Expression” Example

Another Example of Syntax-Directed Translation

\[
\begin{align*}
Decl & \rightarrow Type \ VarList \\
Type & \rightarrow integer \\
Type & \rightarrow float \\
VarList & \rightarrow id, VarList \\
VarList & \rightarrow id
\end{align*}
\]
Information Flow for “Type” Example

Syntax-Directed Definitions with yacc/Bison

\[
egin{align*}
E & \rightarrow E_1 \times E_2 \quad \{ E.val := E_1.val \times E_2.val \} \\
E & \rightarrow E_1 + E_2 \quad \{ E.val := E_1.val + E_2.val \} \\
E & \rightarrow \text{int} \quad \{ E.val := \text{int}.val \}
\end{align*}
\]

\[
\begin{align*}
E & : \ E \text{ \text{MULT} } E \quad \{ \$.val = \$1.val \times \$3.val \} \\
E & : \ E \text{ \text{PLUS} } E \quad \{ \$.val = \$1.val + \$3.val \} \\
E & : \ \text{INT} \quad \{ \$.val = \$1.val \}
\end{align*}
\]
Keep track of attributes of symbols while parsing.

- **Shift-reduce parsing**: Keep a stack of attributes corresponding to stack of symbols. Compute attributes of LHS symbol while performing reduction (i.e., while pushing the symbol on symbol stack)

```
E → E+E
E → E*E
E → int
```

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input Stream</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>int</td>
<td>3 * 2 + 5</td>
<td>$ 3</td>
</tr>
<tr>
<td>E</td>
<td>* 2 + 5</td>
<td>$ 3</td>
</tr>
<tr>
<td>E *</td>
<td>2 + 5</td>
<td>$ 3 ⊥</td>
</tr>
<tr>
<td>E * int</td>
<td>+ 5</td>
<td>$ 3 ⊥ 2</td>
</tr>
<tr>
<td>E</td>
<td>+ 5</td>
<td>$ 6</td>
</tr>
<tr>
<td>E +</td>
<td>5</td>
<td>$ 6 ⊥</td>
</tr>
<tr>
<td>E + int</td>
<td>$</td>
<td>$ 6 ⊥ 5</td>
</tr>
<tr>
<td>E + E</td>
<td>$</td>
<td>$ 6 ⊥ 5</td>
</tr>
<tr>
<td>E</td>
<td>$</td>
<td>$ 11</td>
</tr>
</tbody>
</table>
Inherited Attributes

\[
\begin{align*}
Ss & \rightarrow S ; Ss \\
Ss & \rightarrow \epsilon \\
B & \rightarrow \{ Ss \} \\
S & \rightarrow B \\
S & \rightarrow \text{other}
\end{align*}
\]

\[
\begin{align*}
Ss & \rightarrow S ; Ss_1 \quad \{ S.block = Ss.block; \}
\quad \{ Ss_1.block = Ss.block; \}
Ss & \rightarrow \epsilon \\
B & \rightarrow \{ Ss \} \quad \{ Ss.block = \text{child}(B.block); \}
S & \rightarrow B \quad \{ B.block = S.block; \}
S & \rightarrow \text{other} \quad \{ \text{other}.block = S.block; \}
\end{align*}
\]

Top-down Parsing

```c
parse_Ss() {
    /* production 1 */
    parse_S();
    parse_Ss();
    return;

    /* production 2 */
    return;
}
parse_B() {
    consume(OPN_B); consume(CSL_B);
    parse_Ss();
    consume(OPN_B);
}
```
Inherited Attributes and Top-down Parsing

\[
\text{parse}_S s(block) \{ \\
/* production 1 */ \\
\quad \text{parse}_S(block); \\
\quad \text{parse}_S s(block); \\
\quad \text{return}; \\
/* production 2 */ \\
\quad \text{return}; \\
\}
\]

\[
\text{parse}_B(block) \{ \\
\quad \text{consume(OPEN_BRACE);} \\
\quad \text{parse}_S s(\text{child(block)}); \\
\quad \text{consume(CLOSE_BRACE);} \\
\}
\]

Attributes and Top-down Parsing

- **Inherited**: analogous to function arguments
- **Synthesized**: analogous to return values

L-attributed definitions mean that argument to a parsing function is
- argument of the calling function, or
- return value/argument of a previously called function
Attributes and Bottom-up Parsing

- **Synthesized:** stored along with symbols on stack
- **Inherited:** ???

Inherited Attributes and Bottom-up Parsing

Inherited attributes depend on the context in which a symbol is used. For inherited attributes, we cannot assign an value to a node’s attributes unless the parent’s attributes are known. When building parse trees bottom-up, parent of a node is not known when the node is created!

**Solution:**
- Ensure that all attributes are inherited only from left siblings.
- Use “global” variables to capture inherited values,
- and introduce “marker” nonterminals to manipulate the global variables.
Inherited Attributes & Bottom-up parsing

\[
\begin{align*}
Ss & \rightarrow S ; Ss \\
Ss & \rightarrow \epsilon \\
B & \rightarrow \{ Ss \} \\
S & \rightarrow B \\
S & \rightarrow \text{other}
\end{align*}
\]

\[
\begin{align*}
B & \rightarrow \{ M_1 \ Ss \ M_2 \} \\
M_1 & \rightarrow \epsilon, \quad \{ \text{current\_block++;} \} \\
M_2 & \rightarrow \epsilon, \quad \{ \text{current\_block--;} \}
\end{align*}
\]

$M_1$ and $M_2$ are marker non-terminals.