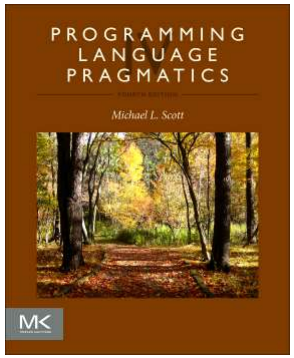
Chapter 4 :: Semantic Analysis

- CSE307/526: Principles of Programming Languages
- https://ppawar.github.io/CSE307-F18/index.html

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
  - information gathered is needed by the code generator

Role of Semantic Analysis

- Describes meaning of a program
- Cannot be described by a context-free grammar
- • Enforces semantic rules
- Builds intermediate representation (e.g., abstract syntax tree)
- Fills symbol table
- Passes results to intermediate code generator
- 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 and then follows with separate, sequential phases for semantic analysis and code generation

Enforcing Semantic Rules

- Static semantic rules
  - Enforced by compiler at compile time
  - Example: Do not use undeclared variable.
- Dynamic semantic rules
  - Compiler generates code for enforcement at runtime.
  - Examples: Division by zero, array index out of bounds
  - Some compilers allow these checks to be disabled.
- Formal mechanism for enforcing semantic rules
  - Attribute grammars

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

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

- This says nothing about what the program MEANS

$E \longrightarrow E + T$
$E \longrightarrow E - T$
$E \longrightarrow T$
$T \longrightarrow T * F$
$T \longrightarrow T / F$
$T \longrightarrow F$
$F \longrightarrow - F$
$F \longrightarrow (E)$
$F \longrightarrow \texttt{const}$

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

Generates all properly formed constant arithmetic expressions without their meaning

- Additional notation based on attributes to tie expressions to mathematical concepts

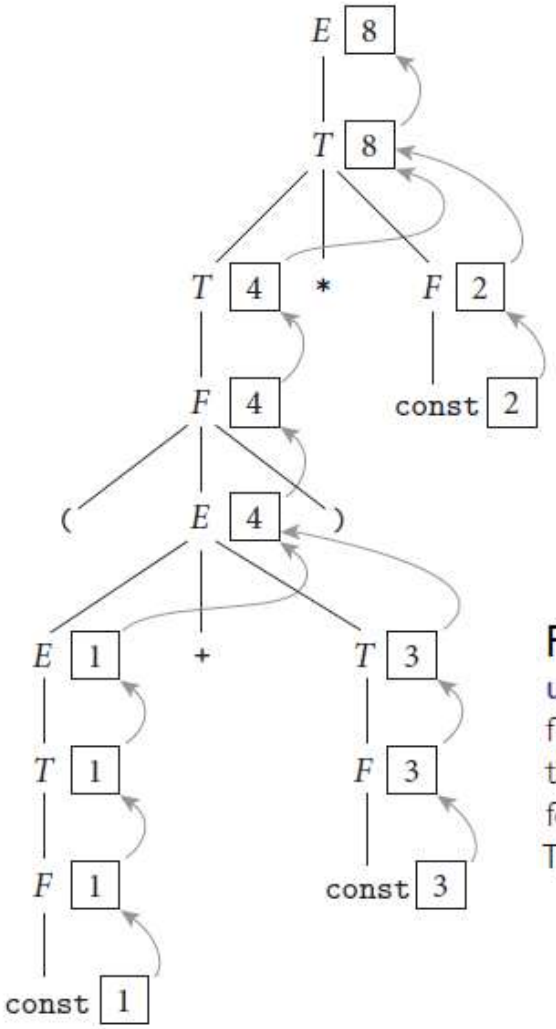
- S.val is an arithmetic value of the token string derived from S
- Val of const is provided by the scanner
- Attribute grammar represents set of rules for each production to specify how the vals of different symbols are related
- When more than one symbol of a production has the same name, subscripts are used to distinguish them.
- Subscripts are not part of CFG.
- The code fragments for the rules are called SEMANTIC FUNCTIONS

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



- Process of evaluating attributes is called annotation or decoration of the parse tree
- Figure 4.2 shows parse tree of synthesized (calculated) attribute flow entirely from bottom to up

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

- The grammar above is called S-ATTRIBUTED because it uses only synthesized attributes
- Its ATTRIBUTE FLOW (attribute dependence graph) is purely bottom-up

SYNTHESIZED ATTRIBUTES: EXAMPLE

The language

$$\mathcal{L} = \{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n > 0\} = \{\mathbf{abc}, \mathbf{aabbcc}, \mathbf{aaabbbccc}, \ldots\}$$

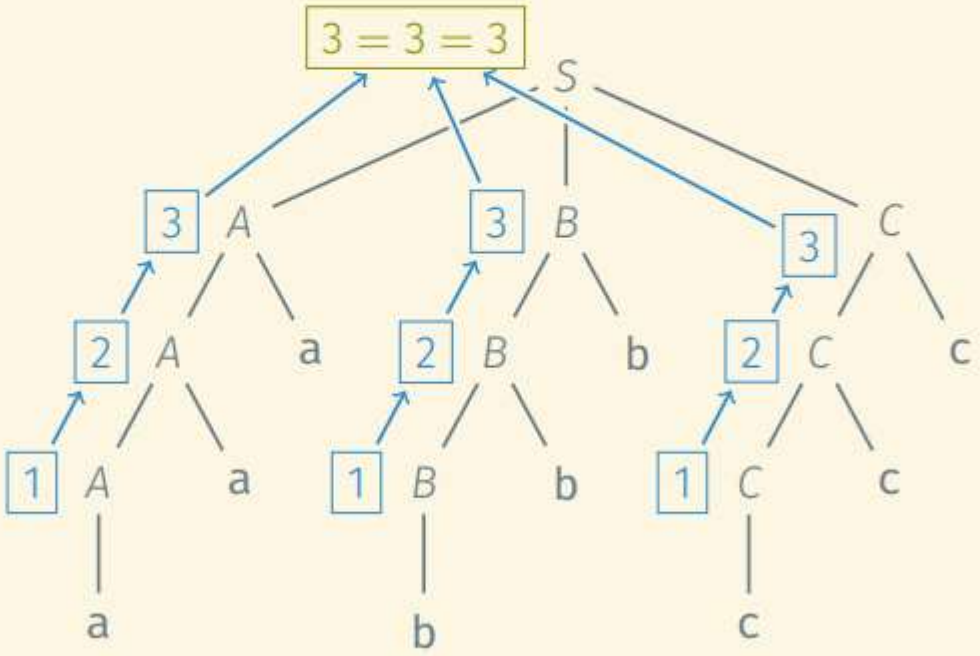
is not context-free but can be defined using an attribute grammar:

$S \to A\,B\,C$	{Condition: $A.count = B.count = C.count$}
$A \to \mathbf{a}$	$\{A.count = 1\}$
$A_1 \to A_2\,\mathbf{a}$	$\{A_1.count = A_2.count + 1\}$
$B \to \mathbf{b}$	$\{B.count = 1\}$
$B_1 \to B_2\,\mathbf{b}$	$\{B_1.count = B_2.count + 1\}$
$C \to \mathbf{c}$	$\{C.count = 1\}$
$C_1 \to C_2\,\mathbf{c}$	$\{C_1.count = C_2.count + 1\}$

SYNTHESIZED ATTRIBUTES: PARSE TREE DECORATION (1)

Input: aaabbbccc

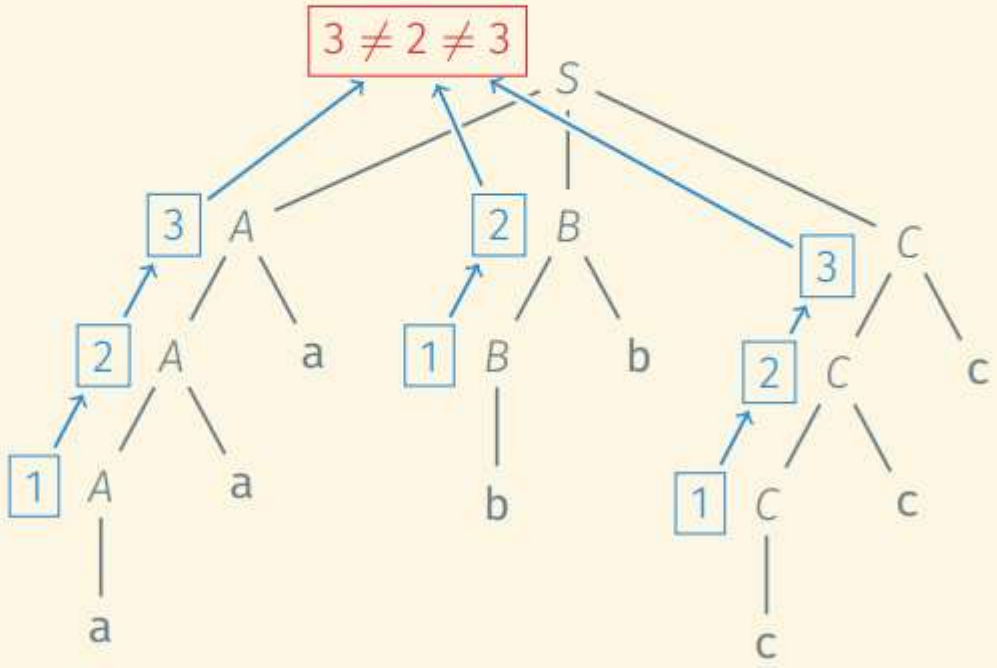
Parse tree:



SYNTHESIZED ATTRIBUTES: PARSE TREE DECORATION (2)

Input: aaabbccc

Parse tree:

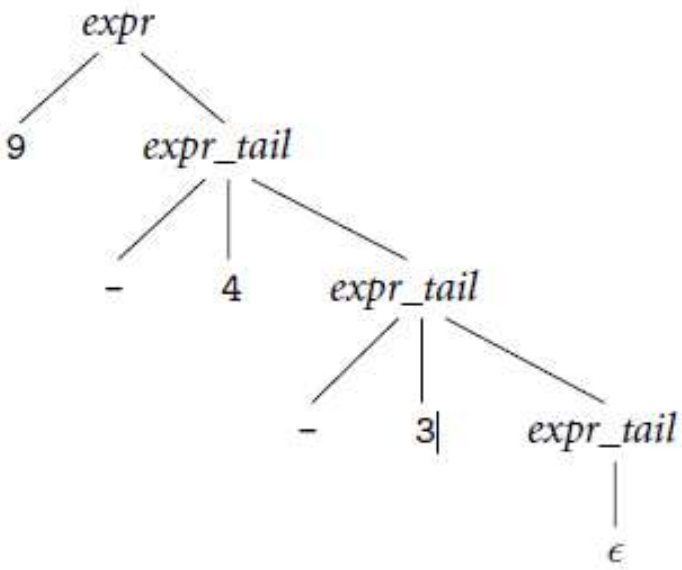


Evaluating Attributes

- 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

expr ⟶ `const` *expr_tail*

expr_tail ⟶ `-` `const` *expr_tail* | ϵ



- Subtraction is left associative, hence right subtree can't be summarized with a single numeric value.
- Solution
  - Allow to pass attributes not only bottom-up but also left-right in the tree
  - Then 9 can be passed into topmost *expr_tail* node
  - It can be combined with 4
  - Resulting 5 can be passed into middle *expr_tail* mode combined with the 3 to make 2
  - Pass the result upward to the root

INHERITED ATTRIBUTES: EXAMPLE

Again, we consider the language

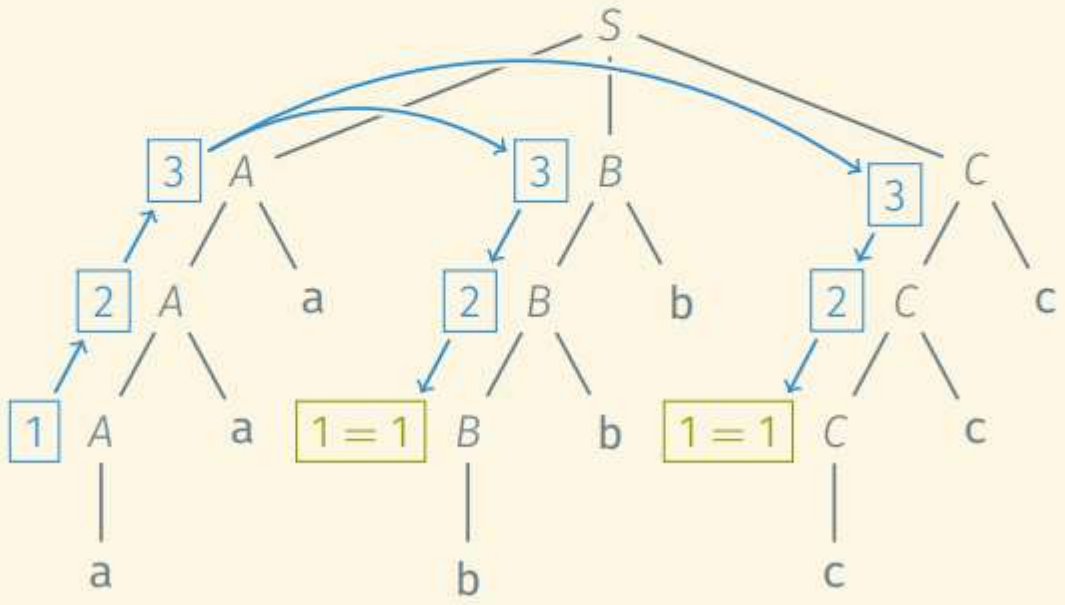
$$\mathcal{L} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n > 0\} = \{\mathtt{abc}, \mathtt{aabbcc}, \mathtt{aaabbbccc}, \ldots\}.$$

$S \to A\,B\,C$	$\{B.inhCount = A.count; C.inhCount = A.count\}$
$A \to \mathtt{a}$	$\{A.count = 1\}$
$A_1 \to A_2\,\mathtt{a}$	$\{A_1.count = A_2.count + 1\}$
$B \to \mathtt{b}$	$\{Condition : B.inhCount = 1\}$
$B_1 \to B_2\,\mathtt{b}$	$\{B_2.inhCount = B_1.inhCount - 1\}$
$C \to \mathtt{c}$	$\{Condition : C.inhCount = 1\}$
$C_1 \to C_2\,\mathtt{c}$	$\{C_2.inhCount = C_1.inhCount - 1\}$

INHERITED ATTRIBUTES: PARSE TREE DECORATION (1)

Input: aaabbbccc

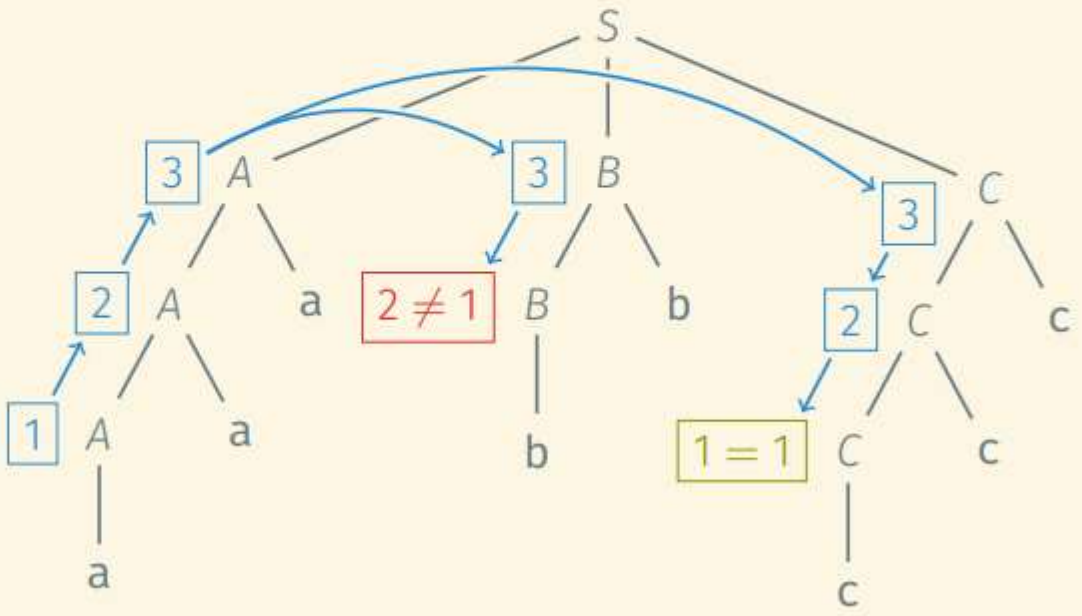
Parse tree:



INHERITED ATTRIBUTES: PARSE TREE DECORATION (2)

Input: aaabbccc

Parse tree:



S-attributed and L-attributed grammars

- S-attributed grammars are those that can be evaluated on-the-fly with an LR parse
- L-attributed grammars are those that can be evaluated on-the-fly with an LL parse
- Evaluating on-the-fly means interleaving parsing and attribute evaluation
- One-pass compiler fully interleaves parsing and code generation
-

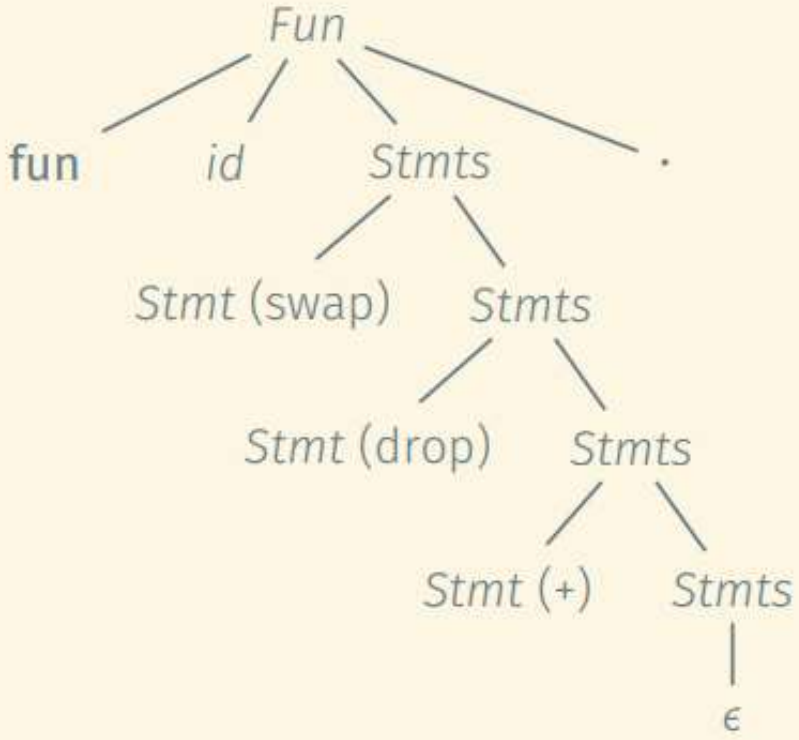
Abstract Syntax Trees

- Problem with parse trees
  - They represent the full derivation of the program using grammar rules.
  - Some grammar variables are there only to aid in parsing (e.g., to eliminate left-recursion or common prefixes).
  - Code generator is easier to implement if the output of the parser is as compact as possible.
- Abstract syntax tree (AST)
  - A compressed parse tree that represents the program structure rather than the parsing process.

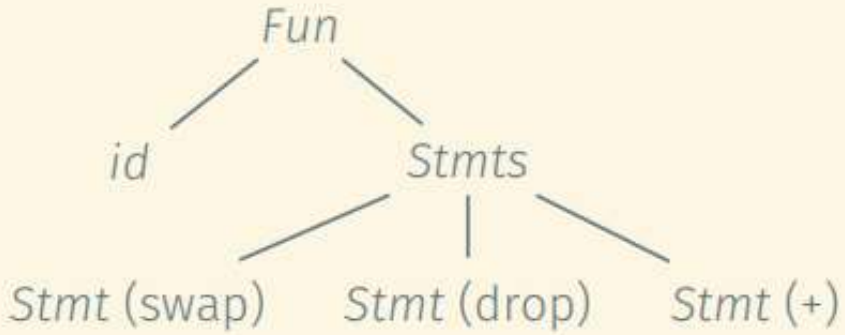
ABSTRACT SYNTAX TREE: EXAMPLE (1)

$$\mathit{Fun} \rightarrow \textbf{fun}\ \mathit{id}\ \mathit{Stmts}\ .$$
$$\mathit{Stmts} \rightarrow \epsilon$$
$$\mathit{Stmts} \rightarrow \mathit{Stmt}\ \mathit{Stmts}$$
$$\mathit{Stmt} \rightarrow \ldots$$

```
fun foo
  swap drop +
.
```



AST:



Decorating a Syntax Tree

- Tree grammar representing structure of syntax tree in Figure 4.12

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

Decorating a Syntax Tree

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

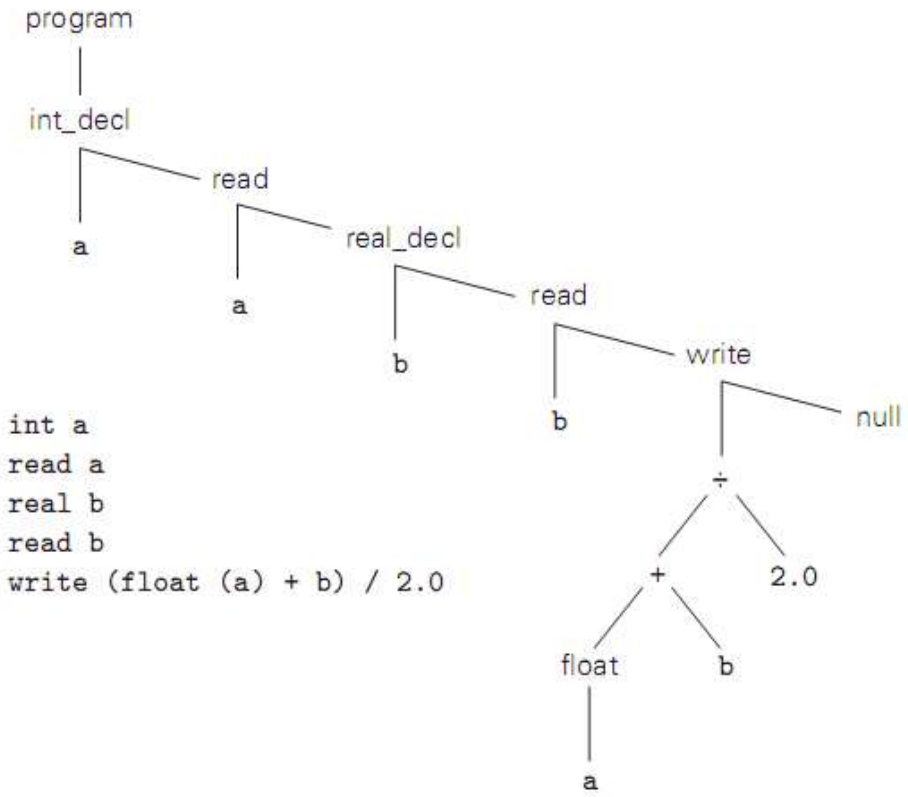


Figure 4.12 Syntax tree for a simple calculator program.

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
- If semantic analysis and code generation are broken out as separate phases, then action routines can be used to build a syntax tree

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.

YACC Grammar

```
%token NAME NUMBER
%%
statement: NAME '=' expression
      |    expression          { printf("= %d\n", $1); }
      ;

expression: expression '+' NUMBER  { $$ = $1 + $3; }
      |     expression '-' NUMBER  { $$ = $1 - $3; }
      |     NUMBER                 { $$ = $1; }
      ;
```

Decorating a Syntax Tree

- Sample of complete tree grammar representing structure of syntax tree in Figure 4.12

id : *expr* ⟶ 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* ⟶ ϵ
  ▷ expr.type := int

real_const : *expr* ⟶ ϵ
  ▷ expr.type := real

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

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

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

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

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

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