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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 \ast E_2$</td>
<td>$E.val := E_1.val \ast E_2.val$</td>
<td>{ $$.val = $1.val \ast $3.val }</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E.val := E_1.val + E_2.val$</td>
<td>{ $$.val = $1.val + $3.val }</td>
</tr>
<tr>
<td>$E \rightarrow \text{int}$</td>
<td>$E.val := \text{int}.val$</td>
<td>{ $$.val = $1.val }</td>
</tr>
</tbody>
</table>

$E : E \text{ MULT } E$  
$E : E \text{ PLUS } E$  
$E : \text{INT}$
Another Example of Syntax-Directed Translation

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

```
Decl → Type VarList
Type → "int" Type.type := Type.type
VarList → id, VarList1
{VarList1.type := VarList.type; id.type := VarList.type}
```

```
VarList → "float" Type.type := \ldots
{Type.type := \ldots};
```

```
VarList → id
{id.type := VarList.type}
```
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
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

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

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

\[
\begin{align*}
\text{Decl} & \rightarrow \text{Type} \ VarList \\
\text{Type} & \rightarrow \text{integer} \\
\text{Type} & \rightarrow \text{float} \\
\text{VarList} & \rightarrow \text{id} , \ VarList \\
\text{VarList} & \rightarrow \text{id}
\end{align*}
\]
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 (\textit{i.e.}, while pushing the symbol on symbol stack)
### Synthesized Attributes and Bottom-Up Parsing

### Grammar Rules

- $E \rightarrow E + E$
- $E \rightarrow E \ast E$
- $E \rightarrow \text{int}$

### Parsing Table

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input Stream</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>3 \ast 2 + 5$</td>
<td>$_$</td>
</tr>
<tr>
<td>$\text{int}$</td>
<td>* 2 + 5 $</td>
<td>$3_</td>
</tr>
<tr>
<td>$E$</td>
<td>* 2 + 5 $</td>
<td>$3_</td>
</tr>
<tr>
<td>$E \ast$</td>
<td>2 + 5 $</td>
<td>$3\bot 2</td>
</tr>
<tr>
<td>$E \ast \text{int}$</td>
<td>+ 5 $</td>
<td>$3\bot 2</td>
</tr>
<tr>
<td>$E$</td>
<td>+ 5 $</td>
<td>$6_</td>
</tr>
<tr>
<td>$E +$</td>
<td>5 $</td>
<td>$6\bot</td>
</tr>
<tr>
<td>$E + \text{int}$</td>
<td>$</td>
<td>$6\bot 5</td>
</tr>
<tr>
<td>$E + E$</td>
<td>$</td>
<td>$6\bot 5</td>
</tr>
<tr>
<td>$E$</td>
<td>$</td>
<td>$11</td>
</tr>
</tbody>
</table>

The table shows the stack, input stream, and attributes for each parsing step. The stack and input stream are updated according to the grammar rules. The attributes are calculated based on the parsed elements, with markers indicating the end of a phrase or attribute.
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

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

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

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

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

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

\[
E \rightarrow \text{int} \quad \{ E.type = \text{integer} \}
\]
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
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
   |
   T
   |
   *
   |
   F
   |
   int

Binary_Exp
   +
   |
   Int_Exp
   |
   Binary_Exp
   |
   *
   |
   Int_Exp
   |
   Int_Exp

+( *( 2, 3))5,
Actions and AST

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

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

\[
F \rightarrow (E) \ \{F.ast = E.ast;\} \\
F \rightarrow \text{int} \\
\{F.ast = new IntValNode(int.val); \}
\]
Actions and AST: Another Example

\[ S \rightarrow \begin{cases} \text{if } E \ S_1 \ 	ext{else } S_2 \\ S.\ast = \text{new IfStmtNode}(E.\ast, \\ \quad S_1.\ast, S_2.\ast) \\ \end{cases} \]

\[ S \rightarrow \begin{cases} \text{return } E \\ S.\ast = \text{new ReturnNode}(E.\ast) \\ \end{cases} \]