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

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

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

Syntax-Directed Definitions with yacc

\[
\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.val}\} \\
E & \rightarrow \text{E MULT E} \quad \{\$$.\text{val} = $$1.\text{val} \ast $$3.\text{val}\} \\
E & \rightarrow \text{E PLUS E} \quad \{\$$.\text{val} = $$1.\text{val} + $$3.\text{val}\} \\
E & \rightarrow \text{INT} \quad \{\$$.\text{val} = $$1.\text{val}\}
\end{align*}
\]
Another Example of Syntax-Directed Translation

<table>
<thead>
<tr>
<th>Production</th>
<th>Right-Hand Side</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decl → Type VarList</td>
<td>{VarList.type := Type.type}</td>
<td></td>
</tr>
<tr>
<td>Type → . . .</td>
<td>{Type.type := . . .}</td>
<td></td>
</tr>
<tr>
<td>VarList → id , VarList</td>
<td>{VarList.type := VarList.type; id.type := VarList.type}</td>
<td></td>
</tr>
<tr>
<td>VarList → id</td>
<td>{id.type := VarList.type}</td>
<td></td>
</tr>
</tbody>
</table>

Attributes

- **Synthesized** Attribute: Value of the attribute computed from the values of attributes of grammar symbols on RHS.
  - Example: `val` in Expression grammar
- **Inherited** Attribute: Value of attribute computed from values of attributes of the LHS grammar symbol.
  - Example: `type` of `VarList` in declaration grammar

Syntax-Directed Definition

*Actions* associated with each production in a grammar. For a production `A → 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

Synthesized Attributes: An Example

<table>
<thead>
<tr>
<th>Production</th>
<th>Right-Hand Side</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E * E</td>
<td>{E.val := E_1.val * E_2.val}</td>
<td></td>
</tr>
<tr>
<td>E → E + E</td>
<td>{E.val := E_1.val + E_2.val}</td>
<td></td>
</tr>
<tr>
<td>E → int</td>
<td>{E.val := int.val}</td>
<td></td>
</tr>
</tbody>
</table>

Information Flow for Synthesized Attributes
Another Example of Syntax-Directed Translation

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

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

Information Flow for Inherited Attributes

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

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

Synthesized Attributes and Bottom-up Parsing

Keep track of attributes of symbols while 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)

Synthesized Attributes & Shift-reduce parsing
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: 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

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

Attribute Grammars

- syntax-directed definitions without side-effects
- attribute definitions can be thought of as logical assertions rather than as things that need to be computed
  - distinction between synthesized and inherited attributes disappears

\[
E \rightarrow E_1 * E_2 \quad \{E.type = E_1.type = E_2.type\}
\]

\[
E \rightarrow E_1 + E_2 \quad \{E.type = E_1.type = E_2.type\}
\]

\[
E \rightarrow \text{int} \quad \{E.type = \text{integer}\}
\]

Attribute Grammars

An attribute grammar \( AG \) is given by \((G, V, F)\), where:

- \( G \) is a context-free grammar
- \( V \) is the set of attributes, each of which is associated with a terminal or a nonterminal
- \( F \) is the set of attribute assertions, each of which is associated with a production in the grammar
A string \( s \in L(AG) \) iff \( s \in L(G) \) and the attribute assertions hold for production used to derive \( s \), i.e., \( \exists \) a parse tree for \( s \) w.r.t. \( G \) where assertions associated with each edge in the parse tree are satisfied.

**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

**Abstract Syntax Tree (AST)**

- Represents syntactic structure of a program
- Abstracts out irrelevant grammar details

An AST for the statement:

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

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.
Actions and AST

\[
E \rightarrow E_1 + T \quad \{ E \text{.ast} = \text{new BinaryExpr(OP\_PLUS, } E_1 \text{.ast}, T \text{.ast); } \} \\
E \rightarrow T \quad \{ E \text{.ast} = T \text{.ast; } \} \\
F \rightarrow (E) \quad \{ F \text{.ast} = E \text{.ast; } \} \\
F \rightarrow \text{int} \quad \{ F \text{.ast} = \text{new IntValNode(int.val); } \}
\]

Actions and AST: Another Example

\[
S \rightarrow \text{if } E \text{ S}_1 \text{ else } S_2 \quad \{ S \text{.ast} = \text{new IfStmtNode(E.ast, } S_1 \text{.ast, } S_2 \text{.ast); } \} \\
S \rightarrow \text{return } E \quad \{ S \text{.ast} = \text{new ReturnNode(E.ast)} \}
\]