Programming Languages

Tevfik Koşar

Lecture - IX
February 14th, 2006

Roadmap

• Semantic Analysis
  - Role of Semantic Analysis
  - Static vs Dynamic Analysis
  - Attribute Grammars
  - Evaluating Attributes
    • Decoration of Parse Trees
    • Synthesized and Inherited Attributes
Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are
  - semantic analysis
  - (intermediate) code generation
- The principal job of the semantic analyzer is to enforce static semantic rules
  - constructs a syntax tree (usually first)
  - information gathered is needed by the code generator

Static vs Dynamic Semantics

- Static Semantics
  - At compile time
    - Type checking

- Dynamic Semantics
  - At runtime
    - Division by zero
    - Out-of-bound indexing an array
Role of Semantic Analysis

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved.
- One approach interleaves construction of a syntax tree with parsing (no explicit parse tree), and then follows with separate, sequential phases for semantic analysis and code generation.

Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse or syntax tree.
- ATTRIBUTE GRAMMARS provide a formal framework for decorating such a tree.
Attribute Grammars

• We'll start with decoration of parse trees, then consider syntax trees
• Consider the following LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity:

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T * F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow - F \\
F & \rightarrow (E) \\
F & \rightarrow \text{const}
\end{align*}
\]

• This says nothing about what the program MEANS
Attribute Grammars

• We can turn this into an attribute grammar as follows (similar to Figure 4.1):

\[
\begin{align*}
E & \rightarrow E + T \quad E_1.val = E_2.val + T.val \\
E & \rightarrow E - T \quad E_1.val = E_2.val - T.val \\
E & \rightarrow T \quad E.val = T.val \\
T & \rightarrow T * F \quad T_1.val = T_2.val * F.val \\
T & \rightarrow T / F \quad T_1.val = T_2.val / F.val \\
T & \rightarrow F \quad T.val = F.val \\
F & \rightarrow - F \quad F_1.val = - F_2.val \\
F & \rightarrow (E) \quad F.val = E.val \\
F & \rightarrow \text{const} \quad F.val = C.val
\end{align*}
\]

Attribute Grammars

• The attribute grammar serves to define the semantics of the input program

• Attribute rules are best thought of as definitions, not assignments

• They are not necessarily meant to be evaluated at any particular time, or in any particular order, though they do define their left-hand side in terms of the right-hand side
Evaluating Attributes

• The process of evaluating attributes is called annotation, or **DECORATION**, of the parse tree [see Figure 4.2 for \((1+3)*2\)]
  – When a parse tree under this grammar is fully decorated, the value of the expression will be in the `val` attribute of the root

• The code fragments for the rules are called **SEMANTIC FUNCTIONS**
  – Strictly speaking, they are cast as functions, e.g.,
    \[ E_1.val = \text{sum}(E_2.val, T.val), \] cf., Figure 4.1

---

**Figure 4.2: Decoration of a parse tree for \((1 + 3) * 2\).** The `val` attributes of symbols are shown in boxes. Curving arrows represent the attribute flow, which is strictly upward in this case.
Evaluating Attributes

- This is a very simple attribute grammar:
  - Each symbol has at most one attribute
    - the punctuation marks have no attributes
- These attributes are all so-called SYNTHESIZED attributes:
  - They are calculated only from the attributes of things below them in the parse tree

Evaluating Attributes

- In general, we are allowed both synthesized and INHERITED attributes:
  - Inherited attributes may depend on things above or to the side of them in the parse tree
  - Tokens have only synthesized attributes, initialized by the scanner (name of an identifier, value of a constant, etc.).
  - Inherited attributes of the start symbol constitute run-time parameters of the compiler
Evaluating Attributes

- The grammar above is called **S-ATTRIBUTED** because it uses only synthesized attributes.
- Its **ATTRIBUTE FLOW** (attribute dependence graph) is purely bottom-up
  - It is SLR(1), but not LL(1)
- An equivalent LL(1) grammar requires inherited attributes:

LL(1) Grammar

- Example 1:

  \[
  \text{expr} \rightarrow \text{const} \quad \text{expr\_tail} \\
  \text{expr\_tail} \rightarrow \quad \text{- const} \quad \text{expr\_tail} \mid \varepsilon
  \]

Create the parse tree for **9 - 4 - 3** and evaluate it.
LL(1) Grammar

• Example 2:

\[
\begin{align*}
E & \rightarrow T \ TT \\
TT_1 & \rightarrow + \ T \ TT_2 \\
TT_1 & \rightarrow - \ T \ TT_1 \\
TT & \rightarrow \varepsilon \\
T & \rightarrow F \ FT \\
FT_1 & \rightarrow * \ F \ FT_2 \\
FT_1 & \rightarrow / \ F \ FT_2 \\
FT & \rightarrow \varepsilon \\
F_1 & \rightarrow - \ F_2 \\
F & \rightarrow ( \ E ) \\
F & \rightarrow \text{const}
\end{align*}
\]

Evaluating Attributes – Example

• Attribute grammar :

\[
\begin{align*}
\text{E} & \rightarrow T \ TT \quad \text{E.val} = \text{TT.val} \\
& \quad \quad \text{TT.st} = \text{T.val} \\
\text{TT}_1 & \rightarrow + \ T \ TT_2 \quad \text{TT}_1 \cdot v = \text{TT}_2 \cdot val \\
& \quad \quad \text{TT}_2 \cdot st = \text{TT}_1 \cdot st + \text{T.val} \\
\text{TT}_1 & \rightarrow - \ T \ TT_1 \quad \text{TT}_1 \cdot val = \text{TT}_2 \cdot val \\
& \quad \quad \text{TT}_2 \cdot st = \text{TT}_1 \cdot st - \text{T.val} \\
\text{TT} & \rightarrow \varepsilon \quad \text{TT.val} = \text{TT.st} \\
\text{T} & \rightarrow F \ FT \quad \text{T.val} = \text{FT.val} \\
& \quad \quad \text{FT.st} = \text{F.val}
\end{align*}
\]
Evaluating Attributes—Example

• Attribute grammar:

\[
\begin{align*}
\text{FT}_1 & \rightarrow \ast \text{ F } \text{ FT}_2 \\
\text{FT}_1 & \rightarrow \div \text{ F } \text{ FT}_2 \\
\text{FT} & \rightarrow \epsilon \\
\text{F}_1 & \rightarrow - \text{ F}_2 \\
\text{F} & \rightarrow ( \text{ E } ) \\
\text{F} & \rightarrow \text{ const} \\
\end{align*}
\]

\[
\begin{align*}
\text{FT}_1 & .val = \text{FT}_2 .val \\
\text{FT}_2 .st & = \text{FT}_1 .st \ast \text{ F }.val \\
\text{FT}_1 .val & = \text{FT}_2 .val \\
\text{FT}_2 .st & = \text{FT}_1 .st \div \text{ F }.val \\
\text{FT}.val & = \text{FT}.st \\
\text{F}_1 .val & = - \text{F}_2 .val \\
\text{F}.val & = \text{E}.val \\
\text{F}.val & = \text{C}.val \\
\end{align*}
\]

Figure 4.4: Decoration of a top-down parse tree for \((1 + 3) \ast 2\), using the attribute grammar of Figure 4.3. Curving arrows again represent attribute flow, which is no longer bottom-up, but is still left-to-right.
Evaluating Attributes – Example

• Attribute grammar in Figure 4.3:
  – This attribute grammar is a good bit messier than the first one, but it is still **L-ATTRIBUTED**, which means that the attributes can be evaluated in a single left-to-right pass over the input
  – In fact, they can be evaluated during an LL parse
  – Each synthetic attribute of a LHS symbol (by definition of synthetic) depends only on attributes of its RHS symbols

Evaluating Attributes – Example

• Attribute grammar in Figure 4.3:
  – Each inherited attribute of a RHS symbol (by definition of L-attributed) depends only on
    • inherited attributes of the LHS symbol, or
    • synthetic or inherited attributes of symbols to its left in the RHS
  – L-attributed grammars are the most general class of attribute grammars that can be evaluated during an LL parse
Evaluating Attributes

- There are certain tasks, such as generation of code for short-circuit Boolean expression evaluation, that are easiest to express with non-L-attributed attribute grammars.
- Because of the potential cost of complex traversal schemes, however, most real-world compilers insist that the grammar be L-attributed.