Compilation

Source Program

Lexical Analysis

Parsing

Semantic Analysis (e.g., type checking)

Intermediate code Generation

Code Optimization(s)

Final code generation

Target Program
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 (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 \(\equiv\) integer 495
- is an attribute of `TOK_INT(yy1val.int_val)`.

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

An example of syntax-directed translation is shown in the diagram. The production rules for the grammar are:

- $E \rightarrow E \ast E$
- $E \rightarrow E + E$
- $E \rightarrow \text{id}$
- $E \rightarrow E_1 \ast E_2 \{ E.val := E_1.val \ast E_2.val \}$
- $E \rightarrow E_1 + E_2 \{ E.val := E_1.val + E_2.val \}$
- $E \rightarrow \text{int} \{ E.val := \text{int}.val \}$
Syntax-Directed Definitions with yacc

\[ E \rightarrow E_1 \ast E_2 \{ E.val := E_1.val \ast E_2.val \} \]
\[ E \rightarrow E_1 + E_2 \{ E.val := E_1.val + E_2.val \} \]
\[ E \rightarrow \text{int} \{ E.val := \text{int.val} \} \]

\[ E : E \ \text{MULT} \ E \{ $$val = 1.val \ast 3.val \} \]
\[ E : E \ \text{PLUS} \ E \{ $$val = 1.val + 3.val \} \]
\[ E : \text{INT} \{ $$val = 1.val \} \]
Another Example of Syntax-Directed Translation

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

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

\[
\begin{align*}
\{ \text{VarList.type := Type.type} \} \\
\{ \text{Type.type := \ldots} \} \rightarrow \text{TYPE_INT}
\end{align*}
\]

\[
\begin{align*}
\{ \text{VarList}_1.type := \text{VarList.type}\; \text{; id.type := VarList.type} \}
\end{align*}
\]
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 $A \rightarrow X Y$, actions may be of the form:

- $A\text{.attr} := f(X\text{.attr}', Y\text{.attr}'')$ for synthesized attributes
- $Y\text{.attr} := f(A\text{.attr}', X\text{.attr}'')$ for inherited attributes
Synthesized Attributes: An Example

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

\[
\begin{align*}
E & \rightarrow E_1 \ast E_2 \quad \{E.val := E_1.val \ast E_2.val\} \\
E & \rightarrow E_1 + E_2 \quad \{E.val := E_1.val + E_2.val\} \\
E & \rightarrow \text{int} \quad \{E.val := \text{int}.val\}
\end{align*}
\]
Information Flow for Synthesized Attributes
Another Example of Syntax-Directed Translation

```
Decl → Type VarList
Type → integer
Type → float
VarList → id , VarList
VarList → id

Decl → Type VarList
Type → integer
Type → float
VarList → id , VarList
VarList → id
```

```
{VarList.type := Type.type}
{Type.type := int}
{Type.type := float}
{VarList₁.type := VarList.type;
 id.type := VarList.type}
{id.type := VarList.type}
```
Information Flow for Inherited Attributes

Decl
Type
VarList
integer id
id
,
integer x, y
int
int
int
int
int, int
int

VarList

integer
int id
,
VarList
int
id
int

integer x, y
**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
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 and 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 \perp</td>
</tr>
<tr>
<td>$ E</td>
<td>+ 5 $</td>
<td>$ 6</td>
</tr>
<tr>
<td>$ E</td>
<td>5 $</td>
<td>$ 6 \perp</td>
</tr>
<tr>
<td>$ E</td>
<td>+ int</td>
<td>$ 6 \perp 5</td>
</tr>
<tr>
<td>$ E</td>
<td>E + int</td>
<td>$ 6 \perp 5</td>
</tr>
<tr>
<td>$ E</td>
<td>E + E</td>
<td>$ 6 \perp 5</td>
</tr>
<tr>
<td>$ E</td>
<td>int</td>
<td>$ 11</td>
</tr>
</tbody>
</table>
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

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

\[
\begin{align*}
B & \rightarrow \{ M_1 \ Ss \ M_2 \} \\
M_1 & \rightarrow \epsilon \\
M_2 & \rightarrow \epsilon
\end{align*}
\]

\{ \text{current\_block}++; \} \quad \{ \text{current\_block}--; \}
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

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

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

\[
E \rightarrow \text{int} \quad \{E.\text{type} = \text{integer}\}
\]
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
AST for S1

0
AST for S2
```

Abstracted version of Parse Tree
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.
Abstract Syntax Trees

Parse Tree

AST

E
+ T
* F
int
int

Binary_Exp
+ Int_Exp Binary_Exp
* Int_Exp Int_Exp
+( *( 2, 3))5,

5 + 2 * 3

5, * (2, 3)
Actions and AST

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

\[ E \rightarrow T \quad \{ E.ast = T.ast; \} \]

\[ F \rightarrow (E) \quad \{ F.ast = E.ast; \} \]

\[ F \rightarrow \text{int} \]
\[
\{ F.ast = \text{new IntValNode(int.val);} \} \]
$S \rightarrow \text{if } E \text{ else } S_2$

\{ 
  S.ast = new IfStmtNode(E.ast, 
                         S_1.ast, S_2.ast); 
\}

$S \rightarrow \text{return } E$

\{ 
  S.ast = new ReturnNode(E.ast) 
\}
stmt → simple_stmt  Tok_SEMI
  | compound_stmt  opt_Semi

compound_stmt → Tok_LBRACE  stmt_list  Tok_RBRACE

opt_Semi → /empty/  |  Tok_SEMI

int a[5] = {1, 2, 3, 4, 5};