Syntax-Directed Translation
Part I
Chapter 5

The Structure of our Compiler
Revisited

Character stream → Lexical analyzer → Token stream → Syntax-directed translator → Java byte code
Lex specification Yacc specification with semantic rules JVM specification

Syntax-Directed Definitions

• A syntax-directed definition (or attribute grammar) binds a set of semantic rules to productions
• Terminals and nonterminals have attributes holding values set by the semantic rules
• A depth-first traversal algorithm traverses the parse tree thereby executing semantic rules to assign attribute values
• After the traversal is complete the attributes contain the translated form of the input
Example Attribute Grammar

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E \cdot n$</td>
<td>$print(E, val)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.val := E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val := T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 \cdot F$</td>
<td>$T.val := T_1.val \cdot F.val$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.val := F.val$</td>
</tr>
<tr>
<td>$F \rightarrow ( E )$</td>
<td>$F.val := E.val$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F.val := \text{digit}.lexval$</td>
</tr>
</tbody>
</table>

Note: all attributes in this example are of the synthesized type.

Example Annotated Parse Tree

Annotating a Parse Tree With Depth-First Traversals

```plaintext
procedure visit(n : node);
begin
  for each child m of n, from left to right do
    visit(m);
  evaluate semantic rules at node n
end
```
Depth-First Traversals (Example)

Attributes

- Attribute values may represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Memory locations, such as a frame index of a local variable or function argument
  - A data type for type checking of expressions
  - Scoping information for local declarations
  - Intermediate program representations

Synthesized Versus Inherited Attributes

- Given a production
  \[ A \rightarrow \alpha \]
  then each semantic rule is of the form
  \[ b := f(c_1, c_2, \ldots, c_k) \]
  where \( f \) is a function and \( c_i \) are attributes of \( A \) and \( \alpha \), and either
  - \( b \) is a synthesized attribute of \( A \)
  - \( b \) is an inherited attribute of one of the grammar symbols in \( \alpha \)
Synthesized Versus Inherited Attributes (cont’d)

Production | Semantic Rule
--- | ---
\[ D \rightarrow T \cdot L \] | inherited
\[ T \rightarrow \text{int} \] | 
\[ ... \] | 
\[ L \rightarrow \text{id} \] | synthesized

Production | Semantic Rule
--- | ---
\[ D \rightarrow T \cdot L \] | inherited
\[ T \rightarrow \text{int} \] | Type = 'integer'
\[ ... \] | synthesized
\[ L \rightarrow \text{id} \] | ... = L.in

S-Attributed Definitions

• A syntax-directed definition that uses synthesized attributes exclusively is called an S-attributed definition (or S-attributed grammar)
• A parse tree of an S-attributed definition is annotated with a single bottom-up traversal
• Yacc/Bison only support S-attributed definitions

Example Attribute Grammar in Yacc

```yacc
%%
L : E ' \n' { printf("%d\n", $1); }
E : E '+ ' T { $0 = $1 + $3; }
| T { $0 = $1; }
T : T '*' F { $0 = $1 * $3; }
| F { $0 = $1; }
F : '(' E ')' { $0 = $2; }
| DIGIT { $0 = $1; }

Synthesized attribute of parent node F
```
### Bottom-up Evaluation of S-Attributed Definitions in Yacc

<table>
<thead>
<tr>
<th>Stack</th>
<th>Val</th>
<th>Input</th>
<th>Action</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>3</td>
<td>$3*5+4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$F$</td>
<td>3</td>
<td>$*5+4n$</td>
<td>reduce $F \rightarrow$ digit</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$T$</td>
<td>3</td>
<td>$5+4n$</td>
<td>reduce $T \rightarrow F$</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$T$</td>
<td>3</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$T \cdot 5$</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$F \cdot 5$</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$F$</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$T \cdot 5$</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$T \cdot 5$</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$T \cdot 5$</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
<tr>
<td>$E \cdot 4$</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = S1$</td>
</tr>
</tbody>
</table>

### Example Attribute Grammar with Synthesized+Inherited Attributes

#### Production Semantics

- $D \rightarrow T L$
  
  $L.in := T.type$

- $T \rightarrow \text{int}$
  
  $T.type := \text{integer}'$

- $T \rightarrow \text{real}$
  
  $T.type := \text{real}'$

- $L \rightarrow L_1, id$
  
  $L.in := L.in; \text{addtype}(id.entry, L.in)$

- $L \rightarrow id$
  
  $\text{addtype}(id.entry, L.in)$

**Synthesized:** $T.type, id.entry$

**Inherited:** $L.in$

### Acyclic Dependency Graphs for Parse Trees

- $A \rightarrow XY$
  
  $A.a := f(X.x, Y.y)$

- $X \rightarrow A_a$
  
  $X.x := f(A_a, Y.y)$

- $Y \rightarrow A_a$
  
  $Y.y := f(A_a, X.x)$

**Direction of value dependence**
Dependency Graphs with Cycles?

- Edges in the dependency graph determine the evaluation order for attribute values
- Dependency graphs cannot be cyclic

\[
A.a := f(X.x) \\
X.x := f(Y.y) \\
Y.y := f(A.a)
\]

Error: cyclic dependence

Example Annotated Parse Tree

\[
D \\
T.type = \text{"real"} \\
L.in = \text{"real"} \\
\text{real} \\
L.in = \text{"real"}, \text{id}_1.entry \\
L.in = \text{"real"}, \text{id}_2.entry \\
id_1.entry
\]

Example Annotated Parse Tree with Dependency Graph

\[
D \\
T.type = \text{"real"} \\
L.in = \text{"real"} \\
\text{real} \\
L.in = \text{"real"}, \text{id}_1.entry \\
L.in = \text{"real"}, \text{id}_2.entry \\
id_1.entry
\]
Evaluation Order

• A topological sort of a directed acyclic graph (DAG) is any ordering $m_1, m_2, \ldots, m_n$ of the nodes of the graph, such that if $m_i \rightarrow m_j$ is an edge, then $m_i$ appears before $m_j$
• Any topological sort of a dependency graph gives a valid evaluation order of the semantic rules

Example Parse Tree with Topologically Sorted Actions

Topological sort:
1. Get $id_1.entry$
2. Get $id_2.entry$
3. Get $id_3.entry$
4. $T_1.type =$ 'real'
5. $L_1.in =$ 'real'
6. $L_1.in =$ $T_1.type$
7. $L_2.in =$ $L_1.in$
8. $L_2.in =$ $L_1.in$
9. $L_2.in =$ $L_1.in$
10. $L_2.in =$ $L_1.in$

Evaluation Methods

• Parse-tree methods determine an evaluation order from a topological sort of the dependence graph constructed from the parse tree for each input
• Rule-base methods the evaluation order is predetermined from the semantic rules
• Oblivious methods the evaluation order is fixed and semantic rules must be (re)written to support the evaluation order (for example S-attributed definitions)
L-Attributed Definitions

- The example parse tree on slide 18 is traversed “in order”, because the direction of the edges of inherited attributes in the dependency graph point top-down and from left to right
- More precisely, a syntax-directed definition is \( L \)-attributed if each inherited attribute of \( X_j \) on the right side of \( A \rightarrow X_1 X_2 \ldots X_n \) depends only on
  1. the attributes of the symbols \( X_1, X_2, \ldots, X_{j-1} \)
  2. the inherited attributes of \( A \)

Shown: dependences of inherited attributes

L-Attributed Definitions (cont’d)

- \( L \)-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

\[
A \rightarrow X \ Y \\
X_i := A_i \\
Y_i := X_s \\
A_s := Y_s
\]

- Note: every \( S \)-attributed syntax-directed definition is also \( L \)-attributed

Using Translation Schemes for L-Attributed Definitions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D \rightarrow TL )</td>
<td>( L.in := T.type )</td>
</tr>
<tr>
<td>( T \rightarrow \text{int} )</td>
<td>( T.type := \text{'integer'} )</td>
</tr>
<tr>
<td>( T \rightarrow \text{real} )</td>
<td>( T.type := \text{'real'} )</td>
</tr>
<tr>
<td>( L \rightarrow L_s, \text{id} )</td>
<td>( L_i.in := \text{addtype}(id.entry, L.in) )</td>
</tr>
<tr>
<td>( L \rightarrow \text{id} )</td>
<td>( \text{addtype}(id.entry, L.in) )</td>
</tr>
</tbody>
</table>

Translation Scheme

\[
D \rightarrow T \ (L.in := T.type ) \ L \\
T \rightarrow \text{int} \ (T.type := \text{'integer'}) \\
T \rightarrow \text{real} \ (T.type := \text{'real'}) \\
L \rightarrow \{ L_i.in := L.in \} L_s, \text{id} \ { \text{addtype}(id.entry, L.in) } \\
L \rightarrow \text{id} \ (\text{addtype}(id.entry, L.in) )
\]
Implementing L-Attributed Definitions in Top-Down Parsers

Attributes in L-attributed definitions implemented in translation schemes are passed as arguments to procedures (synthesized) or returned (inherited)

\[ D \rightarrow T \{ L.in := T.type \} L \]
\[ T \rightarrow \text{int} \{ T.type := 'integer' \} \]
\[ T \rightarrow \text{real} \{ T.type := 'real' \} \]
\[ \text{void D()} \{ \text{Type Ttype = T(); Type Lin = Ttype; L(Lin);} \} \]
\[ \text{Type T()} \{ \text{Type Ttype; if (lookahead == INT) \{ Ttype = TYPE_INT; match(INT); \} else if (lookahead == REAL) \{ Ttype = TYPE_REAL; match(REAL); \} else error(); \text{return}(Ttype) \} \]
\[ \text{void L(Type Lin) \{ \}} \]

Attributes in L-attributed definitions implemented in translation schemes are passed as arguments to procedures (synthesized) or returned (inherited)

Implementing L-Attributed Definitions in Bottom-Up Parsers

- More difficult and also requires rewriting L-attributed definitions into translation schemes
- Insert marker nonterminals to remove embedded actions from translation schemes, that is
  \[ A \rightarrow X \{ \text{actions} \} Y \]
  is rewritten with marker nonterminal \( N \) into
  \[ A \rightarrow XNY \]
  \[ N \rightarrow \epsilon \{ \text{actions} \} \]
- Problem: inserting a marker nonterminal may introduce a conflict in the parse table

Emulating the Evaluation of L-Attributed Definitions in Yacc

\{( %
D \rightarrow T \{ L.in := T.type \} L
T \rightarrow \text{int} \{ T.type := 'integer' \}
T \rightarrow \text{real} \{ T.type := 'real' \}
L \rightarrow \{ L_1.in := Lin \} \{ \text{addtype(id.entry, Lin)} \}
L \rightarrow \text{id} \{ \text{addtype(id.entry, Lin)} \}
\}

%{ 
D \text{ Lin: /* global variable */} 
Lin 
T : \text{Ts L} 
T : \text{INT} \{ $5 = \text{TYPE_INT}; \}
| \text{REAL} \{ $5 = \text{TYPE_REAL}; \}
| L : \text{'} ID \{ \text{addtype($3, Lin)} \}
| \text{ID} \{ \text{addtype($1, Lin)} \}
\}

%%
Rewriting a Grammar to Avoid Inherited Attributes

Production

D → L ; T
T → int
T → real
L → L , id
L → id

Semantic Rule

addtype(id.entry, L.type)
addtype(id.entry, L.type)

T.type := 'integer'
T.type := 'real'

L.type := T.type