Attribute Grammars
Chapter 4

Programming Languages

Objective

Today's primary objective:

- Distinguish between syntax and semantics in programming languages
- Know how attribute grammars can be used to enforce semantics of programming languages
- Practice writing attribute grammars

Syntax and Semantics

In general:

- Syntax – form
- Semantics – meaning

In programming languages:

- Syntax – part of the language definition that can be described via a context-free grammar
- Semantics – part of language definition that can't.

Non-Syntax Items

Can't be enforced by a context-free grammar:

- Variables must be declared before use
- Variables must be assigned before use
- Type compatibility in expressions
- Number and type of arguments must match the number and type of formal parameters
- Procedure must contain a return statement for each execution path

Attribute Grammars

Semantic analysis and 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

Example Context-Free Grammar

$$
\begin{aligned}
E &\rightarrow E + T \\
E &\rightarrow E - T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow T / F \\
T &\rightarrow F \\
F &\rightarrow - F
\end{aligned}
$$

Says nothing about what the program means

Example Attribute Grammar

```
E → E + T
        E1.val = E2.val + T.val
E → E - T
        E1.val = E2.val - T.val
E → T
        E.val  = T.val
T → T * F
        T1.val = T2.val * F.val
T → T / F
        T1.val = T2.val / F.val
T → F
        T.val  = F.val
F → - F
        F1.val = - F2.val
```

Evaluating Attribute Grammars

Attribute grammar variables can be characterized as:

- Intrinsic – look-up value
- Synthesized – Variables on the left side of a production get attribute values calculated from values on the right side of the production. In other words, attribute values come from values lower in the parse tree.
- Inherited – Variables on the right side of a production get attribute values from items on the left side of the production or items to the left of the variable. In other words, values come from parents or siblings on the right in the parse tree.

Synthesized Attributes

1. $E_1 \longrightarrow E_2$ + T
   ▷ E_1.val := sum(E_2.val, T.val)
2. $E_1 \longrightarrow E_2$ - T
   ▷ E_1.val := difference(E_2.val, T.val)
3. $E \longrightarrow T$
   ▷ E.val := T.val
4. $T_1 \longrightarrow T_2$ * F
   ▷ T_1.val := product(T_2.val, F.val)
5. $T_1 \longrightarrow T_2$ / F
   ▷ T_1.val := quotient(T_2.val, F.val)
6. $T \longrightarrow F$
   ▷ T.val := F.val
7. $F_1 \longrightarrow$ - F_2
   ▷ F_1.val := additive_inverse(F_2.val)
8. $F \longrightarrow$ (E)
   ▷ F.val := E.val
9. $F \longrightarrow$ `const`
   ▷ F.val := const.val

Simple attribute grammar for constant expressions using standard arithmetic operations

Figure 4.1, page 186

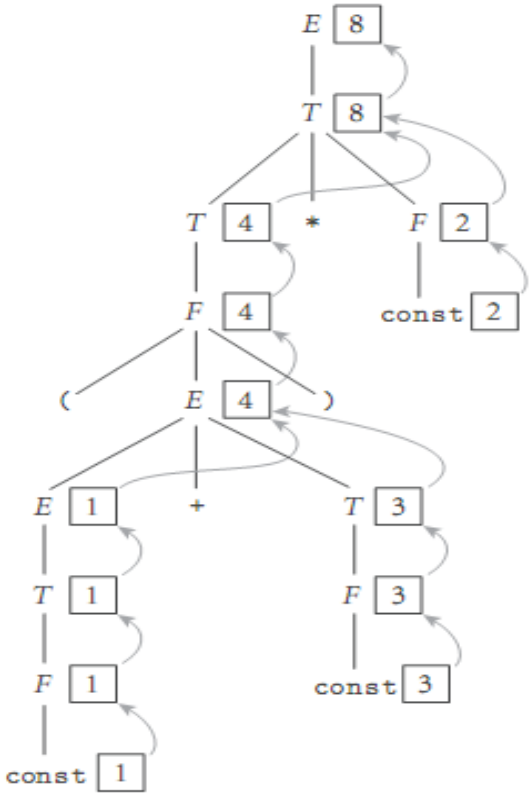

Figure 4.2 **Decoration of a parse tree for (1 + 3) * 2, using the attribute grammar of Figure 4.1.** The val attributes of symbols are shown in boxes. Curving arrows show the attribute flow, which is strictly upward in this case. Each box holds the output of a single semantic rule; the arrow(s) entering the box indicate the input(s) to the rule. At the second level of the tree, for example, the two arrows pointing into the box with the 8 represent application of the rule T_1.val := product(T_2.val, F.val).

Synthesized & Inherited Attributes

1. $E \longrightarrow T\ TT$
   ▷ TT.st := T.val ▷ E.val := TT.val
2. $TT_1 \longrightarrow +\ T\ TT_2$
   ▷ TT_2.st := TT_1.st + T.val ▷ TT_1.val := TT_2.val
3. $TT_1 \longrightarrow -\ T\ TT_2$
   ▷ TT_2.st := TT_1.st − T.val ▷ TT_1.val := TT_2.val
4. $TT \longrightarrow \epsilon$
   ▷ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   ▷ FT.st := F.val ▷ T.val := FT.val
6. $FT_1 \longrightarrow *\ F\ FT_2$
   ▷ FT_2.st := FT_1.st × F.val ▷ FT_1.val := FT_2.val
7. $FT_1 \longrightarrow /\ F\ FT_2$
   ▷ FT_2.st := FT_1.st ÷ F.val ▷ FT_1.val := FT_2.val
8. $FT \longrightarrow \epsilon$
   ▷ FT.val := FT.st
9. $F_1 \longrightarrow -\ F_2$
   ▷ F_1.val := − F_2.val
10. $F \longrightarrow (\ E\ )$
    ▷ F.val := E.val
11. $F \longrightarrow$ `const`
    ▷ F.val := const.val

Attribute grammar for constant expressions based on LL(1) context-free grammar. Several productions have two semantic rules

Figure 4.3, page 190

Evaluating Attributes - Example

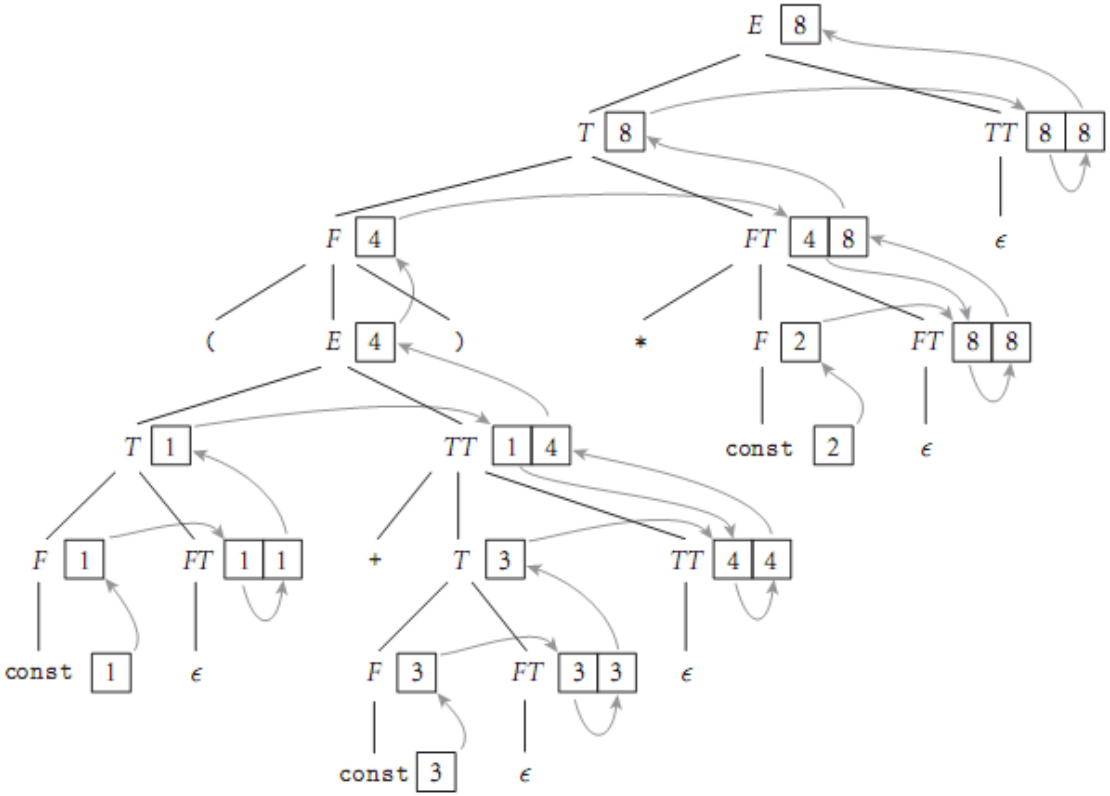

Figure 4.4 **Decoration of a top-down parse tree for `(1 + 3) * 2`, using the AG of Figure 4.3.** Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right. At *FT* and *TT* nodes, the left box holds the st attribute; the right holds `val`.