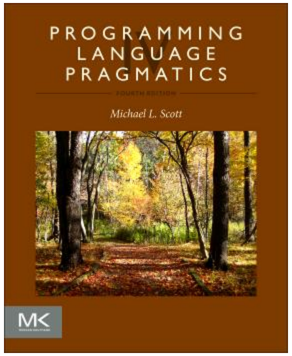
Chapter 4 :: Semantic Analysis

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



ELSEVIER

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

Role of Semantic Analysis

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved
- A common 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

Role of Semantic Analysis

- The PL/0 compiler has no optimization to speak of (there's a tiny little trivial phase, which operates on the syntax tree)
- Its code generator produces MIPs assembler, rather than a machine-independent intermediate form

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
- The notes below discuss attribute grammars and their ad-hoc cousins, ACTION ROUTINES

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:

Attribute Grammars

$$E \rightarrow E + T$$
$$E \rightarrow E - T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow T / F$$
$$T \rightarrow F$$
$$F \rightarrow - F$$

- 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):

```
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
F → (E)        F.val  = E.val
F → 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 should be cast as functions, e.g., E1.val = sum (E2.val, T.val), cf., Figure 4.1

Evaluating Attributes

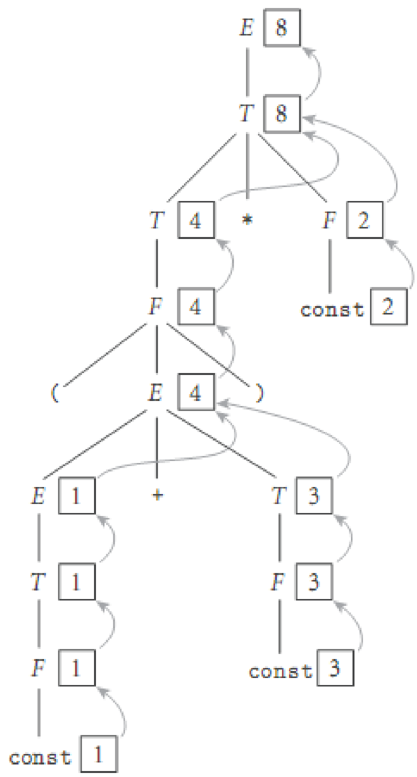


Figure 4.2 Decoration of a parse tree for (1 + 3) * 2, using the attribute grammar of Figure 4.1. The val attributes of symbols are shown in boxes. Curving arrows show the attribute flow, which is strictly upward in this case. Each box holds the output of a single semantic rule; the arrow(s) entering 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).

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:

Evaluating Attributes – Example

- Attribute grammar in Figure 4.3:

1. $E \longrightarrow T\ TT$

   $\triangleright$ TT.st := T.val $\qquad$ $\triangleright$ E.val := TT.val

2. $TT_1 \longrightarrow$ **+** $T\ TT_2$

   $\triangleright$ TT_2.st := TT_1.st + T.val $\qquad$ $\triangleright$ TT_1.val := TT_2.val

3. $TT_1 \longrightarrow$ **-** $T\ TT_2$

   $\triangleright$ TT_2.st := TT_1.st − T.val $\qquad$ $\triangleright$ TT_1.val := TT_2.val

4. $TT \longrightarrow \epsilon$

   $\triangleright$ TT.val := TT.st

5. $T \longrightarrow F\ FT$

   $\triangleright$ FT.st := F.val $\qquad$ $\triangleright$ T.val := FT.val

Evaluating Attributes– Example

- Attribute grammar in Figure 4.3 (continued):

6. $FT_1 \longrightarrow * \; F \; FT_2$
   $\triangleright$ $FT_2.st := FT_1.st \times F.val$ $\triangleright$ $FT_1.val := FT_2.val$

7. $FT_1 \longrightarrow / \; F \; FT_2$
   $\triangleright$ $FT_2.st := FT_1.st \div F.val$ $\triangleright$ $FT_1.val := FT_2.val$

8. $FT \longrightarrow \epsilon$
   $\triangleright$ $FT.val := FT.st$

9. $F_1 \longrightarrow - \; F_2$
   $\triangleright$ $F_1.val := - F_2.val$

10. $F \longrightarrow (\; E \;)$
    $\triangleright$ $F.val := E.val$

11. $F \longrightarrow \texttt{const}$
    $\triangleright$ $F.val := const.val$

Evaluating Attributes– Example

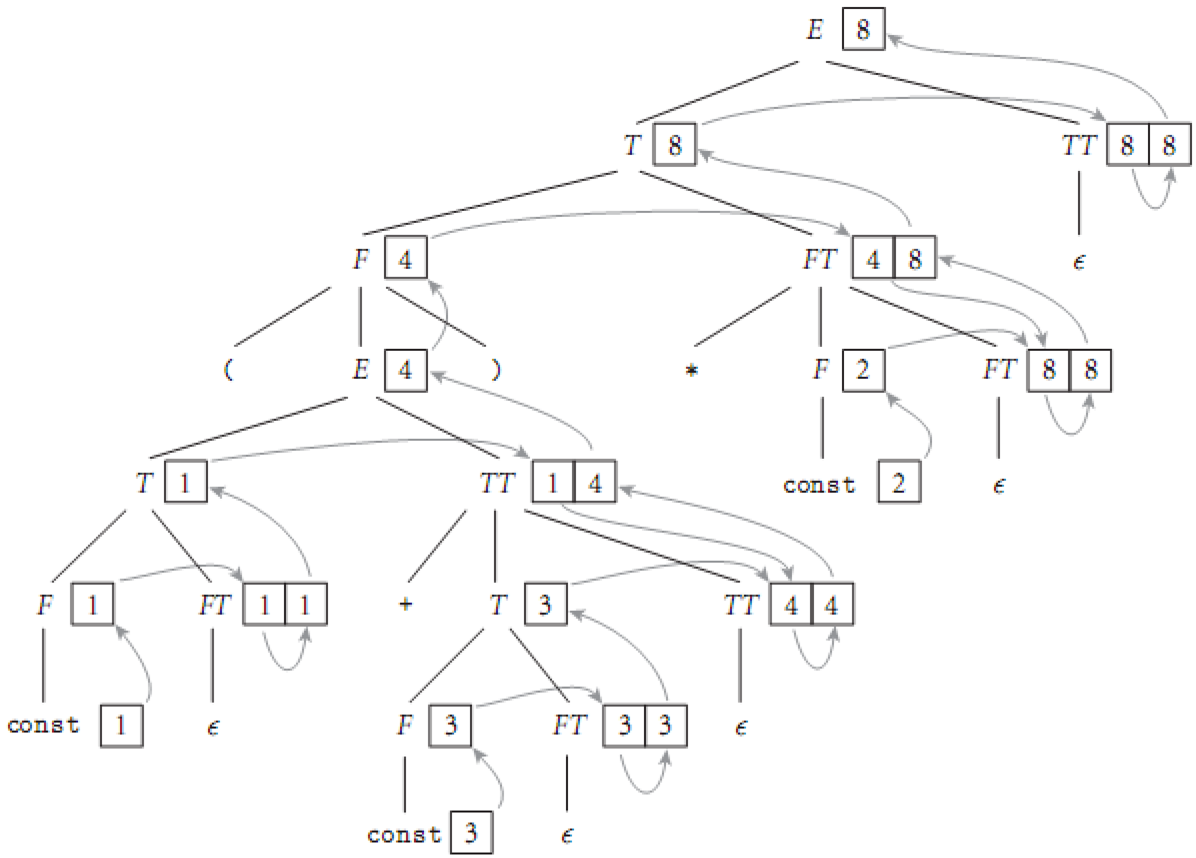


Figure 4.4 Decoration of a top-down parse tree for `(1 + 3) * 2`, using the AG of Figure 4.3. Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right.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

ELSEVIER

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

Evaluating Attributes – Syntax Trees

$E_1 \longrightarrow E_2 \texttt{ + } T$

$\triangleright$ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)

$E_1 \longrightarrow E_2 \texttt{ - } T$

$\triangleright$ E_1.ptr := make_bin_op("−", E_2.ptr, T.ptr)

$E \longrightarrow T$

$\triangleright$ E.ptr := T.ptr

$T_1 \longrightarrow T_2 \texttt{ * } F$

$\triangleright$ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)

$T_1 \longrightarrow T_2 \texttt{ / } F$

$\triangleright$ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)

$T \longrightarrow F$

$\triangleright$ T.ptr := F.ptr

$F_1 \longrightarrow \texttt{ - } F_2$

$\triangleright$ F_1.ptr := make_un_op("+/_", F_2.ptr)

$F \longrightarrow \texttt{ (} E \texttt{) }$

$\triangleright$ F.ptr := E.ptr

$F \longrightarrow \texttt{const}$

$\triangleright$ F.ptr := make_leaf(const.val)

Evaluating Attributes – Syntax Trees

$E \longrightarrow T\ TT$

▷ TT.st := T.ptr

▷ E.ptr := TT.ptr

$TT_1 \longrightarrow$ + $T\ TT_2$

▷ TT_2.st := make_bin_op("+", TT_1.st, T.ptr)

▷ TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow$ - $T\ TT_2$

▷ TT_2.st := make_bin_op("–", TT_1.st, T.ptr)

▷ TT_1.ptr := TT_2.ptr

$TT \longrightarrow \epsilon$

▷ TT.ptr := TT.st

$T \longrightarrow F\ FT$

▷ FT.st := F.ptr

▷ T.ptr := FT.ptr

Evaluating Attributes – Syntax Trees

$FT_1 \longrightarrow$ `*` $F\ FT_2$

▷ FT_2.st := make_bin_op("×", FT_1.st, F.ptr)

▷ FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow$ `/` $F\ FT_2$

▷ FT_2.st := make_bin_op("÷", FT_1.st, F.ptr)

▷ FT_1.ptr := FT_2.ptr

$FT \longrightarrow \epsilon$

▷ FT.ptr := FT.st

$F_1 \longrightarrow$ `-` F_2

▷ F_1.ptr := make_un_op("+/_", F_2.ptr)

$F \longrightarrow$ `(` E `)`

▷ F.ptr := E.ptr

$F \longrightarrow$ `const`

▷ F.ptr := make_leaf(const.val)

Evaluating Attributes – Syntax Trees

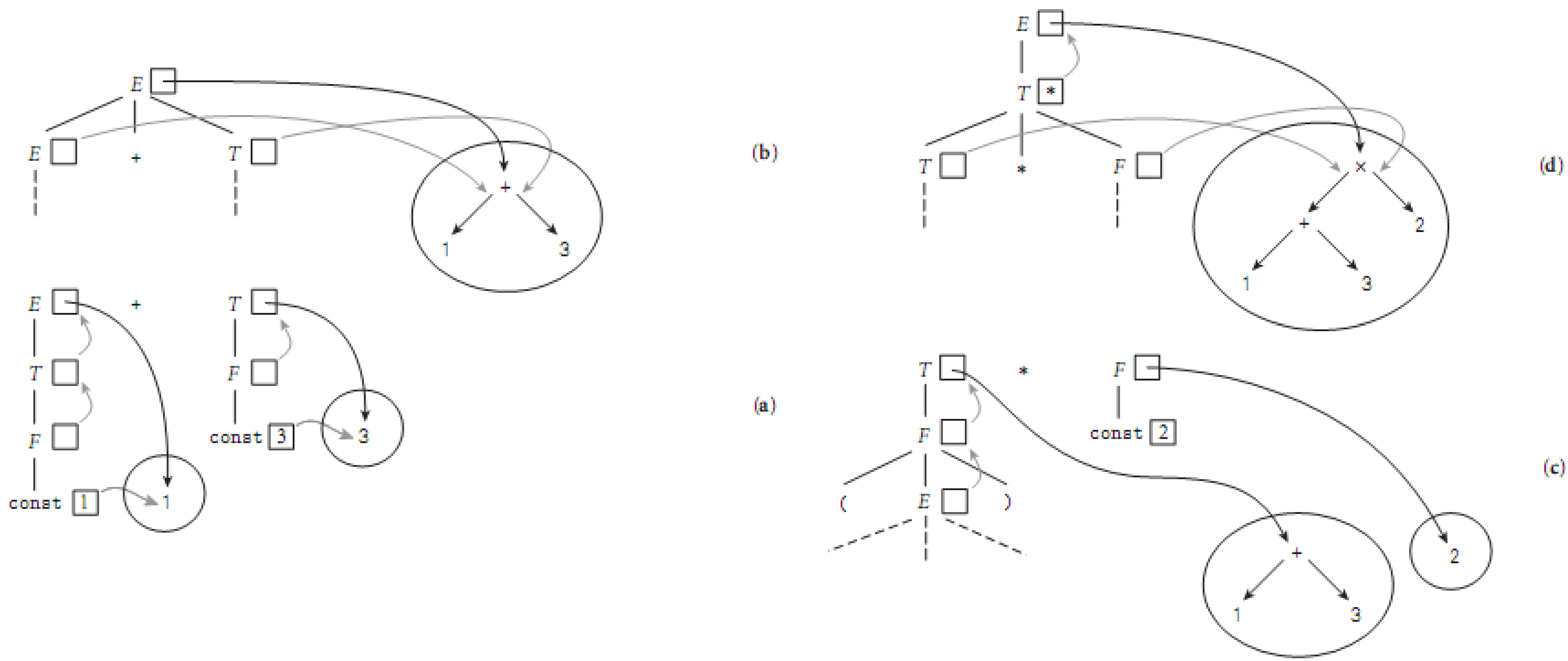


Figure 4.7 Construction of a syntax tree for (1 + 3) * 2 via decoration of a bottom-up parse tree, using the grammar of Figure 4.5. This figure reads from bottom to top. In diagram (a), the values of the constants 1 and 3 have been placed in new syntax tree leaves. Pointer s to these leaves propagate up into the attributes of E and T. In (b), the pointer s to these leaves become child pointer s of a new internal + node. In (c) the pointer to this node propagates up into the attributes of T, and a new leaf is created for 2 . Finally, in (d), the pointer s from T and F become child pointer s of a new internal × node, and a pointer to this node propagates up into the attributes of E .

Evaluating Attributes – Syntax Trees

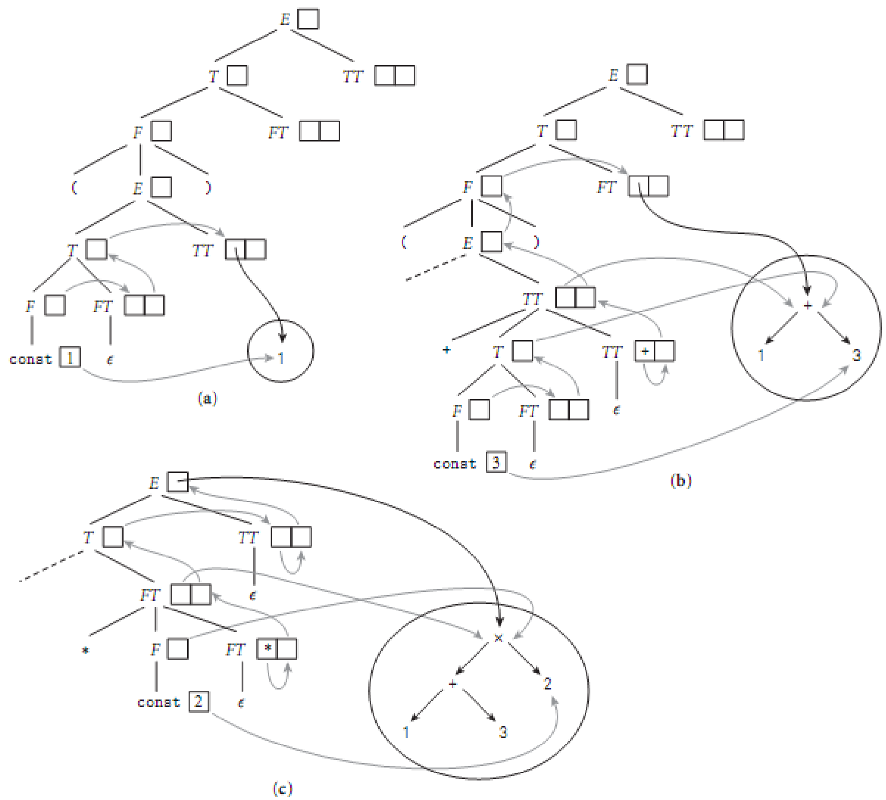


Figure 4.8 Construction of a syntax tree via decoration of a top-down parse tree, using the grammar of Figure 4.6. In the top diagram, (a), the value of the constant 1 has been placed in a new syntax tree leaf. A pointer to this leaf then propagates to the st attribute of TT. In (b), a second leaf has been created to hold the constant 3 . Pointer s to the two leaves then become child pointer s of a new internal + node, a pointer to which propagates from the st attribute of the bottom-most TT, where it was created, all the way up and over to the st attribute of the top-most FT. In (c), a third leaf has been created for the constant 2 . Pointer s to this leaf and to the + node then become the children of a new ×node, a pointer to which propagates from the st of the lower FT, where it was created, all the way to the root of the tree

Action Routines

- We can tie this discussion back into the earlier issue of separated phases v. on-the-fly semantic analysis and/or code generation
- If semantic analysis and/or code generation are interleaved with parsing, then the TRANSLATION SCHEME we use to evaluate attributes MUST be L-attributed

Action Routines

- If we break semantic analysis and code generation out into separate phase(s), then the code that builds the parse/syntax tree must still use a left-to-right (L-attributed) translation scheme
- However, the later phases are free to use a fancier translation scheme if they want

Action Routines

- There are automatic tools that generate translation schemes for context-free grammars or tree grammars (which describe the possible structure of a syntax tree)
  - These tools are heavily used in syntax-based editors and incremental compilers
  - Most ordinary compilers, however, use ad-hoc techniques

Action Routines

- An ad-hoc translation scheme that is interleaved with parsing takes the form of a set of ACTION ROUTINES:
  - An action routine is a semantic function that we tell the compiler to execute at a particular point in the parse
- If semantic analysis and code generation are interleaved with parsing, then action routines can be used to perform semantic checks and generate code

Action Routines

- If semantic analysis and code generation are broken out as separate phases, then action routines can be used to build a syntax tree
  - A parse tree could be built completely automatically
  - We wouldn't need action routines for that purpose

Action Routines

- Later compilation phases can then consist of ad-hoc tree traversal(s), or can use an automatic tool to generate a translation scheme
  – The PL/0 compiler uses ad-hoc traversals that are almost (but not quite) left-to-right
- For our LL(1) attribute grammar, we could put in explicit action routines as follows:

ELSEVIER

Action Routines - Example

- Action routines (Figure 4.9)

$E \longrightarrow T$ { TT.st := T.ptr } TT { E.ptr := TT.ptr }

$TT_1 \longrightarrow$ + T { TT_2.st := make_bin_op("+", TT_1.st, T.ptr) } TT_2 { TT_1.ptr := TT_2.ptr }

$TT_1 \longrightarrow$ - T { TT_2.st := make_bin_op("−", TT_1.st, T.ptr) } TT_2 { TT_1.ptr := TT_2.ptr }

$TT \longrightarrow \epsilon$ { TT.ptr := TT.st }

$T \longrightarrow F$ { FT.st := F.ptr } FT { T.ptr := FT.ptr }

$FT_1 \longrightarrow$ * F { FT_2.st := make_bin_op("×", FT_1.st, F.ptr) } FT_2 { FT_1.ptr := FT_2.ptr }

$FT_1 \longrightarrow$ / F { FT_2.st := make_bin_op("÷", FT_1.st, F.ptr) } FT_2 { FT_1.ptr := FT_2.ptr }

$FT \longrightarrow \epsilon$ { FT.ptr := FT.st }

$F_1 \longrightarrow$ - F_2 { F_1.ptr := make_un_op("+/_", F_2.ptr) }

$F \longrightarrow$ (E) { F.ptr := E.ptr }

$F \longrightarrow$ `const` { F.ptr := make_leaf(const.ptr) }

Figure 4.9 LL(1) grammar with action routines to build a syntax tree.

ELSEVIER

Space Management for Attributes

- Entries in the attributes stack are pushed and popped automatically

$$
\begin{aligned}
\textit{program} &\longrightarrow \textit{stmt\_list}\ \texttt{\$\$} \\
\textit{stmt\_list} &\longrightarrow \textit{stmt\_list}\ \textit{decl} \mid \textit{stmt\_list}\ \textit{stmt} \mid \epsilon \\
\textit{decl} &\longrightarrow \texttt{int id} \mid \texttt{real id} \\
\textit{stmt} &\longrightarrow \texttt{id :=}\ \textit{expr} \mid \texttt{read id} \mid \texttt{write}\ \textit{expr} \\
\textit{expr} &\longrightarrow \textit{term} \mid \textit{expr}\ \textit{add\_op}\ \textit{term} \\
\textit{term} &\longrightarrow \textit{factor} \mid \textit{term}\ \textit{mult\_op}\ \textit{factor} \\
\textit{factor} &\longrightarrow \texttt{(}\ \textit{expr}\ \texttt{)} \mid \texttt{id} \mid \texttt{int\_const} \mid \texttt{real\_const} \mid \\
&\qquad \texttt{float (}\ \textit{expr}\ \texttt{)} \mid \texttt{trunc (}\ \textit{expr}\ \texttt{)} \\
\textit{add\_op} &\longrightarrow \texttt{+} \mid \texttt{-} \\
\textit{mult\_op} &\longrightarrow \texttt{*} \mid \texttt{/}
\end{aligned}
$$

Figure 4.11 Context-free grammar for a calculator language with types and declarations. The intent is that every identifier be declared before use, and that types not be mixed in computations.

Decorating a Syntax Tree

- Syntax tree for a simple program to print an average of an integer and a real

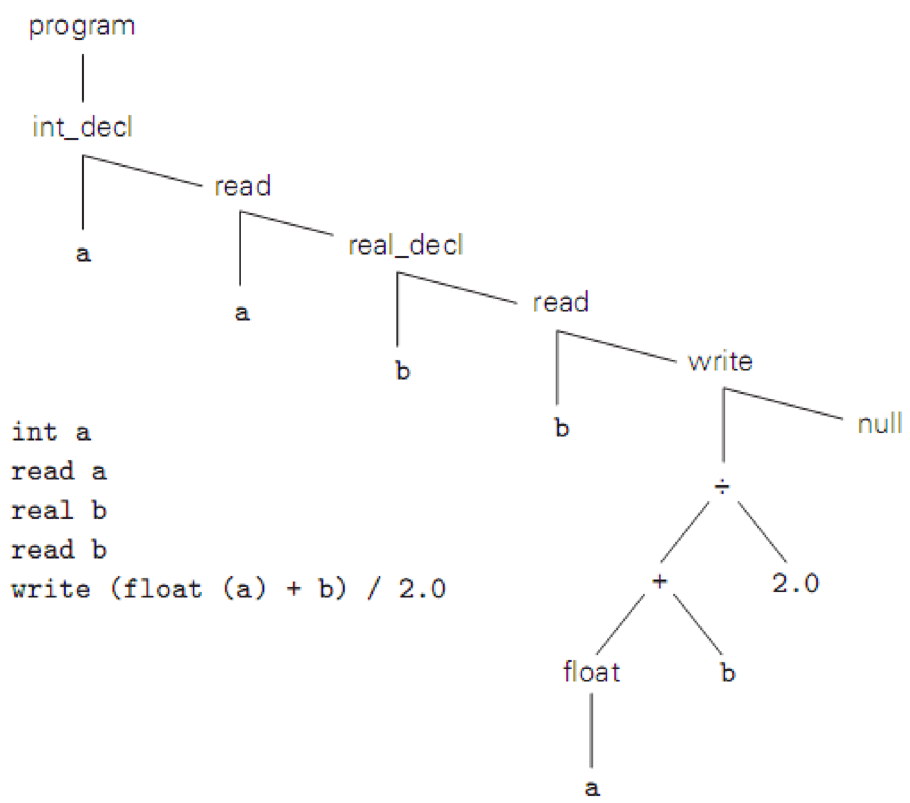


Figure 4.12 Syntax tree for a simple calculator program.

Decorating a Syntax Tree

- Tree grammar representing structure of syntax tree in Figure 4.12

$$
\begin{aligned}
&\mathit{program} \longrightarrow \mathit{item}\\
&\mathit{int\_decl} : \mathit{item} \longrightarrow \mathit{id}\ \mathit{item}\\
&\mathit{read} : \mathit{item} \longrightarrow \mathit{id}\ \mathit{item}\\
&\mathit{real\_decl} : \mathit{item} \longrightarrow \mathit{id}\ \mathit{item}\\
&\mathit{write} : \mathit{item} \longrightarrow \mathit{expr}\ \mathit{item}\\
&\mathit{null} : \mathit{item} \longrightarrow \epsilon\\
&\text{`}\div\text{'} : \mathit{expr} \longrightarrow \mathit{expr}\ \mathit{expr}\\
&\text{`}+\text{'} : \mathit{expr} \longrightarrow \mathit{expr}\ \mathit{expr}\\
&\mathit{float} : \mathit{expr} \longrightarrow \mathit{expr}\\
&\mathit{id} : \mathit{expr} \longrightarrow \epsilon\\
&\mathit{real\_const} : \mathit{expr} \longrightarrow \epsilon
\end{aligned}
$$

Decorating a Syntax Tree

- Sample of <u>complete</u> tree grammar representing structure of syntax tree in Figure 4.12

id : *expr* $\longrightarrow$ ϵ
▷ if ⟨id.name, A⟩ ∈ expr.symtab   -- for some type A
    expr.errors := null
    expr.type := A
  else
    expr.errors := [id.name "undefined at" id.location]
    expr.type := error

int_const : *expr* $\longrightarrow$ ϵ
▷ expr.type := int

real_const : *expr* $\longrightarrow$ ϵ
▷ expr.type := real

'+' : $expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

'−' : $expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

'×' : $expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

'÷' : $expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

float : $expr_1 \longrightarrow expr_2$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ convert_type($expr_2$, $expr_1$, int, real, "float of non-int")

trunc : $expr_1 \longrightarrow expr_2$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ convert_type($expr_2$, $expr_1$, real, int, "trunc of non-real")