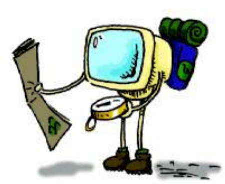
Programming Languages

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 Atributes



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

3

Static vs Dynamic Semantics

- Static Semantics
  - At compile time
    - Type checking
- Dynamic Semantics
  - At runtime
    - Division by zero
    - Out-of-bound indexing an array

4

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

5

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

6

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:

7

Attribute Grammars

```
E → E + T
E → E − T
E → T
T → T * F
T → T / F
T → F
F → - F
F → (E)
F → const
```

- This says nothing about what the program MEANS

8

Attribute Grammars

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

$E \rightarrow E + T$	$E_1.val = E_2.val + T.val$
$E \rightarrow E - T$	$E_1.val = E_2.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T_1.val = T_2.val * F.val$
$T \rightarrow T / F$	$T_1.val = T_2.val / F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow - F$	$F_1.val = - F_2.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow const$	$F.val = C.val$

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 = sum (E_2.val, T.val), cf., Figure 4.1

11

Evaluating Attributes

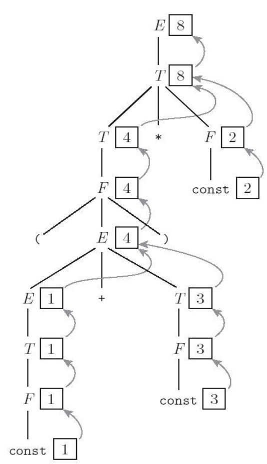


Figure 4.2: **Decoration of a parse tree for** `(1 + 3) * 2`. The val attributes of symbols are shown in boxes. Curving arrows represent the attribute flow, which is strictly upward in this case.

12

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

13

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

14

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:

expr → const expr_tail
expr_tail → - const expr_tail | ε

Create the parse tree for 9 – 4 – 3 and evaluate it.

LL(1) Grammar

- Example 2:

$$
\begin{array}{ll}
E & \rightarrow T\ TT \\
TT_1 & \rightarrow +\ T\ TT_2 \\
TT_1 & \rightarrow -\ T\ TT_1 \\
TT & \rightarrow \varepsilon \\
T & \rightarrow F\ FT \\
FT_1 & \rightarrow *\ F\ FT_2 \\
FT_1 & \rightarrow /\ F\ FT_2 \\
FT & \rightarrow \varepsilon \\
F_1 & \rightarrow -\ F_2 \\
F & \rightarrow (\ E\ ) \\
F & \rightarrow \mathrm{const}
\end{array}
$$

17

Evaluating Attributes – Example

- Attribute grammar :

$$
\begin{array}{lll}
E & \rightarrow T\ TT & E.val = TT.val \\
 & & TT.st = T.val \\
TT_1 & \rightarrow +\ T\ TT_2 & TT_1.v = TT_2.val \\
 & & TT_2.st = TT_1.st + T.val \\
TT_1 & \rightarrow -\ T\ TT_1 & TT_1.val = TT_2.val \\
 & & TT_2.st = TT_1.st - T.val \\
TT & \rightarrow \varepsilon & TT.val = TT.st \\
T & \rightarrow F\ FT & T.val = FT.val \\
 & & FT.st = F.val
\end{array}
$$

18

Evaluating Attributes– Example

- Attribute grammar:

$FT_1 \rightarrow$ * F FT_2	FT_1.val = FT_2.val
	FT_2.st = FT_1.st * F.val
$FT_1 \rightarrow$ / F FT_2	FT_1.val = FT_2.val
	FT_2.st = FT_1.st / F.val
FT $\rightarrow \varepsilon$	FT.val = FT.st
$F_1 \rightarrow$ - F_2	F1.val = - F2.val
F $\rightarrow$ (E)	F.val = E.val
F $\rightarrow$ const	F.val = C.val

19

Evaluating Attributes– Example

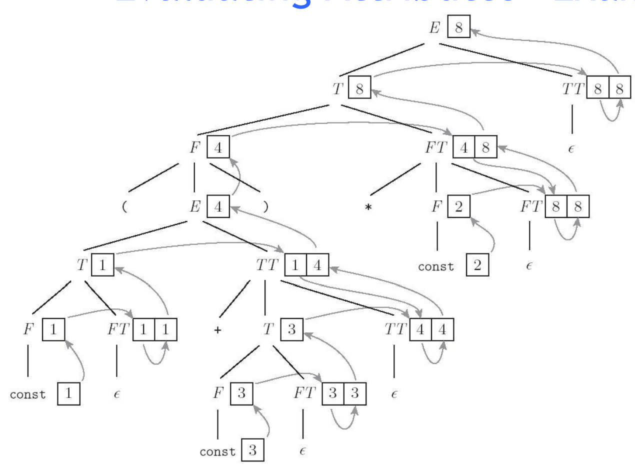


Figure 4.4: **Decoration of a top-down parse tree for (1 + 3) * 2, using the attribute grammar of Figure 4.3.** Curving arrows again represent attribute flow, which is no longer bottom-up, but is still left-to-right.

20

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

21

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

22

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