The Structure of our Compiler Revisited

Lexical analyzer

Syntax-directed translator

Java bytecode

Lex specification

Yace specification with semantic rules

JVM specification

Character stream → Lexical analyzer

Token stream → Syntax-directed translator
Syntax-Directed Definitions

- A syntax-directed definition (or attribute grammar) binds a set of semantic rules to productions
- Terminals and nonterminals have attributes
- A depth-first traversal algorithm is used to compute the values of the attributes in the parse tree using the semantic rules
- After the traversal is completed, 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>$L \rightarrow E \ n$</td>
<td>$\text{print}(E.\text{val})$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.\text{val} := E_1.\text{val} + T.\text{val}$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.\text{val} := T.\text{val}$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$T.\text{val} := T_1.\text{val} * F.\text{val}$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.\text{val} := F.\text{val}$</td>
</tr>
<tr>
<td>$F \rightarrow (\ E\ )$</td>
<td>$F.\text{val} := E.\text{val}$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F.\text{val} := \text{digit.\text{lexval}}$</td>
</tr>
</tbody>
</table>

Note: all attributes in this example are of the synthesized type
Example Attribute Grammar in Yacc

```yacc
%token DIGIT

L : E '\n' { printf("%d\n", $1); }
| ;;

E : E '+' T { $$ = $1 + $3; }
| T { $$ = $1; }
| ;;

T : T '*' F { $$ = $1 * $3; }
| F { $$ = $1; }
| ;;

F : '(' E ')' { $$ = $2; }
| DIGIT { $$ = $1; }
| ;;

%%
```

Example Annotated Parse Tree

```

L

E.val = 16
```

```

E.val = 14
```

```

E.val = 9
```

```

T.val = 5
```

```

T.val = 9
```

```

F.val = 5
```

```

F.val = 9
```

```

9 + 5 + 2 n
```

Note: all attributes in this example are of the synthesized type
Annotating a Parse Tree With Depth-First Traversals

\begin{quote}
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
\end{quote}

Depth-First Traversals (Example)

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

• Attribute values can 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

\[ D \rightarrow T \ L \]
\[ T \rightarrow \text{int} \]
\[ \ldots \]
\[ L \rightarrow \text{id} \]

Semantic Rule

\[ L . \text{in} := T . \text{type} \]
\[ T . \text{type} := \text{‘integer’} \]
\[ \ldots := L . \text{in} \]

S-Attributed Definitions

- A syntax-directed definition that uses synthesized attributes exclusively is called an \textit{S-attributed definition} (or \textit{S-attributed grammar})
- A parse tree of an S-attributed definition can be annotated with a simple bottom-up traversal
- Yacc only supports S-attributed definitions
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>$</td>
<td>_</td>
<td>3*5+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ 3</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce F → digit</td>
<td>$5 = $1</td>
</tr>
<tr>
<td>$ F</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce T → $F$</td>
<td>$5 = $1</td>
</tr>
<tr>
<td>$ T</td>
<td>3</td>
<td>*5+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ T *</td>
<td>3</td>
<td>5+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ T * 5</td>
<td>3</td>
<td>+4n$</td>
<td>reduce F → digit</td>
<td>$5 = $1</td>
</tr>
<tr>
<td>$ T * F</td>
<td>3</td>
<td>+4n$</td>
<td>reduce T → T * F</td>
<td>$5 = $1 * $3</td>
</tr>
<tr>
<td>$ T</td>
<td>15</td>
<td>+4n$</td>
<td>reduce E → T</td>
<td>$5 = $1</td>
</tr>
<tr>
<td>$ E</td>
<td>15</td>
<td>+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ E +</td>
<td>15</td>
<td>4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ E + 4</td>
<td>15</td>
<td>n$</td>
<td>reduce F → digit</td>
<td>$5 = $1</td>
</tr>
<tr>
<td>$ E + F</td>
<td>15</td>
<td>n$</td>
<td>reduce T → $F$</td>
<td>$5 = $1</td>
</tr>
<tr>
<td>$ E + T</td>
<td>15</td>
<td>n$</td>
<td>reduce E → E + T</td>
<td>$5 = $1 + $3</td>
</tr>
<tr>
<td>$ E</td>
<td>19</td>
<td>n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ E n</td>
<td>19</td>
<td>$</td>
<td>reduce L → E n</td>
<td>print $1</td>
</tr>
<tr>
<td>$ L</td>
<td>19</td>
<td>$</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

Example Attribute Grammar with Synthesized+Inherited Attributes

Production |
---|
D → T L
T → int
T → real
L → L₁, id
L → id

Semantic Rule

L.in := T.type
T.type := ‘integer’
T.type := ‘real’
L₁.in := L.in; addtype(id.entry, L.in)
addtype(id.entry, L.in)

Synthesized: T.type, id.entry
Inherited: L.in
Acyclic Dependency Graphs for Parse Trees

\[
A \rightarrow X Y
\]

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

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

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

Dependency Graphs with Cycles?

- Edges in the dependence graph show 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

Example Annotated Parse Tree with Dependency Graph
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 for the semantic rules

Example Parse Tree with Topologically Sorted Actions

Topological sort:
1. Get $\text{id}_1$.entry
2. Get $\text{id}_2$.entry
3. Get $\text{id}_3$.entry
4. $T_1$.type=‘real’
5. $L_1$.in=$T_1$.type
6. $\text{addtype}(\text{id}_3$.entry, $L_1$.in)
7. $L_2$.in
8. $\text{addtype}(\text{id}_2$.entry, $L_2$.in)
9. $L_3$.in=$L_2$.in
10. $\text{addtype}(\text{id}_1$.entry, $L_3$.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 dependence 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

\[ A \rightarrow XY \]

Using Translation Schemes for L-Attributed Definitions

Production | Semantic Rule
--- | ---
\( D \rightarrow TL \) | \( L.\text{in} := T.\text{type} \)
\( T \rightarrow \text{int} \) | \( T.\text{type} := \text{‘integer’} \)
\( T \rightarrow \text{real} \) | \( T.\text{type} := \text{‘real’} \)
\( L \rightarrow L_1, \text{id} \) | \( L_1.\text{in} := L.\text{in}; \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \)
\( L \rightarrow \text{id} \) | \( \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \)

Translation Scheme

\[ D \rightarrow T \{ L.\text{in} := T.\text{type} \} L \]
\[ T \rightarrow \text{int} \{ T.\text{type} := \text{‘integer’} \} \]
\[ T \rightarrow \text{real} \{ T.\text{type} := \text{‘real’} \} \]
\[ L \rightarrow \{ L_1.\text{in} := L.\text{in} \} L_1, \text{id} \{ \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \} \]
\[ L \rightarrow \text{id} \{ \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \} \]
Implementing L-Attributed Definitions in Top-Down Parsers

L-attributed definitions are implemented in translation schemes first:

\[ D \rightarrow T \ (L.in := T.type \ } L \]
\[ T \rightarrow \text{int} \ (T.type := 'integer' \} \]
\[ T \rightarrow \text{real} \ (T.type := 'real' \} \]

```c
void D()
{
    Type Ttype = T();
    Type Lin = Ttype;
    L(Lin);
}

Type T()
{
    Type Ttype;
    if (lookahead == INT)
    {
        Ttype = TYPE_INT;
        match(INT);
    } else if (lookahead == REAL)
    {
        Ttype = TYPE_REAL;
        match(REAL);
    } else error();
    return Ttype;
}

void L(Type Lin)
{
    ...
}
```

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
  \[ A \rightarrow X N Y \]
  \[ N \rightarrow \varepsilon \ { \text{actions} } \]
- Problem: inserting a marker nonterminal may introduce a conflict in the parse table
Emulating the Evaluation of L-Attributed Definitions in Yacc

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

Rewriting a Grammar to Avoid Inherited Attributes

\[
D \rightarrow L : T \\
T \rightarrow \text{int} \\
T \rightarrow \text{real} \\
L \rightarrow L_1, \text{id} \\
L \rightarrow \text{id} \\
D \rightarrow \text{id} L \\
T \rightarrow \text{int} \\
T \rightarrow \text{real} \\
L \rightarrow \text{id} L_1 \\
L \rightarrow \text{id} \\
L \rightarrow : T
\]

Semantic Rule

\[
\text{addtype(id.entry, L.type)} \text{ \quad} T.\text{type} := \text{‘integer’} \\
\text{addtype(id.entry, L.type)} \text{ \quad} T.\text{type} := \text{‘real’} \\
\text{addtype(id.entry, L.type)} \text{ \quad} L.\text{type} := T.\text{type}
\]