



CS49K Programming Languages

Chapter 4 Semantic Analysis

LECTURE 6: SEMANTIC ANALYSIS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Attribute Grammar
- Evaluation of Attributes
- Action Routine
- Tree Grammars and Syntax Tree Decoration

Attribute Grammar I

SECTION 1

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

- 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

Bottom-Up CFG for Constant Expressions

- This grammar will generate all properly formed constant expressions over the basic arithmetic operators, but it says nothing about their meaning. To tie these expressions to mathematical concepts, we need **additional notation**.
- The most common is based on attributes.
- In our expression grammar, we can associate a **val** attribute with each **E**, **T**, **F**, and **const** in the grammar. The intent is that for any symbol **S**, **S.val** will be the meaning, as an arithmetic value, of the token string derived from **S**.

$$E \longrightarrow E + T$$

$$E \longrightarrow E - T$$

$$E \longrightarrow T$$

$$T \longrightarrow T * F$$

$$T \longrightarrow T / F$$

$$T \longrightarrow F$$

$$F \longrightarrow - F$$

$$F \longrightarrow (E)$$

$$F \longrightarrow \text{const}$$

Bottom-Up CFG for Constant Expressions

- We assume that the **val** of a **const** is provided to us by the scanner. We must then invent a set of rules for each production, to specify how the **vals** of different symbols are related. The resulting attribute grammar (**AG**) is shown in Figure 4.1.

$$E \longrightarrow E + T$$

$$E \longrightarrow E - T$$

$$E \longrightarrow T$$

$$T \longrightarrow T * F$$

$$T \longrightarrow T / F$$

$$T \longrightarrow F$$

$$F \longrightarrow - F$$


$$F \longrightarrow (E)$$

$$F \longrightarrow \text{const}$$

Attribute Grammar (AG) from CFG

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

Attribute Grammar - Relaxed

- In a strict definition of attribute grammars, copy **rules** and **semantic function calls** are the only two kinds of permissible rules. In our examples, we use a  symbol to introduce each code fragment corresponding to a single rule. In practice, it is common to allow rules to consist of small fragments of code in some well-defined notation (e.g., the language in which a compiler is being written), so that simple semantic functions can be written out “**in-line.**”

Attribute Grammar - Relaxed

- In relaxed notation, the rule for the first production in Figure 4.1 might be simply $E_1.val := E_2.val T.val$. As another example, suppose we wanted to count the elements of a comma-separated list:

$L \rightarrow id \quad LT$

$LT \rightarrow , \quad L$

$LT \rightarrow \varepsilon$

▷ $L.c := 1 + LT.c$

▷ $LT.c := L.c$

▷ $LT.c := 0$

Note: rule 1 sets the LHS count to be 1 greater than RHS. Like explicit semantic functions, in-line rules are not allowed to refer to any variables or attributes outside the current productions. We will relax this restriction when we introduce action routines in Section 4.4.

Attribute Grammar

- The purpose of AG is to associate meaning with the grammar. We can use any notation and types whose meaning are already well defined.
- In Examples 4.4 and 4.5, we associated numeric values with the symbols in CFG – and thus with parse tree nodes – using semantic functions drawn from ordinary arithmetic.
- In a compiler or interpreter for a full programming language, the **attributes** of tree nodes might include:

Note: For purposes other than translation – e.g., in a theorem prover or machine-independent language definition – attributes might be drawn from the disciplines of denotational, operational, or axiomatic semantics.

Attribute Grammar

1. for an identifier, a **reference** to information about it in the symbol table.
2. for an expression, its **type**
3. for a statement expression, a **reference** to corresponding code in the compiler's Intermediate form.
4. for almost **construct indication** of the file name, line, and column where the corresponding source code begins
5. for any internal node, a list of **semantic errors** found in the subtree below

Attribute Grammar II

SECTION 2

Attribute Grammar

- **Attribute grammar** is a special form of **context-free grammar** where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute Grammar

- **Attribute grammar** is a medium to provide **semantics** to the **context-free grammar** and it can help specify the syntax and semantics of a programming language.
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Example:

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Attribute Grammars

- Carry some semantic information along through parse tree
- Useful for
 - Static semantic specification
 - Static semantic checking in compilers

Attribute Grammars

- An attribute grammar is a CFG $G = (S, N, T, P)$ with annotations
 - For each grammar symbol x , there is a set $A(x)$ of **attribute values**
 - Each production rule has a set of functions that define certain attributes of the non-terminals in the rule.
 - Each production rule has a (possibly empty) **set of predicates** to check for attribute consistency
 - Valid derivations have predicates true for each node

Attribute Grammars

- Synthesized attributes
 - Are determined from nodes of children in parse tree
 - If $X_0 \rightarrow X_1 \dots X_n$ is a rule, the $S(X_0) = f(A(X_1), \dots, A(X_n))$
 - the value of X_0 determined by children
- Pass semantic information up the tree

Attribute Grammars

- Inherited attributes
 - Are determined from parent and siblings
 - $I(X_j) = f(A(X_0), \dots, A(X_n))$, Often just $X_0 \dots X_{j-1}$
 - Siblings to the left in parse tree
- Pass semantic information down the tree

Synthesized attributes

- These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$S \rightarrow ABC$

- If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S .
- As in our previous example ($E \rightarrow E + T$), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes

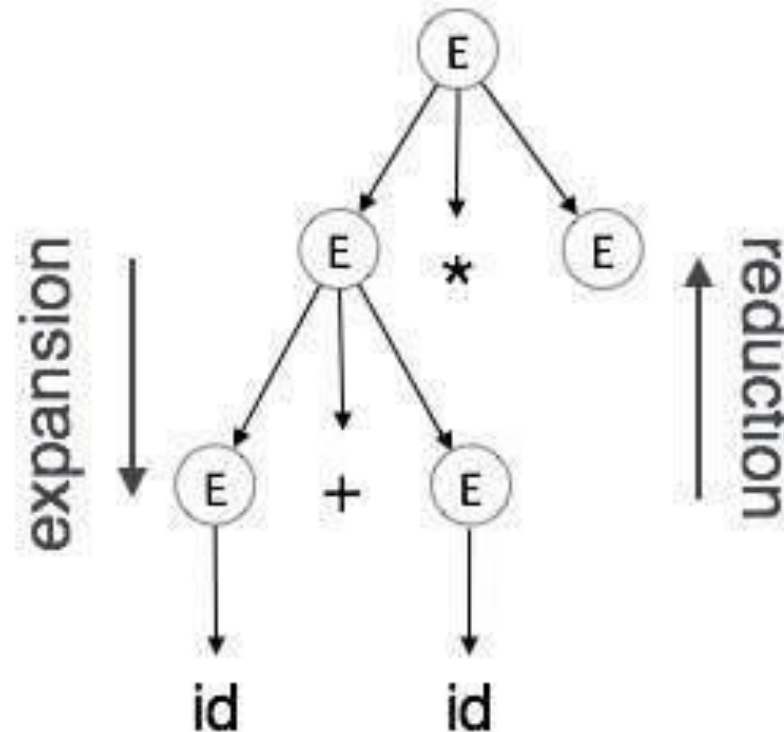
- In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

$S \rightarrow ABC$

- A can get values from S, B and C.
- B can take values from S, A, and C.
- Likewise, C can take values from S, A, and B.

Expansion

When a non-terminal is expanded to terminals as per a grammatical rule:



Reduction

- When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).
- Semantic analysis uses Syntax Directed Translations to perform the above tasks.
- Semantic analyzer receives **AST** (Abstract Syntax Tree) from its previous stage (syntax analysis).
- Semantic analyzer attaches attribute information with **AST**, which are called **Attributed AST**.
- Attributes are two tuple value, **<attribute name, attribute value>**

Reduction

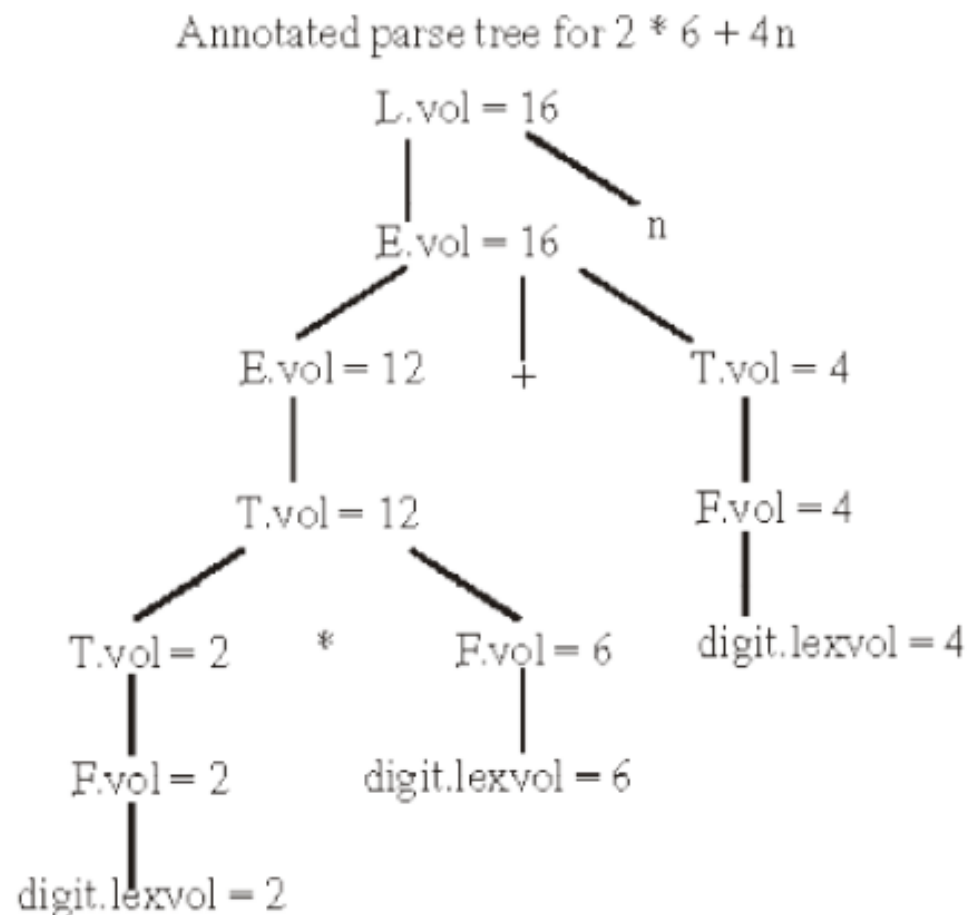
For example:

```
int value = 5;  
<type, "integer">  
<presentvalue, "5">
```

Note: For every production, we attach a semantic rule.

Syntax directed translation schemes (SDT)

Example : An annotated parse tree for $2 * 6 + 4n$



Syntax Directed Translation

The conceptual view of syntax-directed translation,
Input string \rightarrow parse tree \rightarrow dependency graph \rightarrow evaluation order for semantic rules

Syntax directed definitions (SDD)

This is a generalization of context free grammar in which each grammar symbol has an associated set of attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

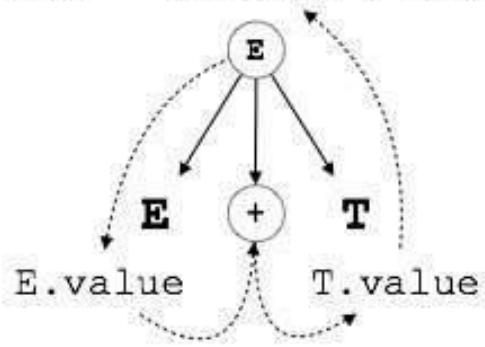
S-attributed SDT

If an SDT uses only synthesized attributes, it is called as **S-attributed SDT**. These attributes are evaluated using **S-attributed SDTs** that have their semantic actions written after the production (right hand side).

As depicted above, attributes in **S-attributed SDTs** are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

[LR-parsing]

`E.value = E.value + T.value`



L-attributed SDT

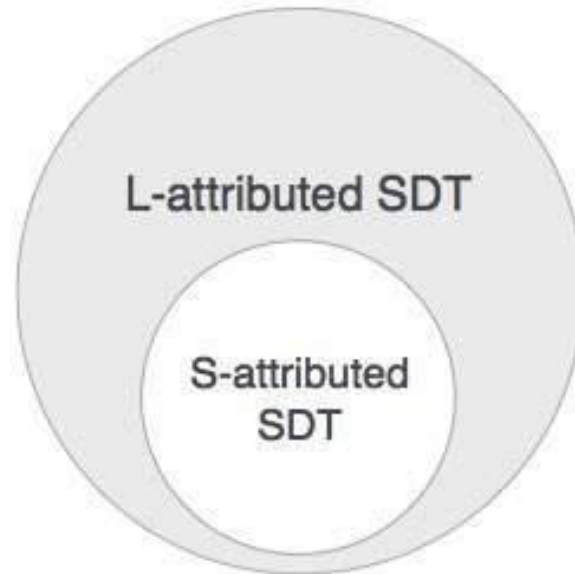
- This form of **SDT** uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
- In **L-attributed SDTs**, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

$S \rightarrow ABC$

- S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

L-attributed SDT

- Attributes in **L-attributed SDTs** are evaluated by **depth-first** and **left-to-right** parsing manner.



- We may conclude that if a definition is **S-attributed**, then it is also **L-attributed** as **L-attributed** definition encloses **S-attributed** definitions.

Evaluating Attributes

SECTION 3

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
- 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 process of evaluating attributes is called **annotation**, or **Decoration**, of the parse tree
 - 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)**

Evaluating Attributes

$$(1 + 3) * 2$$

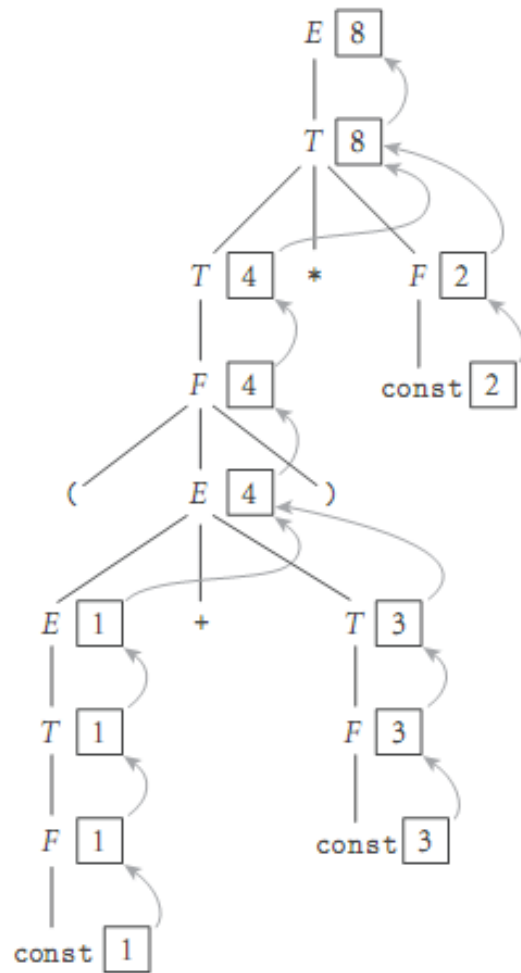


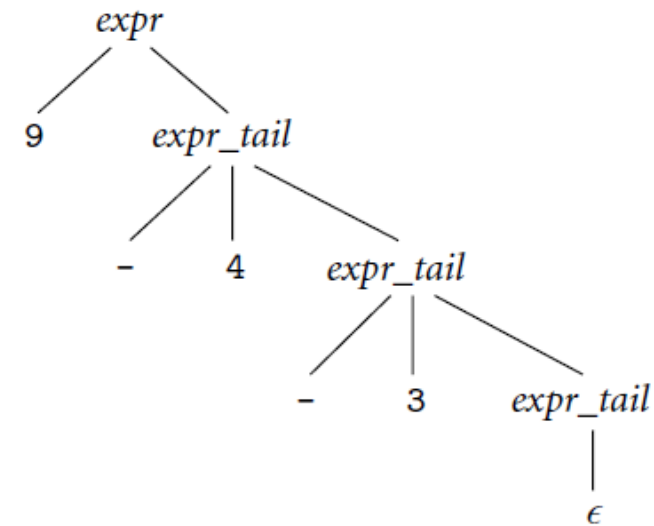
Figure 4.2 Decoration of a parse tree for $(1 + 3) * 2$, using the attribute grammar of Figure 4.1. The val attributes of symbols are shown in boxes. Curving arrows show the attribute flow, which is strictly upward in this case. Each box holds the output of a single semantic rule; the arrow(s) entering 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 $T_1.\text{val} := \text{product}(T_2.\text{val}, F.\text{val})$.

Top-down CFG and parse tree for subtraction

- As a simple example of inherited attributes, consider the following simplified fragment of an LL(1) expression grammar (here covering only subtraction):

$$\text{expr} \longrightarrow \text{const } \text{expr_tail}$$
$$\text{expr_tail} \longrightarrow - \text{const } \text{expr_tail} \mid \epsilon$$

For the expression $9 - 4 - 3$, we obtain the following parse tree:



Top-down CFG and parse tree for subtraction

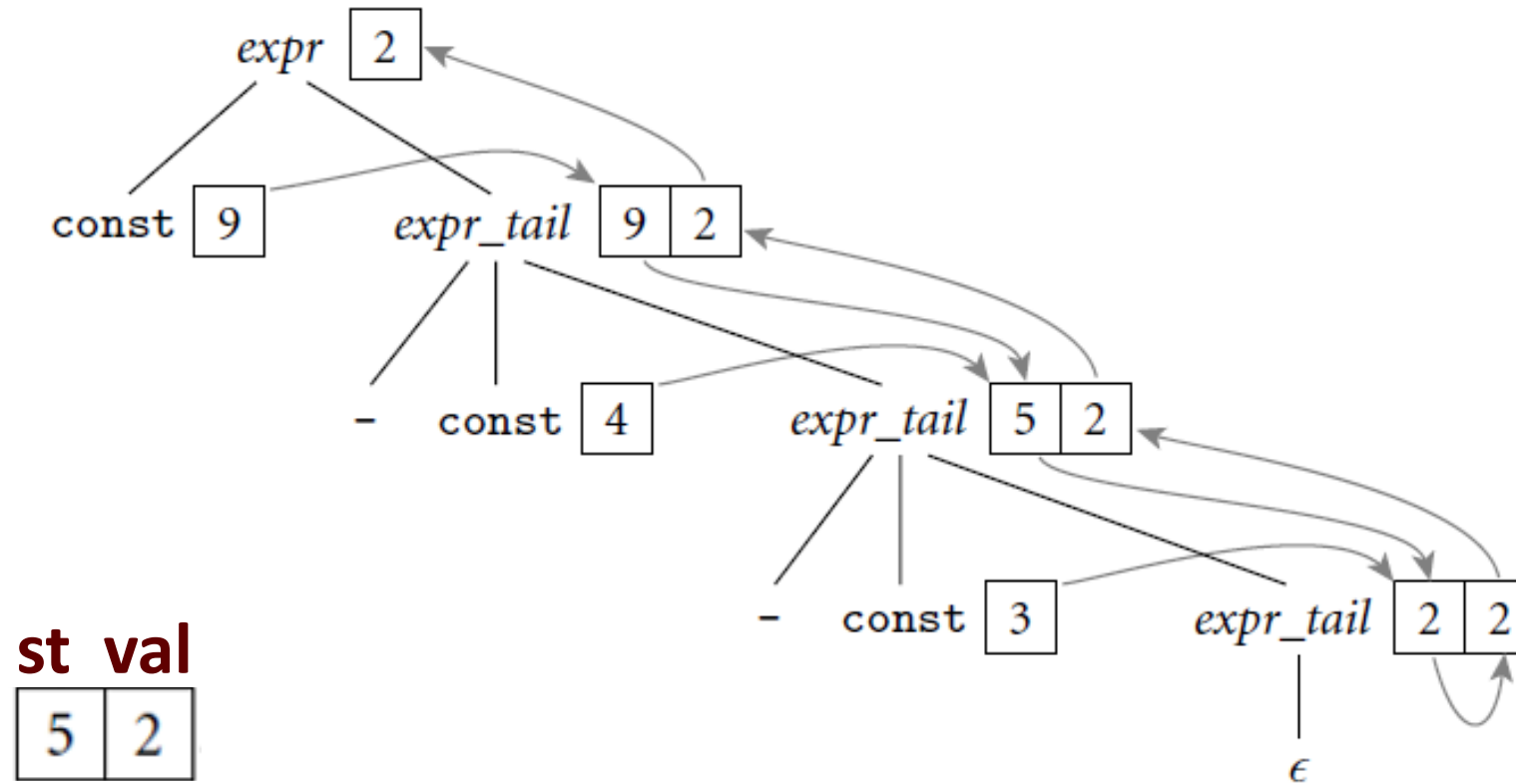
- If we want to create an attribute grammar that accumulates the value of the overall expression into the root of the tree, we have a problem: because **subtraction** is left associative, we cannot summarize the right subtree of the root with a single numeric value. **[No SLR]**
- If we want to decorate the tree bottom-up, with an **S-attributed** grammar, we must be prepared to describe an arbitrary number of right operands in the attributes of the top-most **expr_tail** node.
- This is indeed possible, but it defeats the purpose of the formalism: in effect, it requires us to embed the entire tree into the attributes of a single node, and do all the real work inside a single semantic function.

Note: LL(1) Grammar does not fit S-attributed grammar.

Decoration with Left-to-Right Attribute Flow (L-attributed)

- If, however, we are allowed to pass attribute values not only left-to-right attribute flow also left-to-right in the tree, then we can pass the 9 into the top-most **expr_tail** node, where it can be combined (in proper **left-associative fashion**) with the 4.
- The resulting 5 can then be passed into the middle expr tail node, combined with the 3 to make 2, and then passed upward to the root:

Decoration with Left-to-Right Attribute Flow (L-attributed)



Top-down AG for Subtraction

To effect this style of decoration, we need the following attribute rules:

$expr \rightarrow const\ expr_tail$

▷ $expr_tail.st := const.val$

▷ $expr.val := expr_tail.val$

$expr_tail_1 \rightarrow -\ const\ expr_tail_2$

▷ $expr_tail_2.st := expr_tail_1.st - const.val$

▷ $expr_tail_1.val := expr_tail_2.val$

$expr_tail \rightarrow \epsilon$

▷ $expr_tail.val := expr_tail.st$

st	val
5	2

- In each of the first two productions, the first rule serves to copy the left context into a “subtotal” (**st**) attribute; the second rule copies the final value from the right-most leaf back up to the root.
- In the **expr_tail** nodes of the picture in Example 4.8, the left box holds the **st** attribute; the right holds **val**.

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. $T \longrightarrow F \ FT$
 $\triangleright FT.st := F.val \qquad \triangleright T.val := FT.val$
6. $FT_1 \longrightarrow * \ F \ FT_2$
 $\triangleright FT_2.st := FT_1.st \times F.val \qquad \triangleright FT_1.val := FT_2.val$
7. $FT_1 \longrightarrow / \ F \ FT_2$
 $\triangleright FT_2.st := FT_1.st \div F.val \qquad \triangleright FT_1.val := FT_2.val$
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. $F \longrightarrow \text{const}$
 $\triangleright F.val := \text{const.val}$

Figure 4.3

Top-Down AG for Constant Expression

- We can flesh out the grammar fragment of to produce amore complete expression grammar, as shown (with shorter symbol names) in Figure 4.3.
- The underlying **CFG** for this grammar accepts the same language as the one in **Figure 4.1**, but where that one was **SLR(1)**, this one is **LL(1)**. Attribute flow for a parse of **(1 + 3) * 2**, using the **LL(1)** grammar, appears in **Figure 4.4**.
- As in the grammar fragment of Example 4.9, the value of the left operand of each operator is carried into the **TT** and **FT** productions by the **st (subtotal)** attribute.
- The relative complexity of the **attribute flow** arises from the fact that operators are left associative, but the grammar cannot be left recursive: the left and right operands of a given operator are thus found in separate productions. Grammars to perform semantic analysis for practical languages generally require some **non-S-attributed** flow.

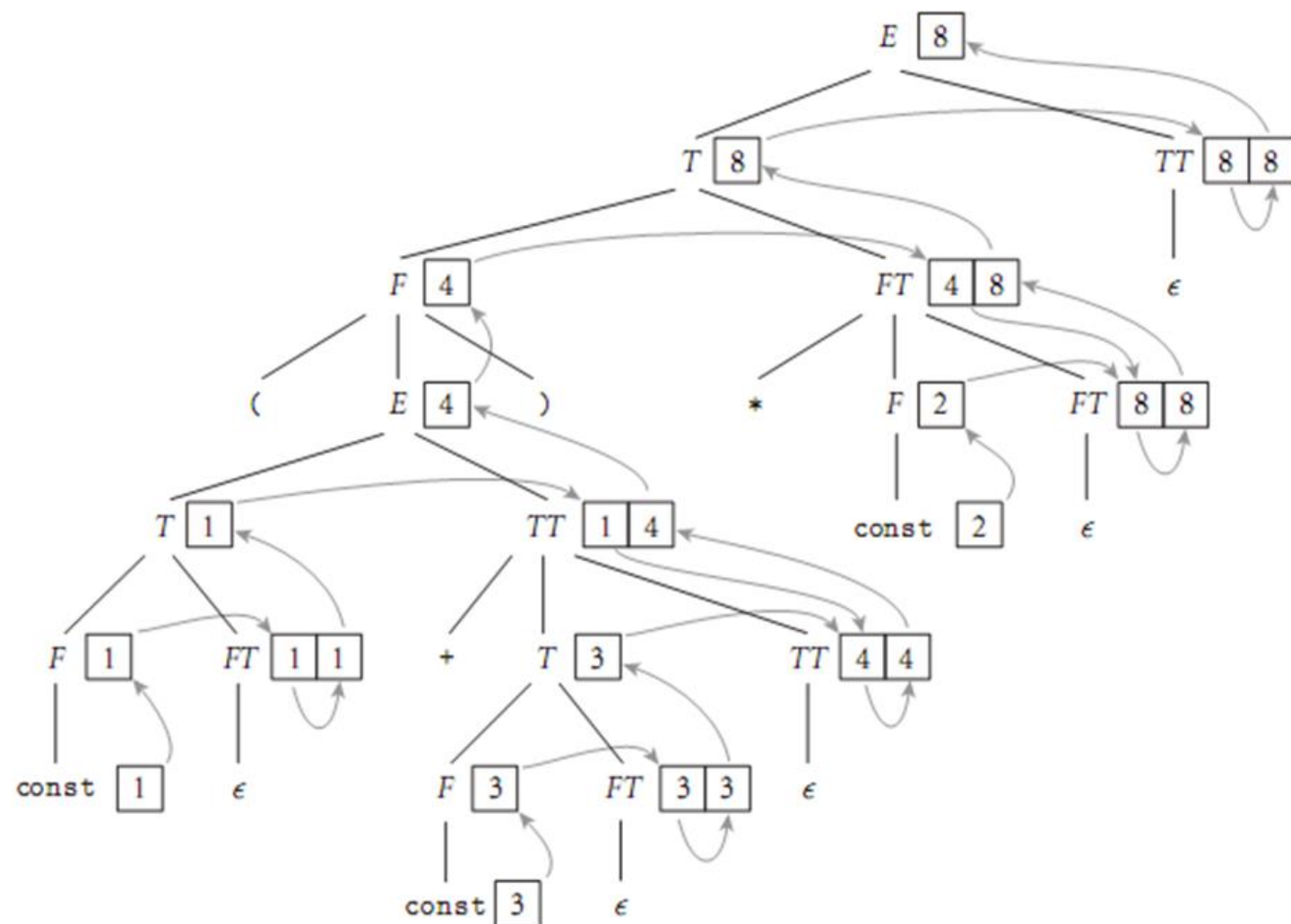


Figure 4.4 Decoration of a top-down parse tree for $(1 + 3) * 2$, using the AG of Figure 4.3. Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right.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

Evaluating Attributes

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

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

Attribute Flow and Syntax Trees

SECTION 4

Attribute Flow

- An attribute flow algorithm propagates attribute values through the parse tree by traversing the tree according to the set and used dependencies between attributes (an attribute must be set before it is read)

Design for Evaluating Attributes

1. Production Rules (CFG)
2. Action (S-Attributed G/L-Attributed G)
3. Attribute Flow (Algorithm)

Production Action	Attribute flow
$\langle E \rangle \rightarrow \langle T \rangle \langle TT \rangle$ $TT.st := T.val$	
$\langle TT_1 \rangle \rightarrow + \langle T \rangle \langle TT_2 \rangle$ $TT_2.st := TT_1.st + T.val$	
$\langle TT \rangle \rightarrow \epsilon$ $TT.val := TT.st$	
$\langle TT_1 \rangle \rightarrow + \langle T \rangle \langle TT_2 \rangle$ $TT_1.val := TT_2.val$	
$\langle E \rangle \rightarrow \langle T \rangle \langle TT \rangle$ $E.val := TT.val$	

Attribute Flow

- Both **context-free grammar** and **attribute grammar** do not specify the order in which attribute rules should be invoked.
 - Both notations are declarative: they define a set of valid trees, but they don't say how to build or decorate them.
 - The order in which attribute rules are listed for a given production is immaterial; **attribute flow** may require them to execute in any order.
 - If, in Figure 4.3, we were to reverse the order in which the rules appear in productions 1, 2, 3, 5, 6, and/or 7 (listing the rule for **symbol.val** first), it would be a purely cosmetic change; the grammar would not be altered.

Attribute Flow

- An algorithm that decorates parse trees by invoking the rules of an attribute grammar in an order consistent with the tree's attribute flow is called a **translation scheme**.
- The simplest scheme is one that makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, and stopping when it completes a pass in which no values change. Such a scheme is said to be **oblivious**, in the sense that it exploits no special knowledge of either the parse tree or the grammar.
- It will halt only if the grammar is well defined. Better performance, at least for noncircular grammars, may be achieved by a dynamic scheme that tailors the evaluation order to the structure of a given parse tree, for example by constructing a topological sort of the attribute flow graph and then invoking rules in an order consistent with the sort.

Types of compilers

- ▶ One pass compilers
- ▶ Multi pass compilers
- ▶ Load and go compilers
- ▶ Optimizing compilers

One-Pass Compilers

- A compiler that interleaves **semantic analysis** and **code generation** with **parsing** is said to be a one-pass compiler.
- It is unclear whether interleaving **semantic analysis** with parsing makes a compiler simpler or more complex; it's mainly a matter of taste. If intermediate code generation is interleaved with parsing, one need not build a syntax tree at all (unless of course the syntax tree is the intermediate code).

One-Pass Compilers

- Moreover, it is often possible to write the intermediate code to an output file on the fly, rather than accumulating it in the attributes of the root of the parse tree. The resulting space savings were important for previous generations of computers, which had very small main memories.
- On the other hand, **semantic analysis** is easier to perform during a separate traversal of a syntax tree, because that tree reflects the program's semantic structure better than the parse tree does, especially with a top-down parser, and because one has the option of traversing the tree in an order other than that chosen by the parser.

Bottom-UP and Top-Down AGs to build a Syntax Tree

- If we choose not to interleave parsing and semantic analysis, we still need to add attribute rules to the context-free grammar, but they serve only to create the syntax tree—not to enforce semantic rules or generate code. [\[gcc\]](#)
- **Figures 4.5** and **4.6** contain bottom-up and top-down attribute grammars, respectively, to build a syntax tree for constant expressions. The attributes in these grammars hold neither numeric values nor target code fragments; instead they point to nodes of the syntax tree. Function `make leaf` returns a pointer to a newly allocated syntax tree node containing the value of a constant. Functions `make un op` and `make bin op` return pointers to newly allocated syntax tree nodes containing a unary or binary operator, respectively, and pointers to the supplied operand(s).
- **Figures 4.7** and **4.8** show stages in the decoration of parse trees for $(1 + 3) * 2$, using the grammars of **Figures 4.5** and **4.6**, respectively.

Note that the final syntax tree is the same in each case.

$$\begin{aligned}
 E_1 &\longrightarrow E_2 + T \\
 &\triangleright E_1.\text{ptr} := \text{make_bin_op}("+", E_2.\text{ptr}, T.\text{ptr}) \\
 E_1 &\longrightarrow E_2 - T \\
 &\triangleright E_1.\text{ptr} := \text{make_bin_op}("-", E_2.\text{ptr}, T.\text{ptr}) \\
 E &\longrightarrow T \\
 &\triangleright E.\text{ptr} := T.\text{ptr} \\
 T_1 &\longrightarrow T_2 * F \\
 &\triangleright T_1.\text{ptr} := \text{make_bin_op}("×", T_2.\text{ptr}, F.\text{ptr}) \\
 T_1 &\longrightarrow T_2 / F \\
 &\triangleright T_1.\text{ptr} := \text{make_bin_op}("÷", T_2.\text{ptr}, F.\text{ptr}) \\
 T &\longrightarrow F \\
 &\triangleright T.\text{ptr} := F.\text{ptr} \\
 F_1 &\longrightarrow - F_2 \\
 &\triangleright F_1.\text{ptr} := \text{make_un_op}("+/-", F_2.\text{ptr}) \\
 F &\longrightarrow (E) \\
 &\triangleright F.\text{ptr} := E.\text{ptr} \\
 F &\longrightarrow \text{const} \\
 &\triangleright F.\text{ptr} := \text{make_leaf}(\text{const.val})
 \end{aligned}$$

Figure 4.5 Bottom-up (S-attributed) attribute grammar to construct a syntax tree.

The symbol +/- is used (as it is on calculators) to indicate change of sign.

$$\begin{aligned}
 E &\longrightarrow T \ TT \\
 &\triangleright TT.st := T.ptr \\
 &\triangleright E.ptr := TT.ptr \\
 TT_1 &\longrightarrow + \ T \ TT_2 \\
 &\triangleright TT_2.st := \text{make_bin_op}("+", TT_1.st, T.ptr) \\
 &\triangleright TT_1.ptr := TT_2.ptr \\
 TT_1 &\longrightarrow - \ T \ TT_2 \\
 &\triangleright TT_2.st := \text{make_bin_op}("-", TT_1.st, T.ptr) \\
 &\triangleright TT_1.ptr := TT_2.ptr \\
 TT &\longrightarrow \epsilon \\
 &\triangleright TT.ptr := TT.st \\
 T &\longrightarrow F \ FT \\
 &\triangleright FT.st := F.ptr \\
 &\triangleright T.ptr := FT.ptr \\
 FT_1 &\longrightarrow * \ F \ FT_2 \\
 &\triangleright FT_2.st := \text{make_bin_op}("x", FT_1.st, F.ptr) \\
 &\triangleright FT_1.ptr := FT_2.ptr \\
 FT_1 &\longrightarrow / \ F \ FT_2 \\
 &\triangleright FT_2.st := \text{make_bin_op}("/\div", FT_1.st, F.ptr) \\
 &\triangleright FT_1.ptr := FT_2.ptr \\
 FT &\longrightarrow \epsilon \\
 &\triangleright FT.ptr := FT.st \\
 F_1 &\longrightarrow - \ F_2 \\
 &\triangleright F_1.ptr := \text{make_un_op}("+/_ ", F_2.ptr) \\
 F &\longrightarrow (\ E \) \\
 &\triangleright F.ptr := E.ptr \\
 F &\longrightarrow \text{const} \\
 &\triangleright F.ptr := \text{make_leaf}(\text{const.val})
 \end{aligned}$$

Figure 4.6 Top-down (L-attributed) attribute grammar to construct a syntax tree.

Here the **st** attribute, like the **ptr** attribute (and unlike the st attribute of Figure 4.3), is a pointer to a syntax tree node.

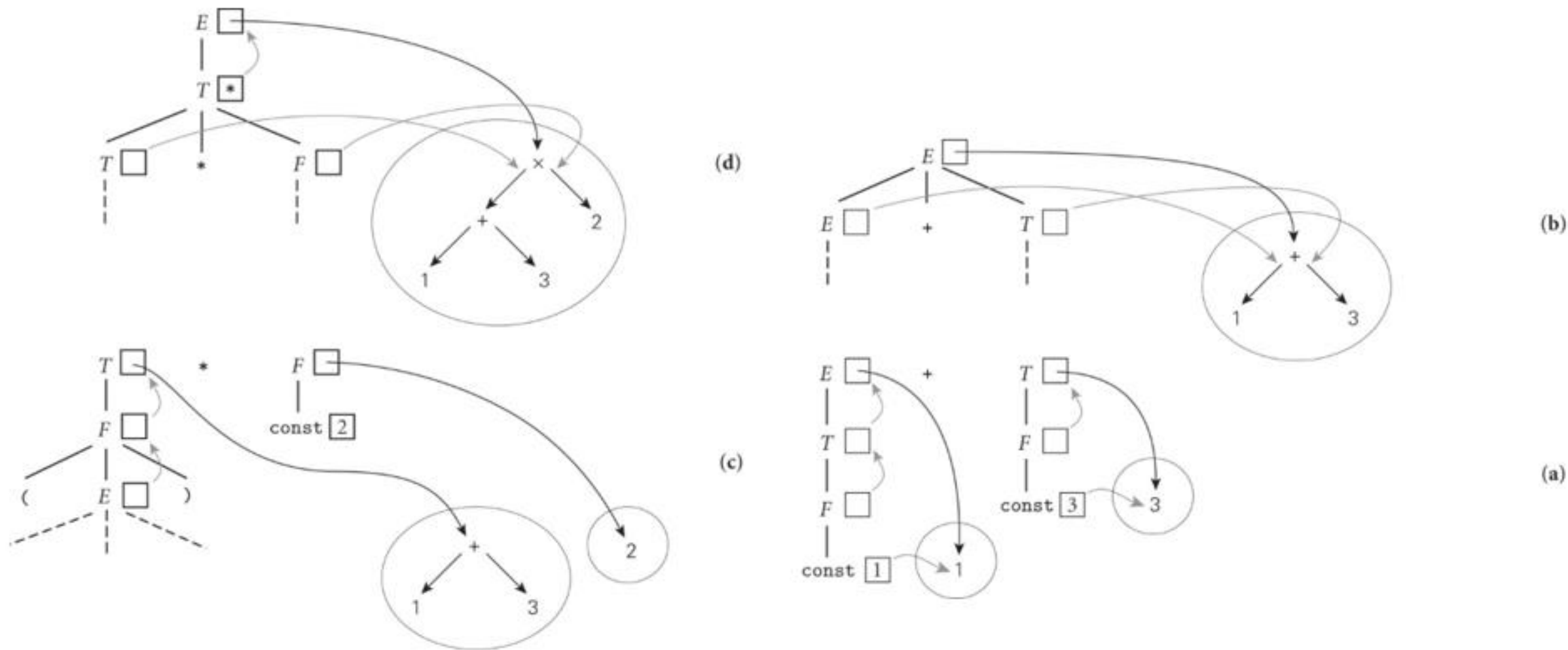


Figure 4.7 Construction of a syntax tree for $(1 + 3) * 2$ via decoration of a bottom-up parse tree, using the grammar of Figure 4.5. This figure reads from bottom to top. In diagram (a), the values of the constants 1 and 3 have been placed in new syntax tree leaves. Pointers to these leaves propagate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

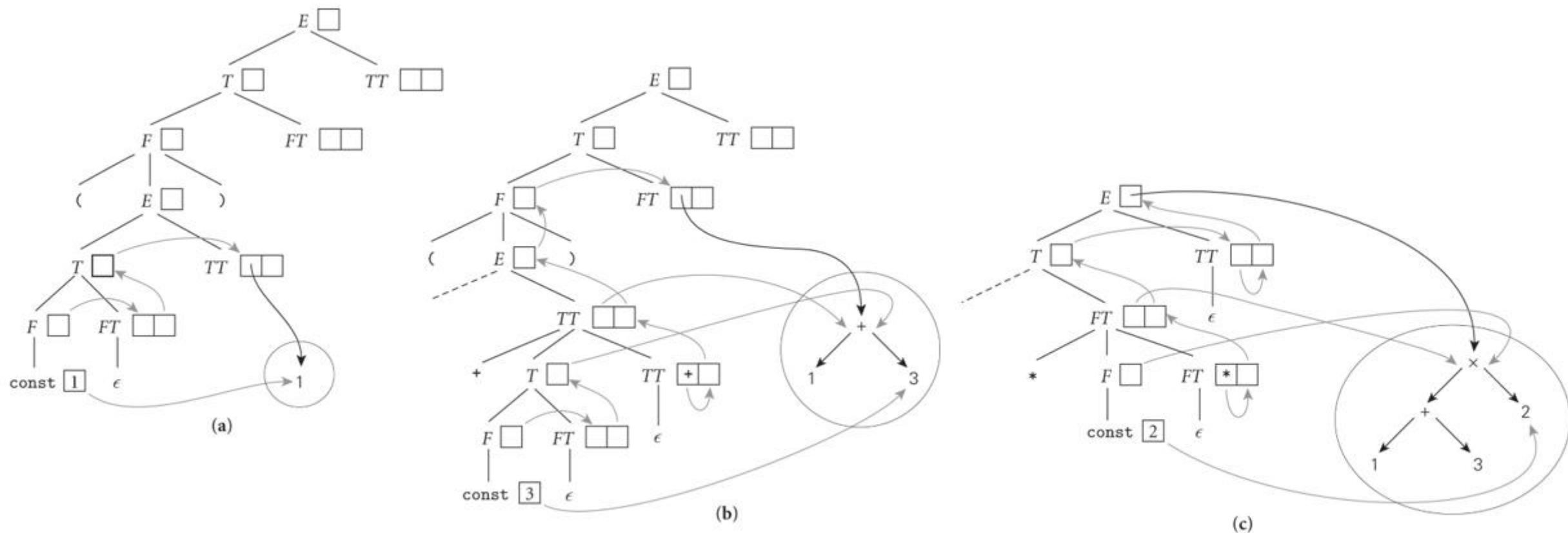


Figure 4.8 Construction of a syntax tree via decoration of a top-down parse tree, using the grammar of Figure 4.6. In the top diagram, (a), the value of the constant 1 has been placed in a new syntax tree leaf. A pointer to this leaf then propagates to the st attribute of TT . In (b), a second leaf has been created to hold the constant 3. Pointers to the two leaves then become child pointers 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. Pointers to this leaf and to the $+$ node then become the children of a new \times 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

SECTION 5

Overview of Action Routines

- There are automatic tools that will construct a **semantic analyzer (attribute evaluator)** for a given attribute grammar.
- **Applications:** Attribute evaluator generators have been used in **syntax-based editors** [RT88], **incremental compilers** [SDB84], and various aspects of programming language research.
- Most production compilers, however, use an **ad hoc**, handwritten translation scheme, interleaving parsing with at least the initial construction of a syntax tree, and possibly all of semantic analysis and intermediate code generation. Because they are able to evaluate the attributes of each production as it is parsed, they do not need to build the full parse tree.

Action Routines – $f(X)$

- An **Action Routine** is a **semantic function** that the programmer (grammar writer) instructs the compiler to execute at a particular point in the parse.
- Most parser generators allow the programmer to specify action routines. In an LL parser generator, an action routine can appear anywhere within a **right-hand side**.
- A routine at the beginning of a right-hand side will be called as soon as the parser predicts the production. A routine embedded in the middle of a right-hand side will be called as soon as the parser has matched the symbol to the left.

Action Routines – $f(X)$

- The implementation mechanism is simple: when it predicts a production, the parser pushes all of the right-hand side onto the stack, including terminals (to be matched), non-terminals (to drive future predictions), and pointers to action routines.
- When it finds a pointer to an action routine at the top of the parse stack, the parser simply calls it, passing the appropriate attributes as arguments.

Top-down Action Routines to Build a Syntax Tree

- To make this process more concrete, consider again our **LL(1)** grammar for constant expressions.
- Action routines to build a syntax tree while parsing this grammar appear in Figure 4.9. The only difference between this grammar and the one in Figure 4.6 is that the action routines (delimited here with curly braces) are embedded among the symbols of the right-hand sides; the work performed is the same.
- The ease with which the attribute grammar can be transformed into the grammar with action routines is due to the fact that the attribute grammar is **L-attributed**. If it required more complicated flow, we would not be able to cast it in the form of action routines.

```

E → T { TT.st := T.ptr } TT { E.ptr := TT.ptr }
TT1 → + T { TT2.st := make_bin_op("+", TT1.st, T.ptr) } TT2 { TT1.ptr := TT2.ptr }
TT1 → - T { TT2.st := make_bin_op("-", TT1.st, T.ptr) } TT2 { TT1.ptr := TT2.ptr }
TT → ε { TT.ptr := TT.st }
T → F { FT.st := F.ptr } FT { T.ptr := FT.ptr }
FT1 → * F { FT2.st := make_bin_op("×", FT1.st, F.ptr) } FT2 { FT1.ptr := FT2.ptr }
FT1 → / F { FT2.st := make_bin_op("÷", FT1.st, F.ptr) } FT2 { FT1.ptr := FT2.ptr }
FT → ε { FT.ptr := FT.st }
F1 → - F2 { F1.ptr := make_un_op("+/-", F2.ptr) }
F → ( E ) { F.ptr := E.ptr }
F → const { F.ptr := make_leaf(const.ptr) }

```

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

Recursive Descent and Action Routines

- As in ordinary parsing, there is a strong analogy between recursive descent and table-driven parsing with action routines. Figure 4.10 shows the **term_tail** routine from Figure 2.16 (page 74), modified to do its part in constructing a syntax tree.
- The behavior of this routine mirrors that of productions 2 through 5 in Figure 4.9. The routine accepts as a parameter a pointer to the syntax tree fragment contained in the attribute grammar's **TT1**.
- Then, given an upcoming **+** or **-** symbol on the input, it
 1. calls add op to parse that symbol (returning a character string representation);
 2. calls term to parse the attribute grammar's **T**;
 3. calls make bin op to create a new tree node;
 4. passes that node to **term_tail**, which parses the attribute grammar's **TT2**; and
 5. returns the result.

$$\begin{aligned}
E &\longrightarrow T \{ \text{TT.st} := \text{T.ptr} \} \quad TT \{ \text{E.ptr} := \text{TT.ptr} \} \\
TT_1 &\longrightarrow + T \{ \text{TT}_2.\text{st} := \text{make_bin_op}("+", \text{TT}_1.\text{st}, \text{T.ptr}) \} \quad TT_2 \{ \text{TT}_1.\text{ptr} := \text{TT}_2.\text{ptr} \} \\
TT_1 &\longrightarrow - T \{ \text{TT}_2.\text{st} := \text{make_bin_op}("-", \text{TT}_1.\text{st}, \text{T.ptr}) \} \quad TT_2 \{ \text{TT}_1.\text{ptr} := \text{TT}_2.\text{ptr} \} \\
TT &\longrightarrow \epsilon \{ \text{TT.ptr} := \text{TT.st} \} \\
T &\longrightarrow F \{ \text{FT.st} := \text{F.ptr} \} \quad FT \{ \text{T.ptr} := \text{FT.ptr} \} \\
FT_1 &\longrightarrow * F \{ \text{FT}_2.\text{st} := \text{make_bin_op}("x", \text{FT}_1.\text{st}, \text{F.ptr}) \} \quad FT_2 \{ \text{FT}_1.\text{ptr} := \text{FT}_2.\text{ptr} \} \\
FT_1 &\longrightarrow / F \{ \text{FT}_2.\text{st} := \text{make_bin_op}("/", \text{FT}_1.\text{st}, \text{F.ptr}) \} \quad FT_2 \{ \text{FT}_1.\text{ptr} := \text{FT}_2.\text{ptr} \} \\
FT &\longrightarrow \epsilon \{ \text{FT.ptr} := \text{FT.st} \} \\
F_1 &\longrightarrow - F_2 \{ \text{F}_1.\text{ptr} := \text{make_un_op}("+/-", \text{F}_2.\text{ptr}) \} \\
F &\longrightarrow (E) \{ \text{F.ptr} := \text{E.ptr} \} \\
F &\longrightarrow \text{const} \{ \text{F.ptr} := \text{make_leaf}(\text{const.ptr}) \}
\end{aligned}$$

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

```

procedure term_tail(lhs : tree_node_ptr)
  case input_token of
    +, - :
      op : string := add_op
      return term_tail(make_bin_op(op, lhs, term))
      -- term is a recursive call with no arguments
    ), id, read, write, $$ :      -- epsilon production
      return lhs
    otherwise parse_error

```

Figure 4.10 Recursive descent parsing with embedded “action routines.” Compare to the routine with the same name in Figure 2.16 (page 74) and with productions 2 through 5 in Figure 4.9.

Bottom-Up Evaluation

- In an **LR parser generator**, one cannot in general embed action routines at arbitrary places in a right-hand side, since the parser does not in general know what production it is in until it has seen all or most of the yield.
- **LR parser generators** therefore permit action routines only after the point at which the production being parsed can be identified unambiguously (this is known as the trailing part of the right-hand side; the ambiguous part is the left corner).

Bottom-Up Evaluation

- If the attribute flow of the action routines is strictly bottom-up (as it is in an **S-attributed** attribute grammar), then execution at the end of right-hand sides is all that is needed. The **attribute grammars** of Figures 4.1 and 4.5, in fact, are essentially identical to the action routine versions.
- If the action routines are responsible for a significant part of semantic analysis, however (as opposed to simply building a syntax tree), then they will often need contextual information in order to do their job. To obtain and use this information in an **LR** parse, they will need some (necessarily limited) access to inherited attributes or to information outside the current production.

Space Management for Attributes

- Any attribute evaluation method requires space to hold the attributes of the grammar symbols.
- The details differ in bottom-up and top-down parsers.
- For a bottom-up parser with an **S-attributed** grammar, the obvious approach is to maintain an attribute stack that directly mirrors the parse stack: next to every state number on the parse stack is an attribute record for the symbol we shifted when we entered that state. Entries in the attribute stack are pushed and popped automatically by the parser driver; **space management is not an issue for the writer of action routines.**
- For a top-down parser with an **L-attributed** grammar, we have two principal options. The first option is automatic, but more complex than for bottom-up grammars. It still uses an attribute stack, but one that does not mirror the parse stack. The second option has lower space overhead, and saves time by “shortcutting” copy rules, but requires action routines to allocate and deallocate space for attributes explicitly.

Tree Grammars and Syntax Tree Decoration I

Language

SECTION 6

Bottom-up CFG for Calculator Language with Types

Figure 4.11 contains a bottom-up **CFG** for a calculator language with types and declarations. The grammar differs from that of Example 2.37 in three ways:

1. we allow declarations to be intermixed with statements,
2. we differentiate between integer and real constants (presumably the latter contain a decimal point), and
3. we require explicit conversions between integer and real operands.

The intended semantics of our language requires that every identifier be declared before it is used, and that types not be mixed in computations.

$$\begin{aligned}
\text{program} &\longrightarrow \text{stmt_list } \$\$ \\
\text{stmt_list} &\longrightarrow \text{stmt_list decl} \mid \text{stmt_list stmt} \mid \epsilon \\
\text{decl} &\longrightarrow \text{int id} \mid \text{real id} \\
\text{stmt} &\longrightarrow \text{id} := \text{expr} \mid \text{read id} \mid \text{write expr} \\
\text{expr} &\longrightarrow \text{term} \mid \text{expr add_op term} \\
\text{term} &\longrightarrow \text{factor} \mid \text{term mult_op factor} \\
\text{factor} &\longrightarrow (\text{expr}) \mid \text{id} \mid \text{int_const} \mid \text{real_const} \mid \\
&\quad \text{float} (\text{expr}) \mid \text{trunc} (\text{expr}) \\
\text{add_op} &\longrightarrow + \mid - \\
\text{mult_op} &\longrightarrow * \mid /
\end{aligned}$$

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.

Syntax Tree to Average an Integer and a Real

- Extrapolating from the example in Figure 4.5, it is easy to add semantic functions or action routines to the grammar of Figure 4.11 to construct a syntax tree for the calculator language.
- The obvious structure for such a tree would represent expressions as we did in Figure 4.7, and would represent a program as a linked list of declarations and statements.
- As a concrete example, Figure 4.12 contains the syntax tree for a simple program to print the average of an integer and a real.

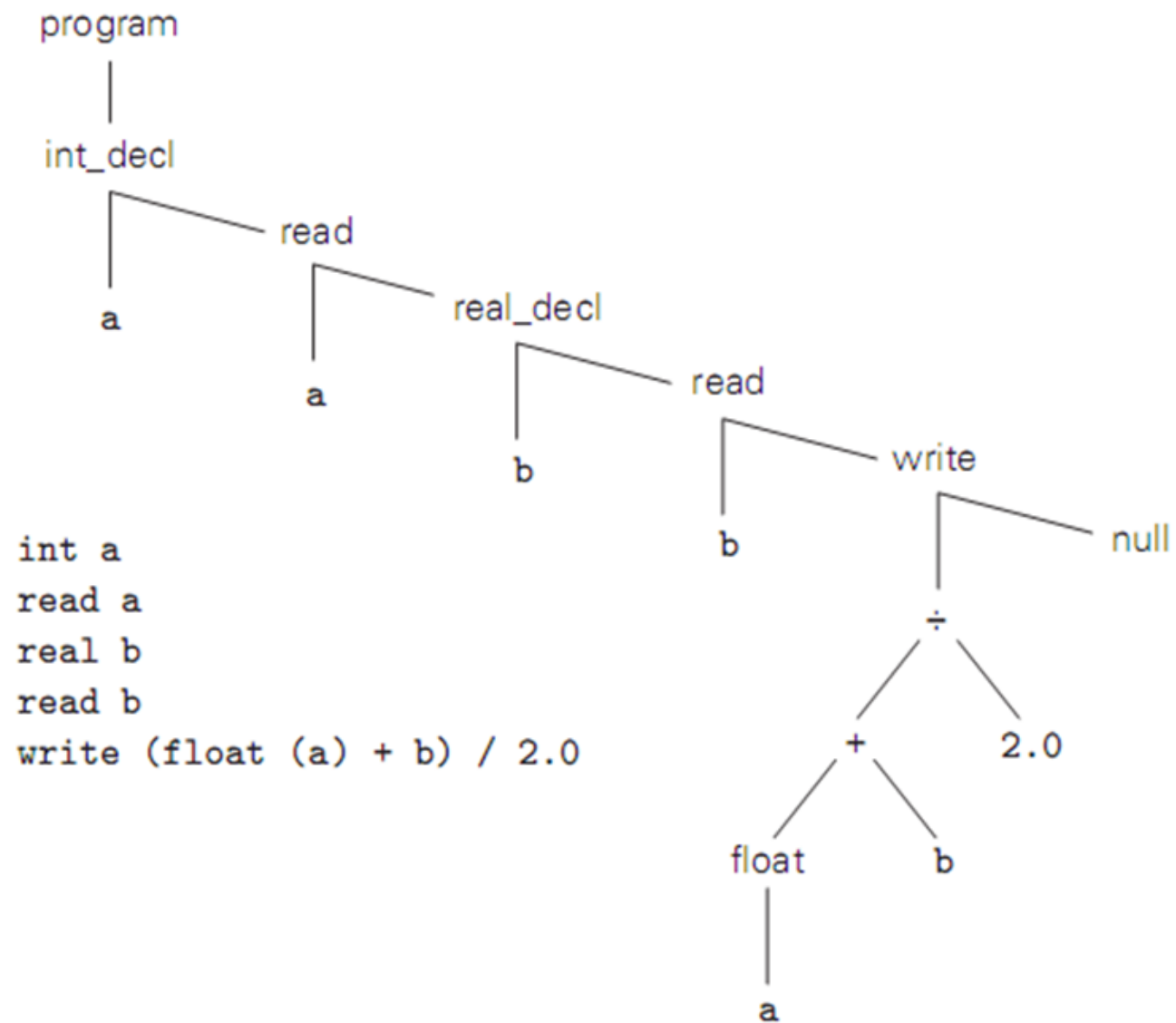


Figure 4.12 Syntax tree for a simple calculator program.

Note

This example is used as the example to conclude our discussion in Unit 1: Foundations. (Chapter 1 – Chapter 4)

This Unit 1, basically, introduces the construction of the compiler front-end of this **Calculator Language** project.

The following lecture will have its complete decorated AG (Attributed AST) design for this Calculator Language.

Tree Grammars and Syntax Tree Decoration II

Example

SECTION 6

Tree Grammar for the Calculator Language with Types

- Much as a context-free grammar describes the possible structure of parse trees for a given programming language, we can use a tree grammar to represent the possible structure of syntax trees.
- As in a **CFG**, each production of a tree grammar represents a possible relationship between a parent and its children in the tree.
- The parent is the symbol on the left-hand side of the production; the children are the symbols on the right-hand side. The productions used in Figure 4.12 might look something like the following.

$program \longrightarrow item$
 $int_decl : item \longrightarrow id\ item$
 $read : item \longrightarrow id\ item$
 $real_decl : item \longrightarrow id\ item$
 $write : item \longrightarrow expr\ item$
 $null : item \longrightarrow \epsilon$
 $'\div' : expr \longrightarrow expr\ expr$
 $'+' : expr \longrightarrow expr\ expr$
 $float : expr \longrightarrow expr$
 $id : expr \longrightarrow \epsilon$
 $real_const : expr \longrightarrow \epsilon$

Tree Grammar for the Calculator Language with Types

- The notation **A** : **B** on the left-hand side of a production means that
 1. **A** is one variant of **B**, and
 2. **A** may appear anywhere a **B** is expected on a right-hand side.
- **Tree grammars** and **context-free grammars** differ in important ways.
 - A **context free grammar** is meant to **generate** a language composed of strings of tokens, where each string is the **yield** of a parse tree.
 - Parsing is the process of finding a tree that has a given yield. A tree grammar, as we use it here, is meant to generate the trees themselves. We have no need for a notion of parsing: we can easily inspect a tree and determine whether and how it can be generated by the grammar.

Tree Grammar for the Calculator Language with Types

- Our purpose in introducing tree grammars is to provide a framework for the decoration of syntax trees. Semantic rules attached to the productions of a tree grammar can be used to define the attribute flow of a syntax tree in exactly the same way that semantic rules attached to the productions of a context-free grammar are used to define the attribute flow of a parse tree.
- We will use a tree grammar in the remainder of this section to perform static semantic checking.

Definition of the Terms in Figures

Each Node keep track of the following data:

name: (token name)

type: (token type)

syntab: reference to the symbol table.

errors_in, errors_out, errors: error message for error propagation.

Inherited means a attribute from parent or left siblings.

Synthesized means from lower level of the tree. (By bottom-up parsing)

Attributes	
Inherited	Synthesized
—	location, errors
syntab, errors_in	location, errors_out
syntab	location, type, errors, name (<i>id</i> only)

Shorthand symbol in the Figures:

ei = errors_in
eo = errors_out
e = errors
s = syntab
t = type
n = name

location attribute not shown

Tree AG for the Calculator Language with Types

A complete tree attribute grammar for our calculator language with types can be constructed using the **node classes**, **variants**, and **attributes** shown in Figure 4.13.

The grammar itself appears in Figure 4.14. Once decorated, the program node at the root of the syntax tree will contain a list, in a **synthesized attribute**, of all static semantic errors in the program.

Each **item** or **expr** node has an inherited attribute **syntab** that contains a list, with types, of all identifiers declared to the left in the tree. Each **item** node also has an **inherited attribute errors** in that lists all static semantic errors found to its left in the tree, and a synthesized attribute errors out to propagate the final error list back to the root. Each expr node has one synthesized attribute that indicates its type and another that contains a list of any static semantic errors found inside.

Tree AG for the Calculator Language with Types

- Our handling of **semantic errors** illustrates a common technique:
 - In order to continue looking for other errors we must provide values for any attributes that would have been set in the absence of an error.
- To avoid cascading error messages, we choose values for those attributes that will pass quietly through subsequent checks.
- In this specific case we employ a pseudo type called **error**, which we associate with any symbol table entry or expression for which we have already generated a message.

Tree AG for the Calculator Language with Types

- Though it takes a bit of checking to verify the fact, our attribute grammar is noncircular and well defined. No attribute is ever assigned a value more than once.
- **Note:** The helper routines at the end of Figure 4.14 should be thought of as macros, rather than semantic functions. For the sake of brevity we have passed them entire tree nodes as arguments. Each macro calculates the values of two different attributes. Under a strict formulation of attribute grammars each macro would be replaced by two separate semantic functions, one per calculated attribute.

Class of node	Variants	Attributes	
		Inherited	Synthesized
<i>program</i>	—	—	location, errors
<i>item</i>	<i>int_decl, real_decl,</i> <i>read, write, :=, null</i>	symtab, errors_in	location, errors_out
<i>expr</i>	<i>int_const, real_const,</i> <i>id, +, −, ×, ÷,</i> <i>float, trunc</i>	symtab	location, type, errors, name (<i>id</i> only)

Figure 4.13 Classes of nodes for the syntax tree attribute grammar of [Figure 4.14](#). With the exception of name, all variants of a given class have all the class's attributes.

```

program  $\rightarrow$  item
   $\triangleright$  item.symtab := null
   $\triangleright$  program.errors := item.errors_out
   $\triangleright$  item.errors_in := null

int_decl : item1  $\rightarrow$  id item2
   $\triangleright$  declare_name(id, item1, item2, int)
   $\triangleright$  item1.errors_out := item2.errors_out

real_decl : item1  $\rightarrow$  id item2
   $\triangleright$  declare_name(id, item1, item2, real)
   $\triangleright$  item1.errors_out := item2.errors_out

read : item1  $\rightarrow$  id item2
   $\triangleright$  item2.symtab := item1.symtab
   $\triangleright$  if (id.name, ?)  $\in$  item1.symtab
    item2.errors_in := item1.errors_in
  else
    item2.errors_in := item1.errors_in + [id.name "undefined at" id.location]
   $\triangleright$  item1.errors_out := item2.errors_out

write : item1  $\rightarrow$  expr item2
   $\triangleright$  expr.symtab := item1.symtab
   $\triangleright$  item2.symtab := item1.symtab
   $\triangleright$  item2.errors_in := item1.errors_in + expr.errors
   $\triangleright$  item1.errors_out := item2.errors_out

':=' : item1  $\rightarrow$  id expr item2
   $\triangleright$  expr.symtab := item1.symtab
   $\triangleright$  item2.symtab := item1.symtab
   $\triangleright$  if (id.name, A)  $\in$  item1.symtab      -- for some type A
    if A  $\neq$  error and expr.type  $\neq$  error and A  $\neq$  expr.type
      item2.errors_in := item1.errors_in + ["type clash at" item1.location]
    else
      item2.errors_in := item1.errors_in + expr.errors
  else
    item2.errors_in := item1.errors_in + [id.name "undefined at" id.location] + expr.errors
   $\triangleright$  item1.errors_out := item2.errors_out

null : item  $\rightarrow$   $\epsilon$ 
   $\triangleright$  item.errors_out := item.errors_in

```

Figure 4.14 Attribute grammar to decorate an abstract syntax tree for the calculator language with types. We use square brackets to delimit error messages and pointed brackets to delimit symbol table entries. Juxtaposition indicates concatenation within error messages; the '+' and '-' operators indicate insertion and removal in lists. We assume that every node has been initialized by the scanner or by action routines in the parser to contain an indication of the location (line and column) at which the corresponding construct appears in the source (see Exercise 4.22). The '?' symbol is used as a "wild card"; it matches any type. (continued)

$id : expr \rightarrow \epsilon$

- ▷ if $\langle id.name, A \rangle \in expr.symtab$ -- for some type A
 $expr.errors := null$
 $expr.type := A$
- else
 $expr.errors := [id.name \text{ "undefined at" } id.location]$
 $expr.type := error$

$int_const : expr \rightarrow \epsilon$

- ▷ $expr.type := int$

$real_const : expr \rightarrow \epsilon$

- ▷ $expr.type := real$

$'+' : expr_1 \rightarrow expr_2 expr_3$

- ▷ $expr_2.symtab := expr_1.symtab$
- ▷ $expr_3.symtab := expr_1.symtab$
- ▷ $check_types(expr_1, expr_2, expr_3)$

$'-' : expr_1 \rightarrow expr_2 expr_3$

- ▷ $expr_2.symtab := expr_1.symtab$
- ▷ $expr_3.symtab := expr_1.symtab$
- ▷ $check_types(expr_1, expr_2, expr_3)$

$'\times' : expr_1 \rightarrow expr_2 expr_3$

- ▷ $expr_2.symtab := expr_1.symtab$
- ▷ $expr_3.symtab := expr_1.symtab$
- ▷ $check_types(expr_1, expr_2, expr_3)$

$'\div' : expr_1 \rightarrow expr_2 expr_3$

- ▷ $expr_2.symtab := expr_1.symtab$
- ▷ $expr_3.symtab := expr_1.symtab$
- ▷ $check_types(expr_1, expr_2, expr_3)$

$float : expr_1 \rightarrow expr_2$

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

$trunc : expr_1 \rightarrow expr_2$

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

Figure 4.14: (Continued)

```

macro declare_name(id, cur_item, next_item : syntax_tree_node; t : type)
    if ⟨id.name, ?⟩ ∈ cur_item.symtab
        next_item.errors_in := cur_item.errors_in + ["redefinition of" id.name "at" cur_item.location]
        next_item.symtab := cur_item.symtab - ⟨id.name, ?⟩ + ⟨id.name, error⟩
    else
        next_item.errors_in := cur_item.errors_in
        next_item.symtab := cur_item.symtab + ⟨id.name, t⟩

macro check_types(result, operand1, operand2)
    if operand1.type = error or operand2.type = error
        result.type := error
        result.errors := operand1.errors + operand2.errors
    else if operand1.type ≠ operand2.type
        result.type := error
        result.errors := operand1.errors + operand2.errors + ["type clash at" result.location]
    else
        result.type := operand1.type
        result.errors := operand1.errors + operand2.errors

macro convert_type(old_expr, new_expr : syntax_tree_node; from_t, to_t : type; msg : string)
    if old_expr.type = from_t or old_expr.type = error
        new_expr.errors := old_expr.errors
        new_expr.type := to_t
    else
        new_expr.errors := old_expr.errors + [msg "at" old_expr.location]
        new_expr.type := error

```

Figure 4.14: (Continued)

Decorating a Tree with the AG of The AG Tree Example

- This Calculator Language Project:
 1. Decorated AG: Figure 4.15.
 2. Decorate CFG Grammar: Figure 4.14
 3. Syntax Tree: Figure 4.12.

The pattern of **attribute flow** appears considerably messier than in previous examples in this chapter, but this is simply because type checking is more complicated than calculating constants or building a syntax tree.

Decorating a Tree with the AG of The AG Tree Example

- Symbol table information flows along the chain of **items** and down into **expr** trees. The **int_decl** and **real_decl** nodes add new information; other nodes simply pass the table along. Type information is synthesized at **id : expr** leaves by looking up an identifier's name in the symbol table.
- The information then propagates upward within an expression tree, and is used to type-check operators and assignments.
- Error messages flow along the chain of items via the errors in attributes, and then back to the root via the errors out attributes. Messages also flow up out of **expr** trees. Wherever a type check is performed, the type attribute may be used to help create a new message to be appended to the growing message list.

Decorating a Tree with the AG of The AG Tree Example

- In our example grammar we accumulate error messages into a synthesized attribute of the root of the syntax tree. In an **ad hoc attribute evaluator** we might print these messages on the fly as the errors are discovered. In practice, particularly in a multi-pass compiler, it makes sense to buffer the messages, so they can be interleaved with messages produced by other phases of the compiler, and printed in program order at the end of compilation. [The error message could only be a code on the tree node. e.g. Error 17]
- One could convert our attribute grammar into executable code using an **automatic attribute evaluator generator**. Alternatively, one could create an ad hoc evaluator in the form of mutually recursive subroutines. In the latter case attribute flow would be explicit in the **calling sequence of the routines**.
- We could then choose if desired to keep the symbol table in global variables, rather than passing it from node to node through attributes. Most compilers employ the **ad hoc** approach.

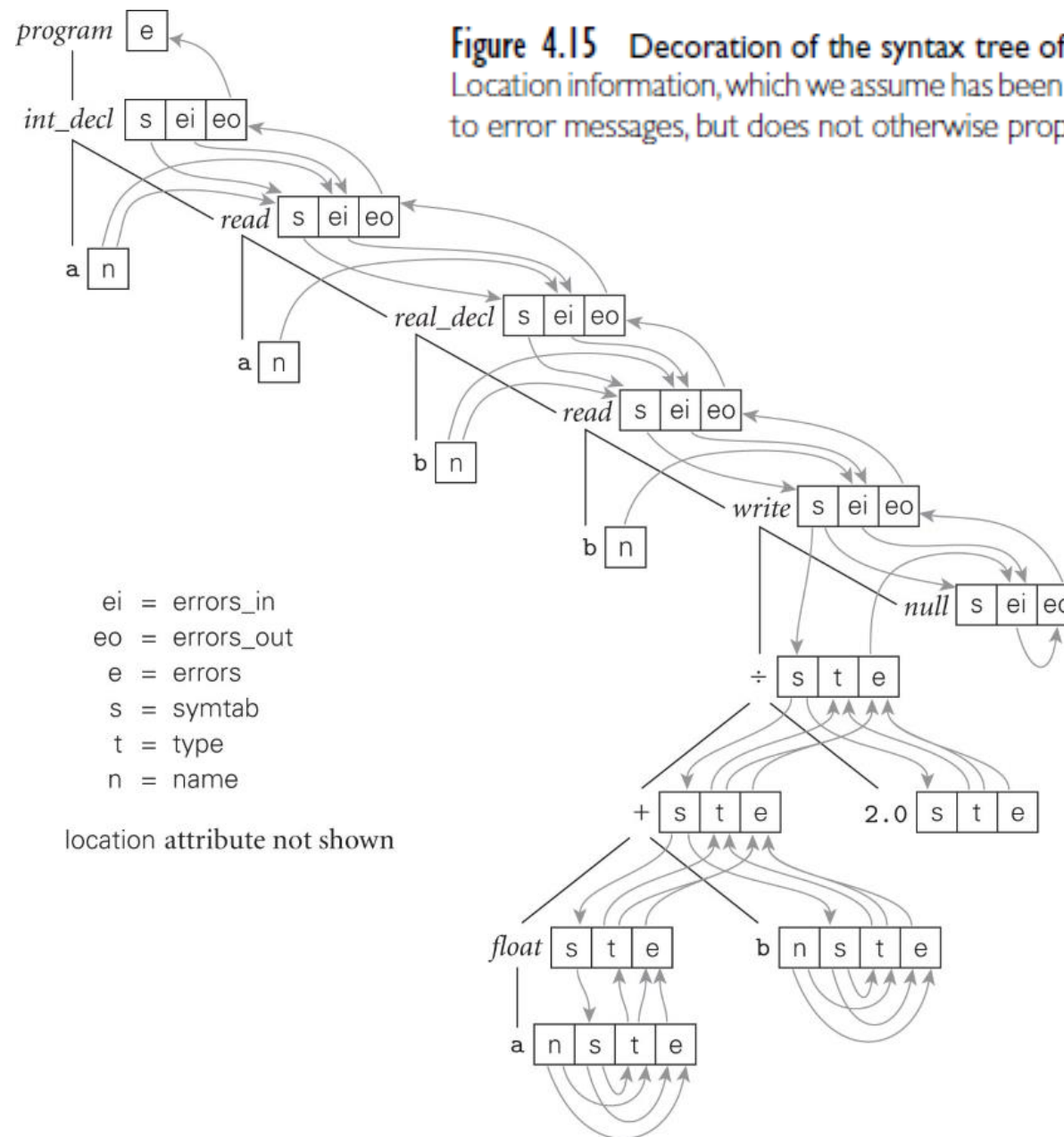


Figure 4.15 Decoration of the syntax tree of Figure 4.12, using the grammar of Figure 4.14. Location information, which we assume has been initialized in every node by the parser, contributes to error messages, but does not otherwise propagate through the tree.

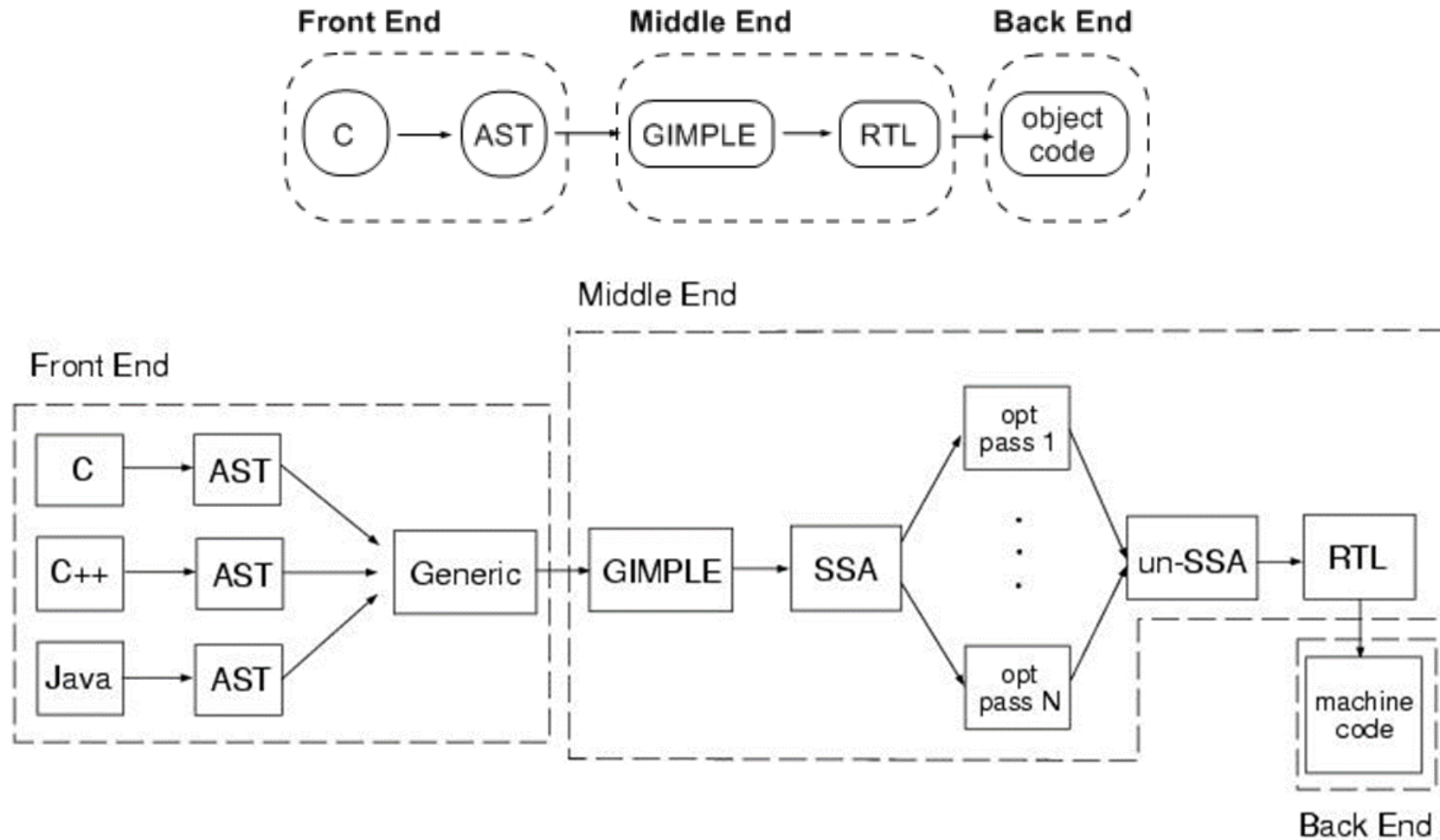
Exit Review of Unit One

SECTION 7

Conclusion

- This chapter has discussed the task of semantic analysis. We reviewed the sorts of language rules that can be classified as **syntax**, **static semantics**, and **dynamic semantics**, and discussed the issue of whether to generate code to perform dynamic semantic checks.
- We also considered the role that the semantic analyzer plays in a typical compiler. We noted that both the enforcement of static semantic rules and the generation of intermediate code can be cast in terms of annotation, or decoration, of a parse tree or syntax tree.
- We then presented **attribute grammars** as a formal framework for this decoration process. [\[One of the Framework\]](#)

Half of the Middle End in gcc is on AG Generation



In Other Chapters

- In subsequent chapters (6–10 in Unit 2) we will consider a wide variety of programming language constructs. Rather than present the actual attribute grammars required to implement these constructs, we will describe their semantics informally, and give examples of the target code.
- We will return to attribute grammars in Chapter 15, when we consider the generation of intermediate code in more detail.

Unit 3 on Programming Paradigms, Unit 4 on Back-End and Programming Environment.

[Unit 3] [Unit 4] will be omitted for 8-week course except chapter 13.