Synthesized Attributes: An Example

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

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

Attributes

- **Synthesized Attribute**: Value of the attribute computed from the values of attributes of grammar symbols on RHS.
- **Inherited Attribute**: Value of attribute computed from values of attributes of the LHS grammar symbol.

Information Flow for Synthesized Attributes

Syntax-Directed Definition

*Actions* associated with each production in a grammar.

For a production \(A \rightarrow X Y\), actions may be of the form:

- \(A, \text{attr} := f(X, \text{attr}', Y, \text{attr}'')\) for synthesized attributes
- \(Y, \text{attr} := f(A, \text{attr}')\) for inherited attributes

If the function \(f\) does not have side effects, syntax directed definitions is also called as *attribute grammars*. 
Syntax-Directed Definitions with yacc

\[ E \rightarrow E_1 \ast E_2 \quad \{ E.val := E_1.val \ast E_2.val \} \]
\[ E \rightarrow E_1 + E_2 \quad \{ E.val := E_1.val + E_2.val \} \]
\[ E \rightarrow \text{int} \quad \{ E.val := \text{int} \} \]
\[ E : E \text{ MULT } E \quad \{ \$$.val = E1.val \ast E2.val \} \]
\[ E : E \text{ PLUS } E \quad \{ \$$.val = E1.val + E2.val \} \]
\[ E : \text{INT} \quad \{ \$$.val = E.val \} \]

Synthesized Attributes and Bottom-up Parsing

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)

Another Example of Syntax-Directed Translation

\[ \text{Decl} \rightarrow \text{Type} \text{ VarList} \]
\[ \text{Type} \rightarrow \text{integer} \]
\[ \text{Type} \rightarrow \text{float} \]
\[ \text{VarList} \rightarrow \text{id} \text{, VarList} \]
\[ \text{VarList} \rightarrow \text{id} \]

\[ \text{Decl} \rightarrow \text{Type} \text{ VarList} \quad \{ \text{VarList} \text{.} \text{type} := \text{Type} \text{.} \text{type} \} \]
\[ \text{Type} \rightarrow \text{integer} \quad \{ \text{Type} \text{.} \text{type} := \text{int} \} \]
\[ \text{Type} \rightarrow \text{float} \quad \{ \text{Type} \text{.} \text{type} := \text{float} \} \]
\[ \text{VarList} \rightarrow \text{id} \text{, VarList} \quad \{ \text{VarList} \text{.} \text{type} := \text{VarList} \text{.} \text{type} \} \]
\[ \text{VarList} \rightarrow \text{id} \quad \{ \text{id} \text{.} \text{type} := \text{VarList} \text{.} \text{type} \} \]
Abstract Syntax Tree (AST)

Represents the syntactic structure of the input program, independent of peculiarities in the grammar.

As Example:

Consider a statement of the form:

"if (m == 0) S1 else S2"

where S1 and S2 stand for some block of statements.

A possible AST for this statement is:

```
If-then-else

==
m 0

AST for S1

AST for S2
```

Construction of Abstract Syntax Trees

Typically done simultaneously with parsing

... as another instance of syntax-directed translation

... for translating concrete syntax (the parse tree) to abstract syntax (AST),

... with AST as a synthesized attribute of each grammar symbol.

Synthesized Attributes & Shift-reduce parsing

```
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>3 * 2 + $</td>
<td>$</td>
</tr>
<tr>
<td>$ int</td>
<td>* 2 + $</td>
<td>$ 3</td>
</tr>
<tr>
<td>$ E</td>
<td>* 2 + $</td>
<td>$ 3</td>
</tr>
<tr>
<td>$ E *</td>
<td>2 + $</td>
<td>$ 3 \perp</td>
</tr>
<tr>
<td>$ E * int</td>
<td>+ $</td>
<td>$ 3 \perp 2</td>
</tr>
<tr>
<td>$ E</td>
<td>* $</td>
<td>$ 6</td>
</tr>
<tr>
<td>$ E +</td>
<td>$</td>
<td>$ 6 \perp</td>
</tr>
<tr>
<td>$ E + int</td>
<td>$</td>
<td>$ 6 \perp 5</td>
</tr>
<tr>
<td>$ E + E</td>
<td>$</td>
<td>$ 6 \perp 5</td>
</tr>
<tr>
<td>$ E</td>
<td>$</td>
<td>$ 11</td>
</tr>
</tbody>
</table>

Semantic Analysis Phases of Compilation

- Build an Abstract Syntax Tree (AST) while parsing
- Decorate the AST with type information (type checking/inference)
- Generate intermediate code from AST
  - Optimize intermediate code
  - Generate final code
Inherited Attributes and Bottom-up Parsing

Inherited attributes depend on the context in which a symbol is used.

For inherited attributes, we cannot assign a 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: Introduce marker nonterminals to capture the attributes of parent.

“Inherit” attributes from siblings.