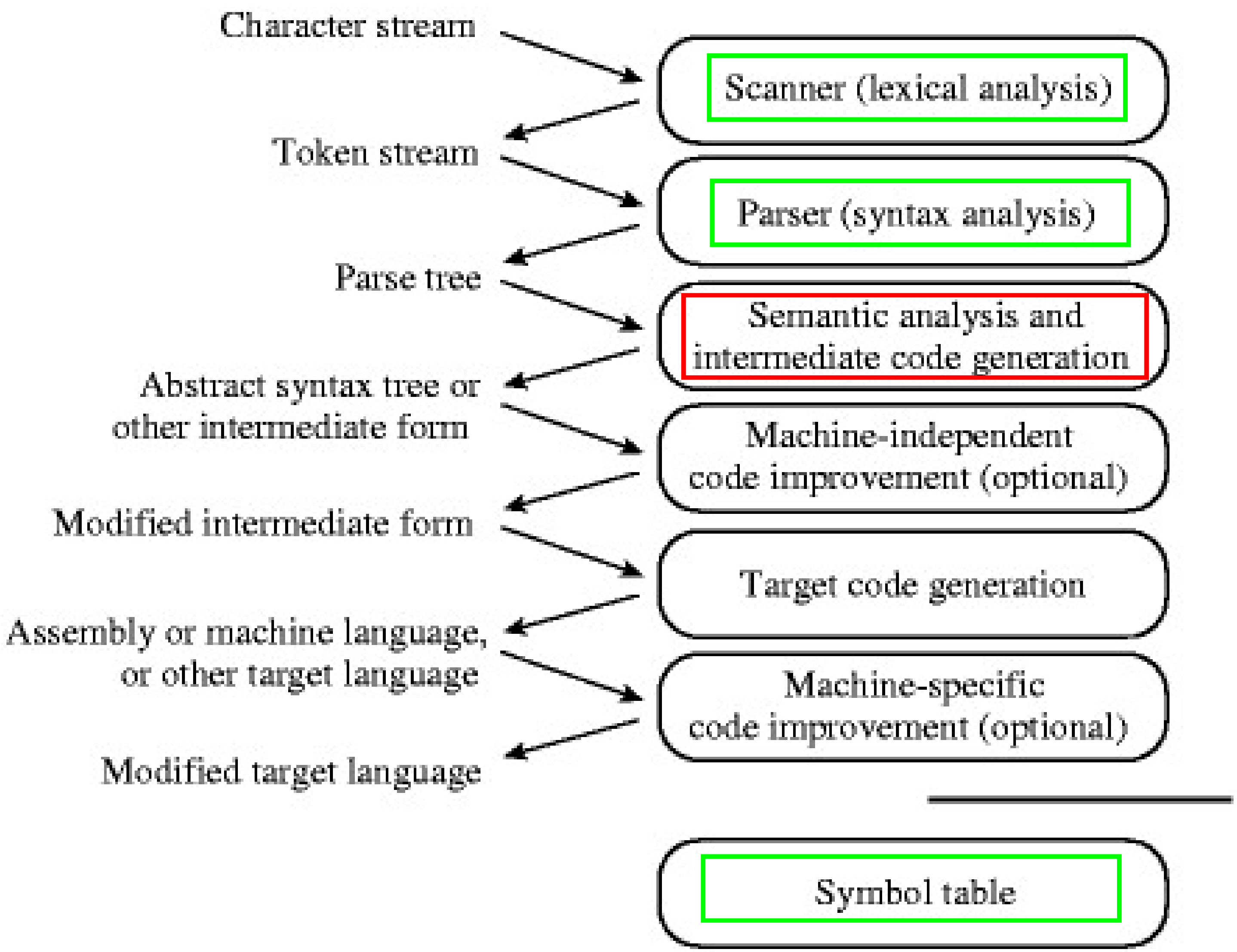
Semantic Analysis

Phases of Compilation



Specification of Programming Languages

- PLs require precise definitions (i.e. no ambiguity)
  - Language *form* (Syntax)
  - **Language *meaning* (Semantics)**
- Consequently, PLs are specified using formal notation:
  - Formal syntax
    - Tokens
    - Grammar
  - **Formal semantics**
    - **Attribute Grammars (static semantics)**
    - **Dynamic Semantics**

The Semantic Analyzer

- The principal job of the semantic analyzer is to enforce static semantic rules.
- In general, anything that requires the requires the compiler to compare things that are separate by a long distance or to count things ends up being a matter of *semantics*.
- The semantic analyzer also commonly constructs a syntax tree (usually first), and much of the information it gathers is needed by the code generator.

Attribute Grammars

- Context-Free Grammars (CFGs) are used to specify the syntax of programming languages
  - *E.g.* arithmetic expressions
- How do we tie these rules to mathematical concepts?
- *Attribute grammars* are annotated CFGs in which *annotations* are used to establish meaning relationships among symbols
  - Annotations are also known as decorations

$$
\begin{array}{lcl}
E & \longrightarrow & E \texttt{ + } T \\
E & \longrightarrow & E \texttt{ - } T \\
E & \longrightarrow & T \\
T & \longrightarrow & T \texttt{ * } F \\
T & \longrightarrow & T \texttt{ / } F \\
T & \longrightarrow & F \\
F & \longrightarrow & \texttt{- } F \\
F & \longrightarrow & \texttt{(} E \texttt{)} \\
F & \longrightarrow & \texttt{const}
\end{array}
$$

Attribute Grammars
Example

- Each grammar symbols has a set of *attributes*
  - *E.g.* the value of E_1 is the attribute E_1.val
- Each grammar rule has a set of rules over the symbol attributes
  - *Copy rules*
  - *Semantic Function rules*
    - » *E.g.* sum, quotient

1: $E_1 \longrightarrow E_2 + T$
  ▷ E_1.val := sum (E_2.val, T.val)

2: $E_1 \longrightarrow E_2 - T$
  ▷ E_1.val := difference (E_2.val, T.val)

3: $E \longrightarrow T$
  ▷ E.val := T.val

4: $T_1 \longrightarrow T_2 * F$
  ▷ T_1.val := product (T_2.val, F.val)

5: $T_1 \longrightarrow T_2 / F$
  ▷ T_1.val := quotient (T_2.val, F.val)

6: $T \longrightarrow F$
  ▷ T.val := F.val

7: $F_1 \longrightarrow - F_2$
  ▷ F_1.val := additive_inverse (F_2.val)

8: $F \longrightarrow (E)$
  ▷ F.val := E.val

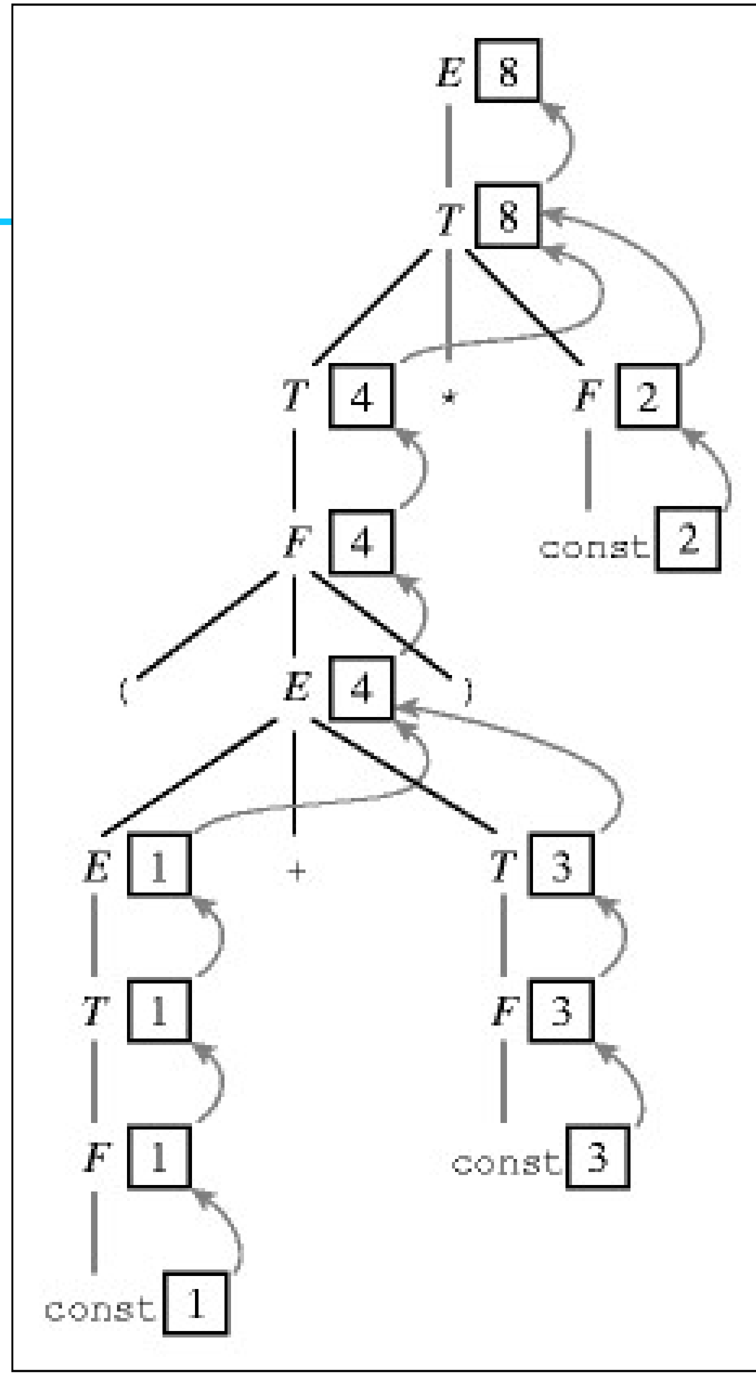
9: $F \longrightarrow$ const
  ▷ F.val := const.val

Attribute Flow

- Context-free grammars are not tied to an specific parsing order
  - *E.g.* Recursive descent, LR parsing
- Attribute grammars are not tied to an specific evaluation order
  - This evaluation is known as the *annotation* or *decoration* of the parse tree

Attribute Flow
Example

- The figure shows the result of annotating the parse tree for `(1+3)*2`
- Each symbols has at most one attribute shown in the corresponding box
  - Numerical value in this example
  - Operator symbols have no value
- Arrows represent *attribute flow*



Attribute Flow

Example

1: $E_1 \longrightarrow E_2$ + T
  ▷ E_1.val := sum (E_2.val, T.val)

2: $E_1 \longrightarrow E_2$ - T
  ▷ E_1.val := difference (E_2.val, T.val)

3: $E \longrightarrow T$
  ▷ E.val := T.val

4: $T_1 \longrightarrow T_2 * F$
  ▷ T_1.val := product (T_2.val, F.val)

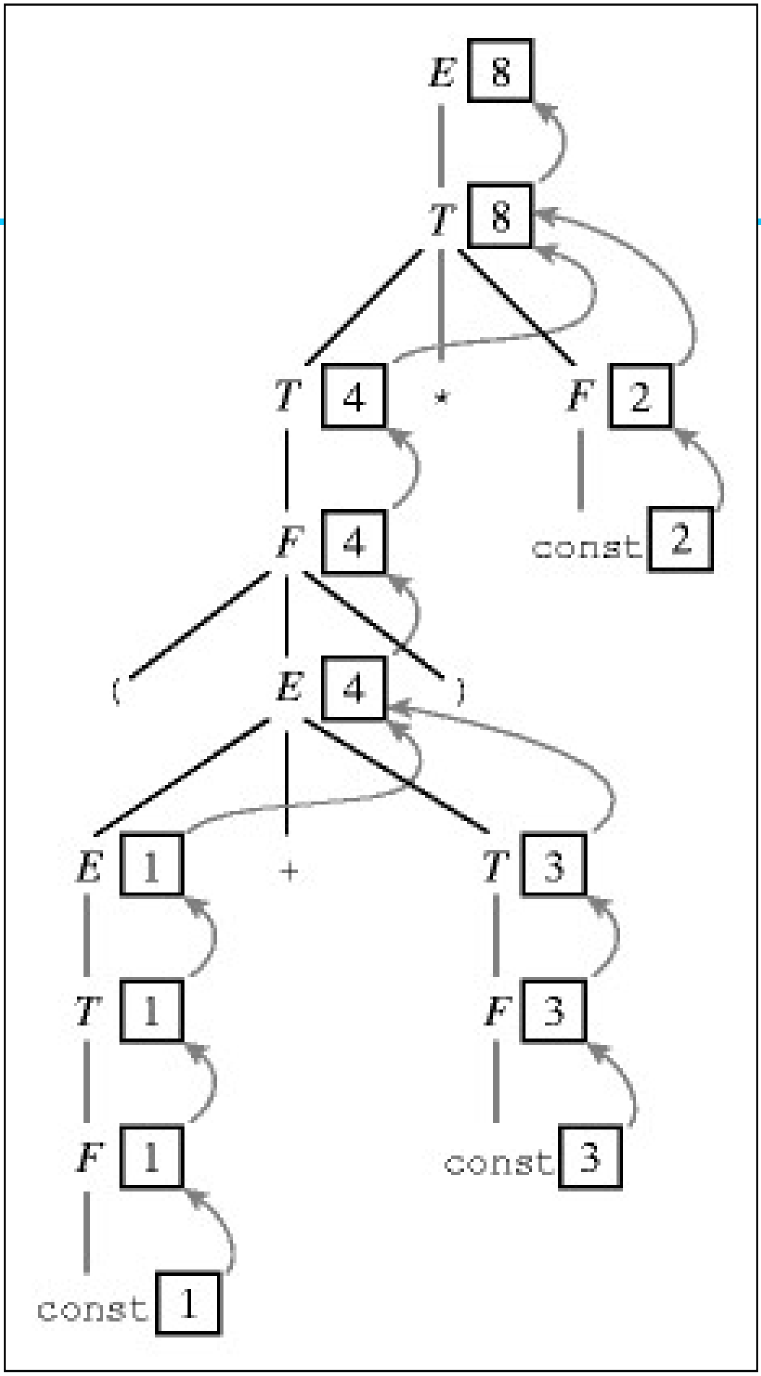
5: $T_1 \longrightarrow T_2$ / F
  ▷ T_1.val := quotient (T_2.val, F.val)

6: $T \longrightarrow F$
  ▷ T.val := F.val

7: $F_1 \longrightarrow$ - F_2
  ▷ F_1.val := additive_inverse (F_2.val)

8: $F \longrightarrow$ (E)
  ▷ F.val := E.val

9: $F \longrightarrow$ const
  ▷ F.val := const.val



Attribute Flow
Synthetic and Inherited Attributes

- In the previous example, semantic information is pass up the parse tree
  - We call this type of attributes are called *synthetic attributes*
  - Attribute grammar with synthetic attributes only are said to be *S-attributed*
- Semantic information can also be passed down the parse tree
  - Using *inherited attributes*
  - Attribute grammar with inherited attributes only are said to be *non-S-attributed*

Attribute Flow
Inherited Attributes

- *L-attributed* grammars, such as the one on the next slide, can still be evaluated in a single left-to-right pass over the input.
- Each synthetic attribute of a LHS symbol (by definition of *synthetic*)depends only on attributes of its RHS symbols.
- Each inherited attribute of a RHS symbol (by definition of *L-attributed)* depends only on inherited attributes of the LHS symbol or on synthetic or inherited attributes of symbols to its left in the RHS.
- Top-down grammars generally require non-S-attributed flows
  - The previous annotated grammar was an S-attributed LR(1)
  - L-attributed grammars are the most general class of attribute grammars that can be evaluated during an LL parse.

LL Grammar

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: $TT_1 \longrightarrow -\ T\ TT_1$

$\triangleright$ 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 × F.val $\qquad$ $\triangleright$ FT_1.val := FT_2.val

7: $FT_1 \longrightarrow /\ F\ FT_2$

$\triangleright$ FT_2.st := FT_1.st ÷ 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: $F \longrightarrow (\ E\ )$

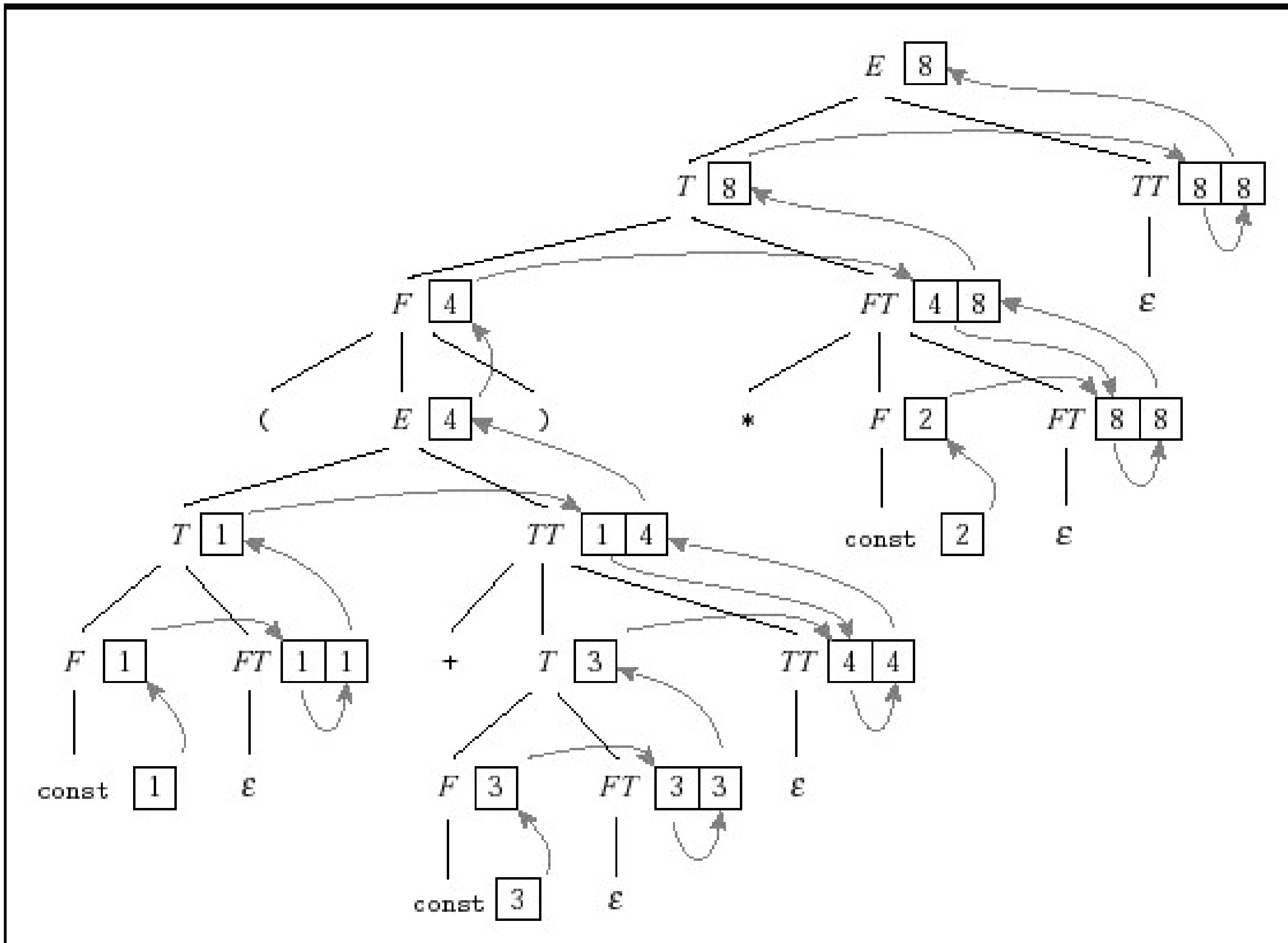
$\triangleright$ F.val := E.val

11: $F \longrightarrow$ const

$\triangleright$ F.val := const.val

Non-S-Attributed Grammars

Example



1: $E \longrightarrow T\ TT$
- ▷ $TT.st := T.val$ ▷ $E.val := TT.val$

2: $TT_1 \longrightarrow +\ T\ TT_2$
- ▷ $TT_2.st := TT_1.st + T.val$ ▷ $TT_1.val := TT_2.val$

3: $TT_1 \longrightarrow -\ T\ TT_1$
- ▷ $TT_2.st := TT_1.st - T.val$ ▷ $TT_1.val := TT_2.val$

4: $TT \longrightarrow \epsilon$
- ▷ $TT.val := TT.st$

5: $T \longrightarrow F\ FT$
- ▷ $FT.st := F.val$ ▷ $T.val := FT.val$

6: $FT_1 \longrightarrow *\ F\ FT_2$
- ▷ $FT_2.st := FT_1.st \times F.val$ ▷ $FT_1.val := FT_2.val$

7: $FT_1 \longrightarrow /\ F\ FT_2$
- ▷ $FT_2.st := FT_1.st \div F.val$ ▷ $FT_1.val := FT_2.val$

8: $FT \longrightarrow \epsilon$
- ▷ $FT.val := FT.st$

9: $F_1 \longrightarrow -\ F_2$
- ▷ $F_1.val := -\ F_2.val$

10: $F \longrightarrow (\ E\ )$
- ▷ $F.val := E.val$

11: $F \longrightarrow$ const
- ▷ $F.val := const.val$

Syntax Tree

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved.
- A *one-pass* compiler interleaves scanning, parsing, semantic analysis, and code generation in a single traversal of the input.
- A common approach interleaves construction of a syntax tree with parsing (eliminating the need to build an explicit parse tree), then follows with separate, sequential phases for semantic analysis and code generation.

Bottom-up Attribute Grammar to Construct a Syntax Tree

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

$\quad \triangleright$ $\mathsf{E_1.ptr := make\_bin\_op\ ("+",\ E_2.ptr,\ T.ptr)}$

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

$\quad \triangleright$ $\mathsf{E_1.ptr := make\_bin\_op\ ("-",\ E_2.ptr,\ T.ptr)}$

$E \longrightarrow T$

$\quad \triangleright$ $\mathsf{E.ptr := T.ptr}$

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

$\quad \triangleright$ $\mathsf{T_1.ptr := make\_bin\_op\ ("\times",\ T_2.ptr,\ F.ptr)}$

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

$\quad \triangleright$ $\mathsf{T_1.ptr := make\_bin\_op\ ("\div",\ T_2.ptr,\ F.ptr)}$

$T \longrightarrow F$

$\quad \triangleright$ $\mathsf{T.ptr := F.ptr}$

$F_1 \longrightarrow \texttt{ - } F_2$

$\quad \triangleright$ $\mathsf{F_1.ptr := make\_un\_op\ ("{}^{+}\!/\_",\ F_2.ptr)}$

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

$\quad \triangleright$ $\mathsf{F.ptr := E.ptr}$

$F \longrightarrow \texttt{const}$

$\quad \triangleright$ $\mathsf{F.ptr := make\_leaf\ (const.val)}$

Construction of the Syntax Tree

$E_1 \longrightarrow E_2 + T$

▷ $E_1.ptr := make\_bin\_op\ (\text{"+"}, E_2.ptr, T.ptr)$

$E_1 \longrightarrow E_2 - T$

▷ $E_1.ptr := make\_bin\_op\ (\text{"−"}, E_2.ptr, T.ptr)$

$E \longrightarrow T$

▷ $E.ptr := T.ptr$

$T_1 \longrightarrow T_2 * F$

▷ $T_1.ptr := make\_bin\_op\ (\text{"×"}, T_2.ptr, F.ptr)$

$T_1 \longrightarrow T_2\ /\ F$

▷ $T_1.ptr := make\_bin\_op\ (\text{"÷"}, T_2.ptr, F.ptr)$

$T \longrightarrow F$

▷ $T.ptr := F.ptr$

$F_1 \longrightarrow -\ F_2$

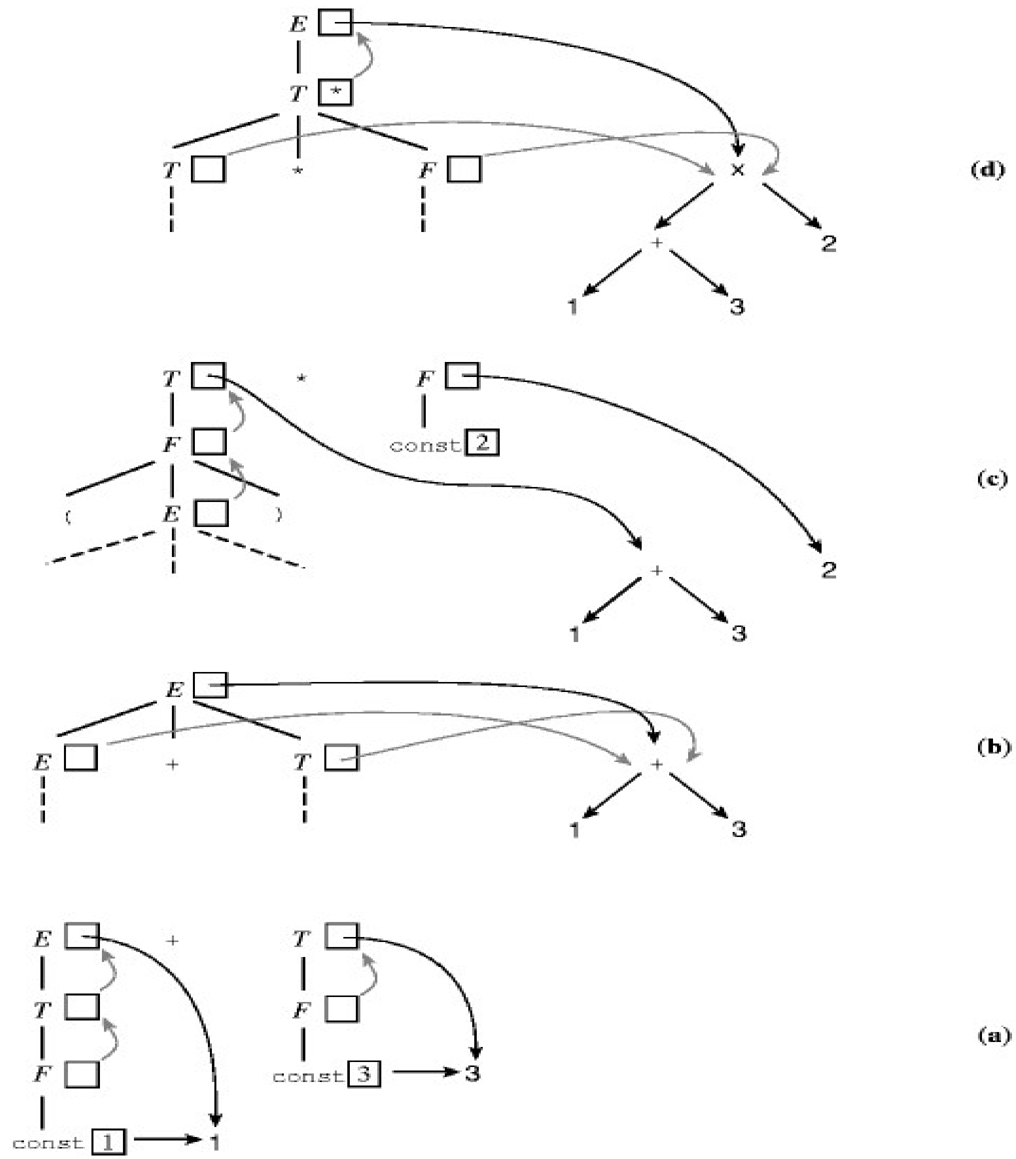
▷ $F_1.ptr := make\_un\_op\ (\text{"+/\_"}, F_2.ptr)$

$F \longrightarrow (\ E\ )$

▷ $F.ptr := E.ptr$

$F \longrightarrow \text{const}$

▷ $F.ptr := make\_leaf\ (const.val)$



Action Routines

- Automatic tools can construct a parser for a given context-free grammar
  - *E.g.* yacc
- Automatic tools can construct a semantic analyzer for an attribute grammar
  - An ad hoc techniques is to annotate the grammar with executable rules
  - These rules are known as *action routines*

Action Rules for the Previous LL(1) attribute grammar

E => T { TT.st := T.v } TT { E.v := TT.v }

TT => + T { TT2.st := TT1.st + T.v } TT { TT1.v := TT2.v }

TT => - T { TT2.st := TT1.st - T.v } TT { TT1.v := TT2.v }

TT => { TT.v := TT.st }

T => F { FT.st := F.v } FT { T.v := FT.v }

FT => * F { FT2.st := FT1.st * F.v } FT { FT1.v := FT2.v }

FT => / F { FT2.st := FT1.st / F.v } FT { FT1.v := FT2.v }

FT => { FT.v := FT.st }

F => - F { F1.v := - F2.v }

F => (E) { F.v := E.v }

F => const { F.v := C.v }

Action Rules

- The ease with which rules were incorporated in the grammar is due to the fact that the attribute grammar is *L-attributed*.
- The action rules for *L-attributed* grammars, in which the attribute flow is depth-first left-to-right, can be evaluated in the order of the parse tree prediction for LL grammars.
- Action rules for *S-attributed* grammars can be incorporated at the end of the right-hand sides of LR grammars. But, if action rules are responsible for a significant part of the semantic analysis, they will need more contextual information to do their job.

Static and Dynamic Semantics

- Attribute grammars add basic semantic rules to the specification of a language
  - They specify *static semantics*
- But they are limited to the semantic form that can be checked at compile time
- Other semantic properties cannot be checked at compile time
  - They are described using *dynamic semantics*

Dynamic Semantics

- Use to formally specify the behavior of a programming language
  - Semantic-based error detection
  - Correctness proofs
- There is not a universally accepted notation
  - **Operational semantics**
    - Executing statements that represent changes in the state of a real or simulated machine
  - **Axiomatic semantics**
    - Using predicate calculus (pre and post-conditions)
  - **Denotational semantics**
    - Using recursive function theory

Semantic Specification

- The most common way of *specifying* the semantics of a language is plain english
- There is a lack of formal rigor in the semantic specification of programming languages