Semantic Analysis

Chapter 4

Role of Semantic Analysis

- Syntax
  - "form" of a program
  - "easy": check membership for CFG
  - linear time
- Semantics
  - meaning of a program
  - *impossible*: program correctness **undecidable**!
  - we do whatever we can

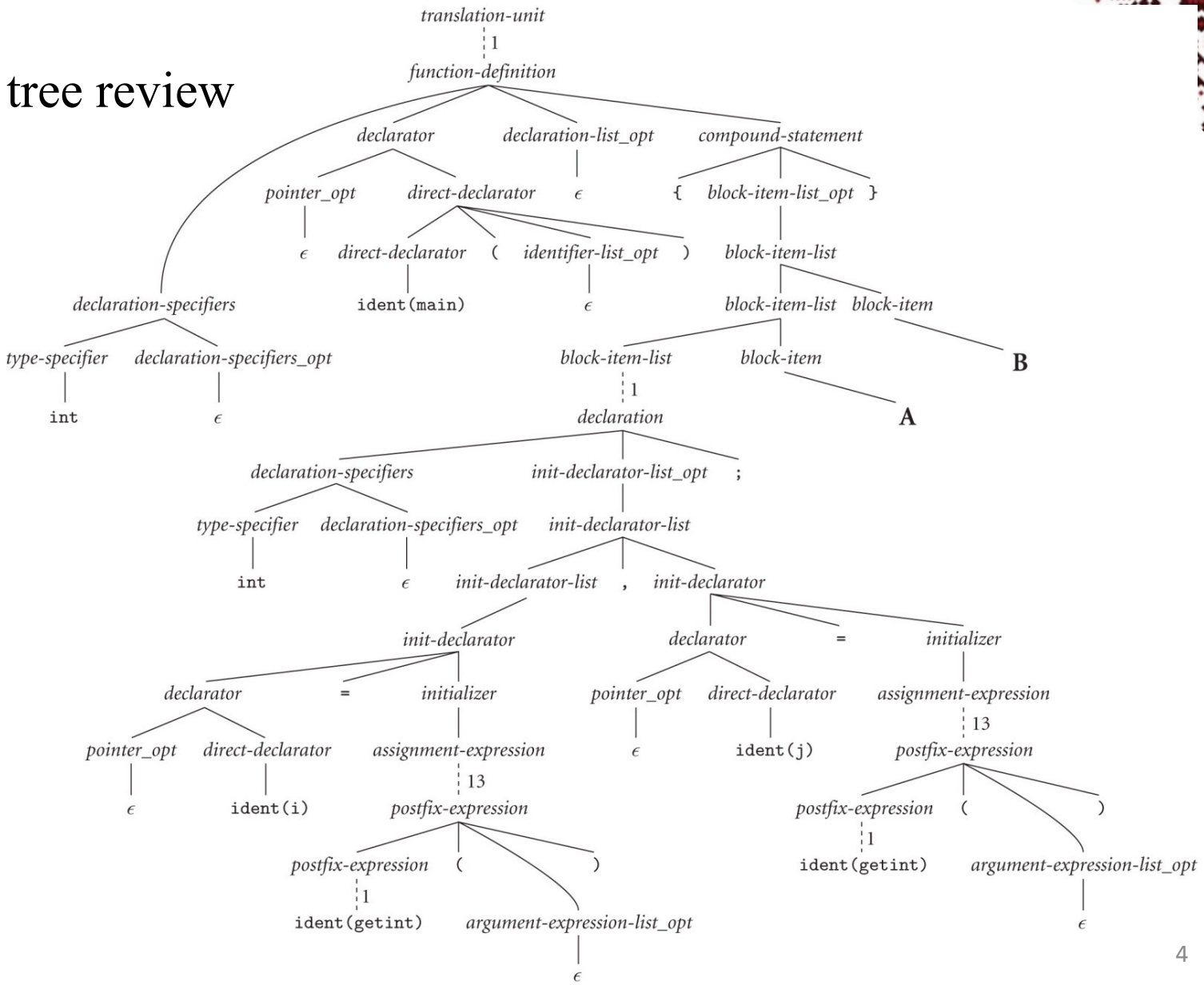
S-attributed grammar — bottom up (LR)
L-attributed grammar — top down (LL)

Role of Semantic Analysis

- *Static* semantics – compile time
  - enforces static semantic rules at compile time
  - generates code to enforce dynamic semantic rules
  - constructs a syntax tree
  - information gathered for the code generator
- *Dynamic* semantics – run time
  - division by zero
  - index out of bounds
- Semantic analysis (and intermediate code generation) – described in terms of annotation (decoration) of parse tree or syntax tree
  - annotations are attributes – *attribute grammars*

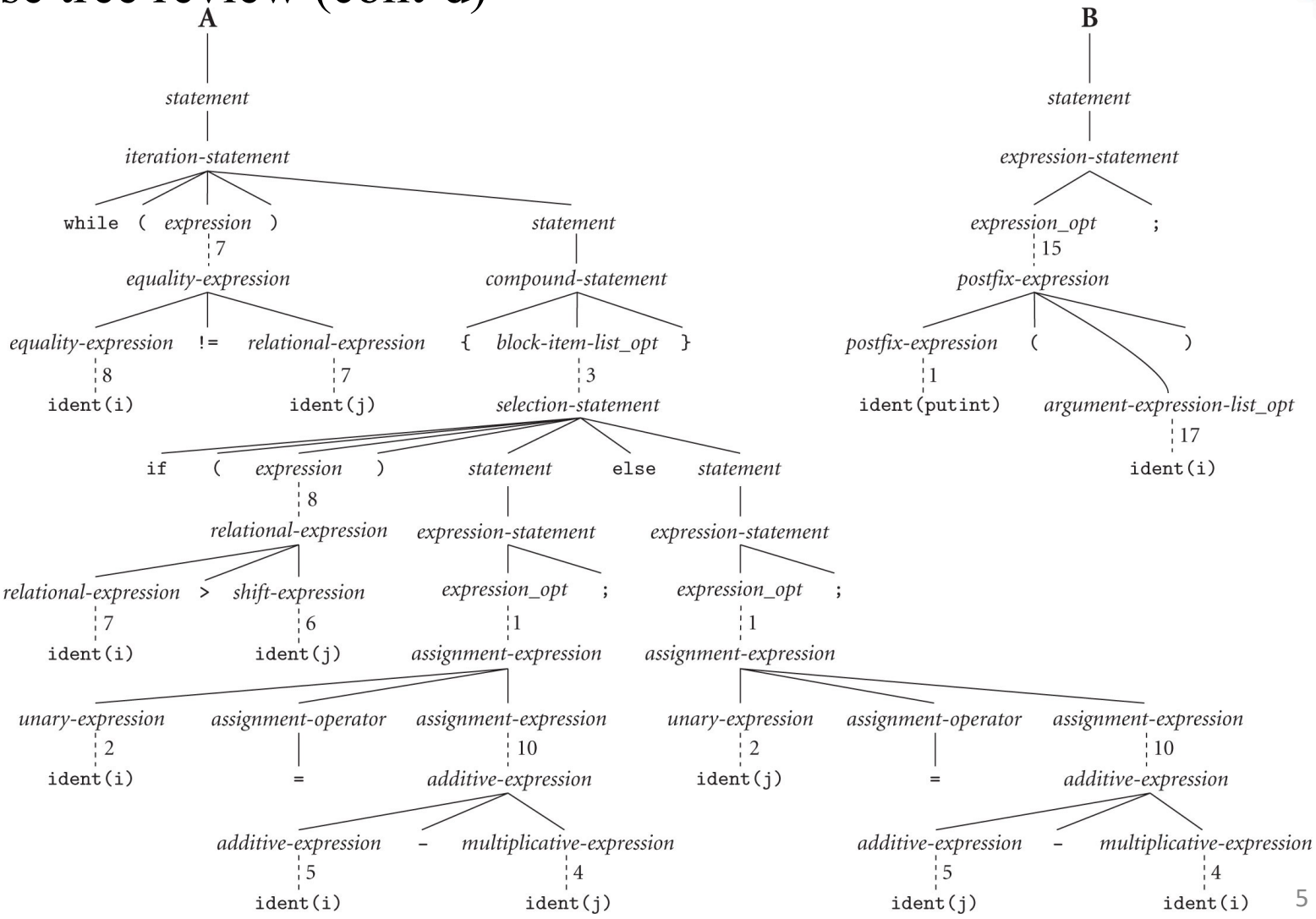
Role of Semantic Analysis

- Parse tree review



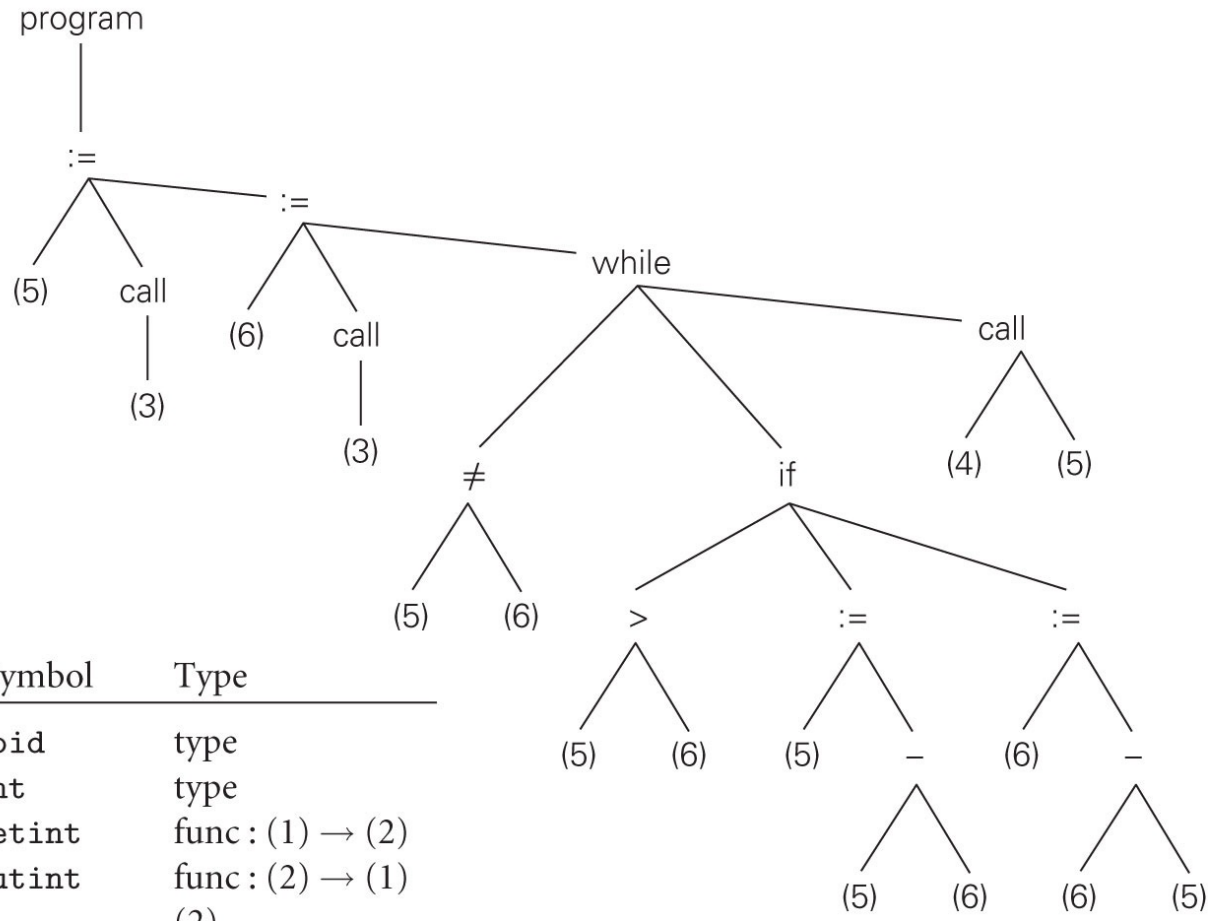
Role of Semantic Analysis

- Parse tree review (cont'd)



Role of Semantic Analysis

- Parse tree can be replaced by the smaller *syntax tree* (review)



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Role of Semantic Analysis

- Dynamic checks
- compiler generates code for dynamic checking
- can be disabled for increased speed
  - Tony Hoare: “The programmer who disables semantic checks is like a sailing enthusiast who wears a life jacket when training on dry land but removes it when going to sea.”
- C – almost no checks
- Java – as many checks as possible
- trend is towards stricter rules
- Example: `3 + “four”`
  - Perl – attempts to infer meaning
  - Python – run-time error

Role of Semantic Analysis

- *Logical Assertions*
- Java:

```
assert denominator != 0;
```

`AssertionError` – exception thrown if semantic check fails

- C:

```
assert(denominator != 0);
myprog.c:42: failed assertion ‘denominator != 0’
```

- Python:

```
assert denominator != 0, “Zero denominator!”
AssertionError: Zero denominator!
```

- Invariants, preconditions, postconditions
  - Euclid, Eiffel, Ada
  - invariant: expected to be true at all check points
  - pre/postconditions: true at beginning/end of subroutines

Role of Semantic Analysis

- Static analysis
  - compile-time algorithms that predict run-time behavior
  - extensive static analysis eliminates the need for some dynamic checks
    - precise type checking
    - enforced initialization of variables

Attribute Grammars

- *Attribute grammar*:
  - formal framework for decorating the parse or syntax tree
  - for semantic analysis
  - for (intermediate) code generation
- Implementation
  - automatic
    - tools that construct semantic analyzers (attribute evaluator)
  - ad hoc
    - action routines

Attribute Grammars

- Example: LR (bottom-up) grammar
  - arithmetic expr. with constants, precedence, associativity
  - the grammar alone says nothing about the meaning
  - *attributes*: connection with mathematical concept

1. $E_1 \rightarrow E_2$ `+` T
2. $E_1 \rightarrow E_2$ `-` T
3. $E \rightarrow T$
4. $T_1 \rightarrow T_2$ `*` F
5. $T_1 \rightarrow T_2$ `/` F
6. $T \rightarrow F$
7. $F_1 \rightarrow$ `-` F_2
8. $F \rightarrow$ `(` E `)`
9. $F \rightarrow$ `const`

Attribute Grammars

- Attribute grammar
- `S.val`: the arithmetic value of the string derived from `S`
- `const.val`: provided by the scanner
- copy rules: 3, 6, 8, 9
- semantic functions: `sum`, `diff`, `prod`, `quot`, `add_inv`
  - use only attributes of the current production

这些是怎么翻译它

1.	$E_1 \rightarrow E_2 + T$	▷ E_1.val := sum(E_2.val, T.val)
2.	$E_1 \rightarrow E_2 - T$	▷ E_1.val := diff(E_2.val, T.val)
3.	$E \rightarrow T$	▷ E.val := T.val
4.	$T_1 \rightarrow T_2 * F$	▷ T_1.val := prod(T_2.val, F.val)
5.	$T_1 \rightarrow T_2 / F$	▷ T_1.val := quot(T_2.val, F.val)
6.	$T \rightarrow F$	▷ T_1.val := F.val
7.	$F_1 \rightarrow -F_2$	▷ F_1.val := add_inv(F_2.val)
8.	$F \rightarrow (E)$	▷ F.val := E.val
9.	$F \rightarrow$ **const**	▷ F.val := const.val

Attribute Grammars

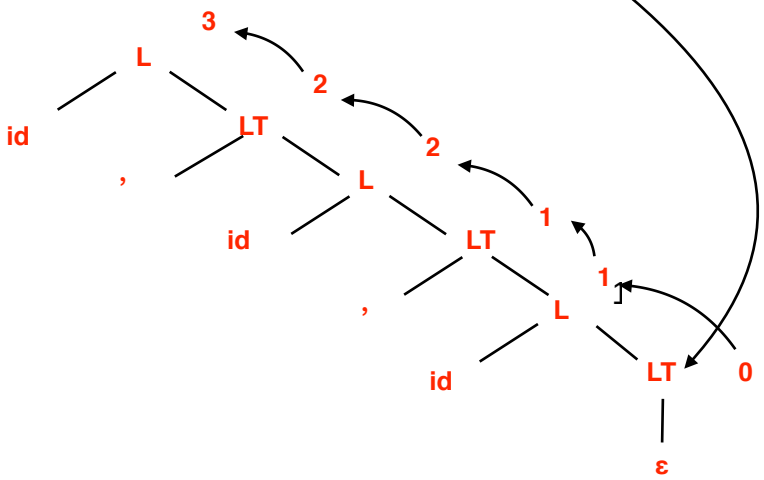
从下往上算，每个expression的总数值存在root上

- Example: LL (top-down) grammar
  - count the elements of a list
  - "in-line" notation of semantic functions

$L \rightarrow$ `id` LT ▷ `L.c := 1 + LT.c`

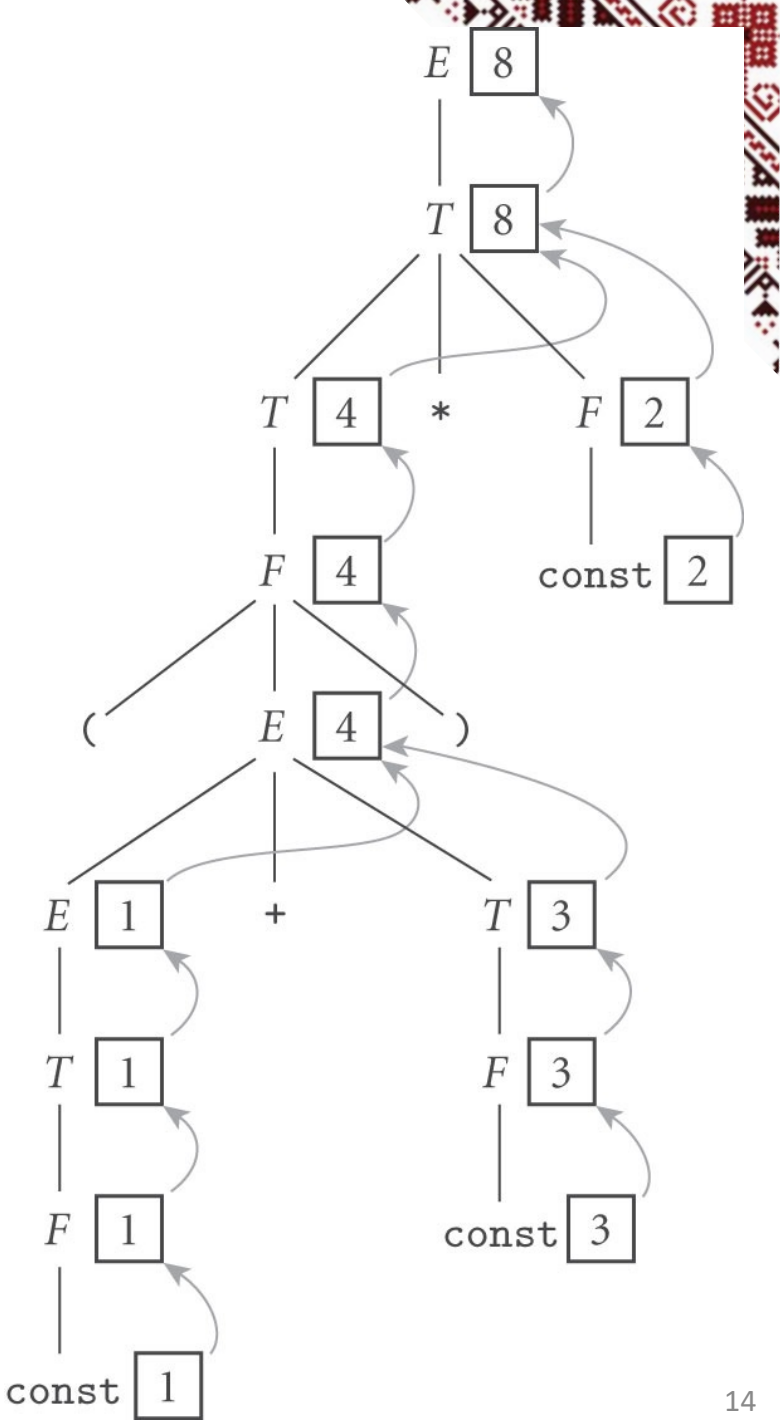
$LT \rightarrow$ `,` L ▷ `LT.c := L.c`

$LT \rightarrow \varepsilon$ ▷ `LT.c := 0`



Evaluating Attributes

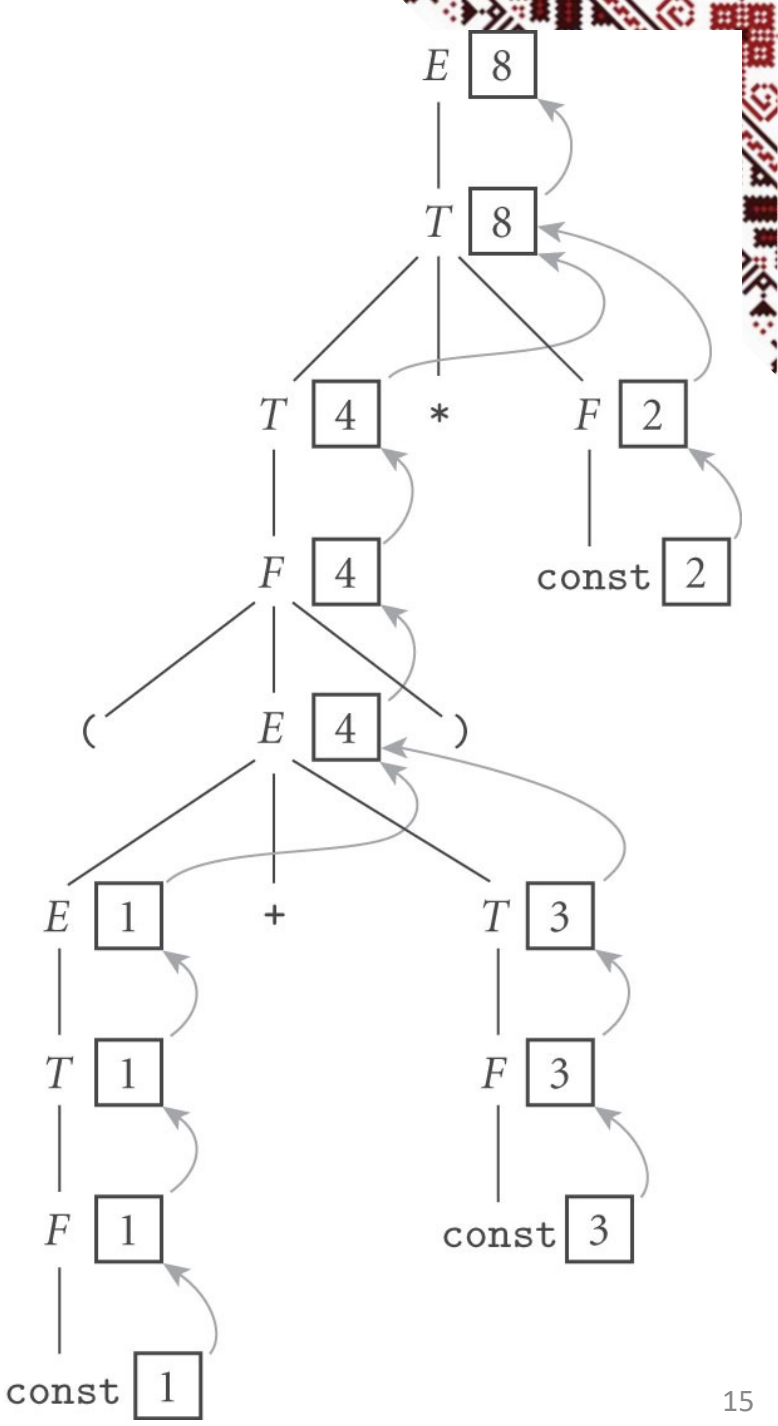
- *Annotation* of parse tree:
  - evaluation of attributes
  - also called *decoration*
- Example:
  - LR(1) grammar (arithm. exp.)
  - string: `(1+3)*2`
  - *val* attribute of root will hold the value of the expression



Evaluating Attributes

- Types of attributes:
  - synthesized
  - inherited
- *Synthesized* attributes:
  - values calculated only in productions where they appear only on the left-hand side
  - *attribute flow*: bottom-up only
- *S-attributed* grammar: all attributes are synthesized

LR(1) Algorithm



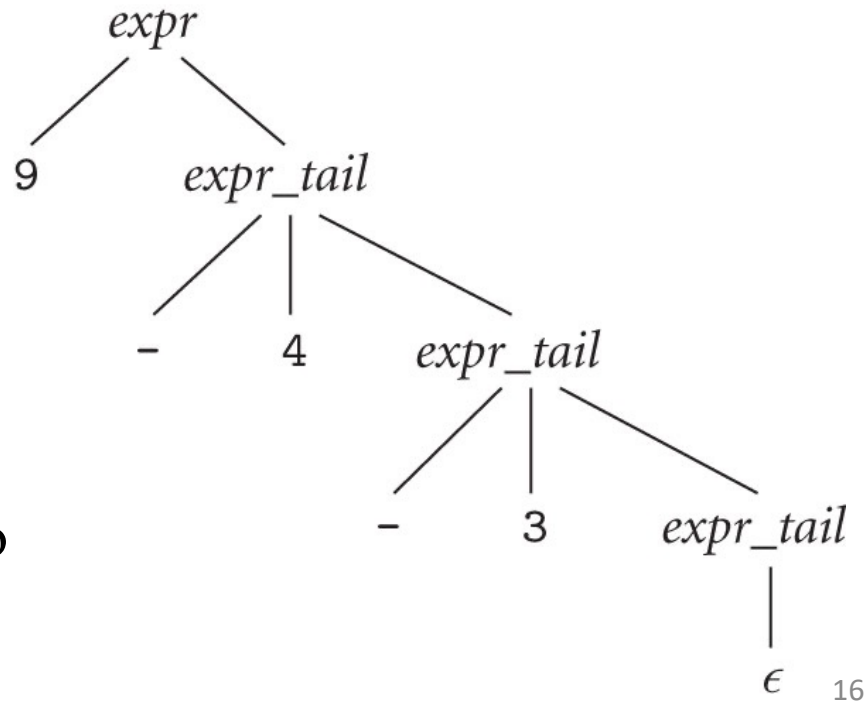
Evaluating Attributes

- *Inherited* attributes:
  - values calculated when their symbol is on RHS of the production
  - Example: LL(1) grammar for subtraction

expr → `const` *expr_tail*

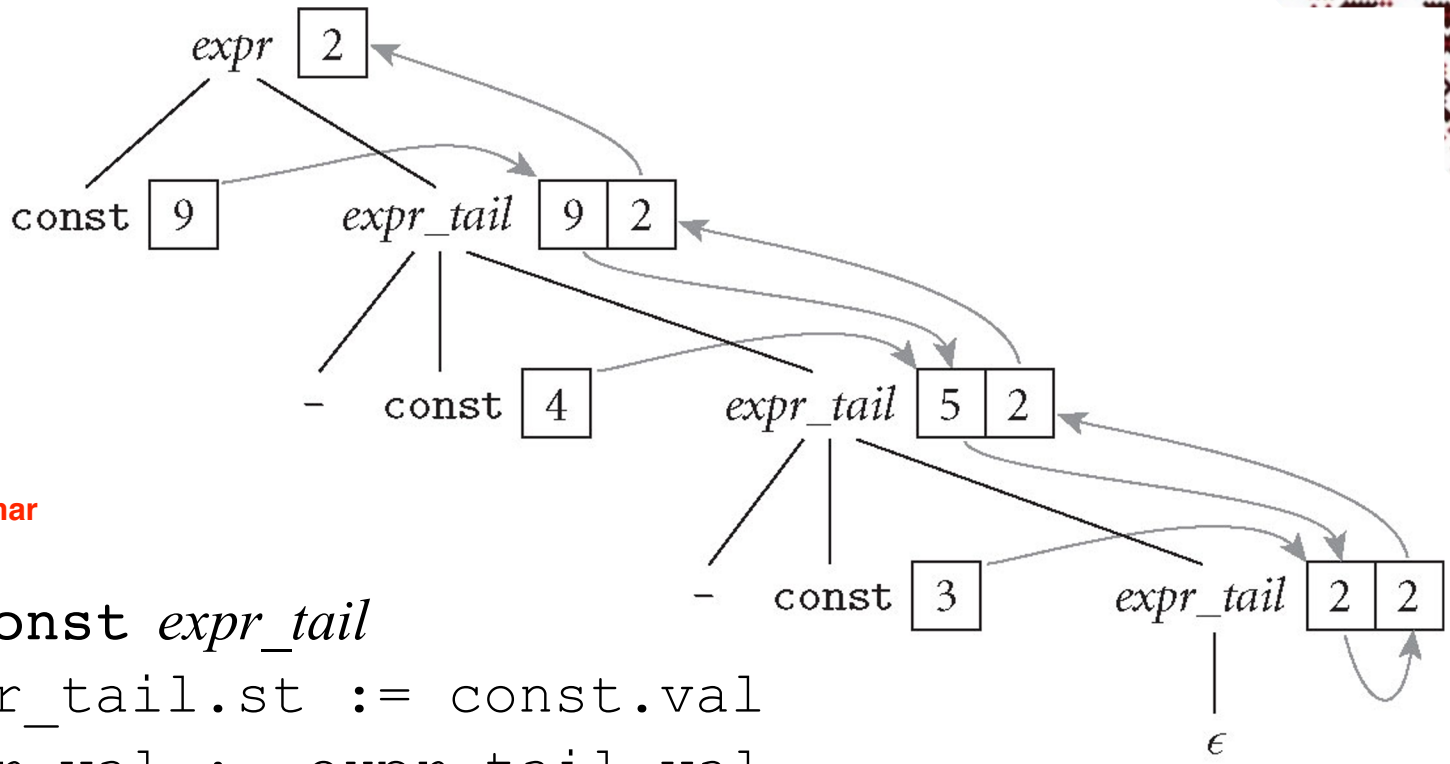
expr_tail → `-` `const` *expr_tail*

→ ε



- string: `9 - 4 – 3`
- `'-'` left-associative means cannot have only bottom-up
- need to pass `9` to *expr_tail* to combine with `4`

Evaluating Attributes



Attribute grammar

expr → **const** *expr_tail*
- ▷ expr_tail.st := const.val
- ▷ expr.val := expr_tail.val

$expr\_tail_1$ → **-** **const** $expr\_tail_2$
- ▷ $expr\_tail_2$.st := $expr\_tail_1$.st − const.val
- ▷ $expr\_tail_1$.val := $expr\_tail_2$.val

expr_tail → ε
- ▷ expr_tail.val := expr_tail.st

Evaluating Attributes

- Example: Complete LL(1) grammar for arithmetic expressions
- Complicated because:
  - Operators are left-associative but grammar cannot be left-recursive
  - Left and right operands of an operator are in separate productions

st —> subtotal

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. $TT_1 \longrightarrow -\ T\ TT_2$
   - ▷ 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

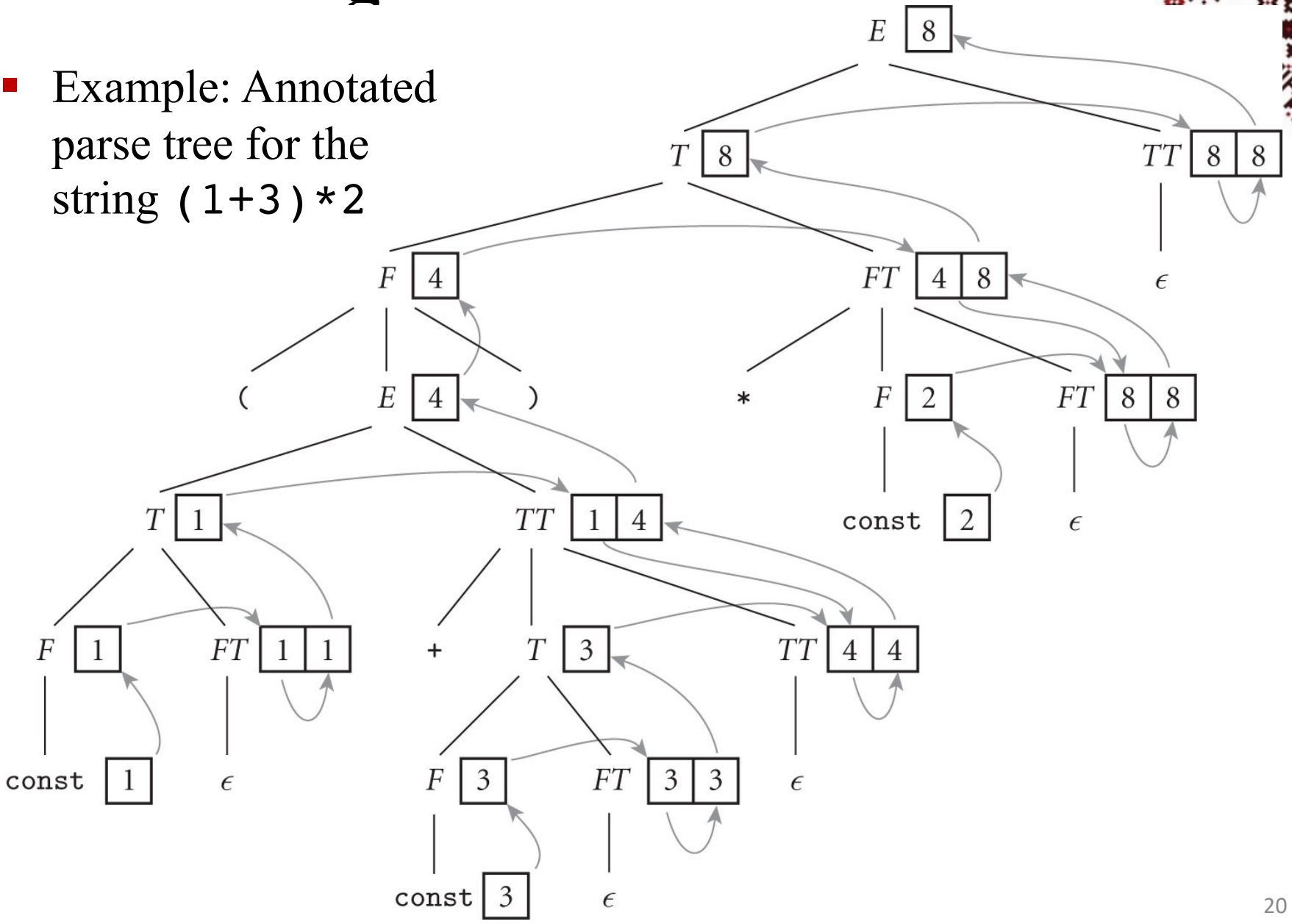
Evaluating Attributes

- Example: LL(1) grammar for arithmetic expressions (cont'd)

6. $FT_1 \longrightarrow * \; F \; FT_2$
   ▷ FT_2.st := FT_1.st × F.val ▷ FT_1.val := FT_2.val
7. $FT_1 \longrightarrow / \; F \; FT_2$
   ▷ FT_2.st := FT_1.st ÷ 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

Evaluating Attributes

- Example: Annotated parse tree for the string `(1+3)*2`



Evaluating Attributes

- *Attribute flow*
- *Declarative* notation:
  - no evaluation order specified for attributes
- *Well-defined* grammar:
  - its rules determine unique values for attributes in any parse tree
- *Non-circular* grammar:
  - no attribute depends on itself in any parse tree
- *Translation scheme*:
  - algorithm that decorates parse tree in agreement with the attribute flow

Evaluating Attributes

- Translation scheme:
  - Obvious scheme: repeated passes until no further changes
    - halts only if well defined
  - Dynamic scheme: better performance
    - topologically sort the attribute flow graph
- *Static* scheme: fastest, $O(n)$
  - based on the structure of the grammar
- S-attributed grammar – simplest static scheme
  - flow is strictly bottom-up; attributes can be evaluated in the same order the nodes are generated by an LR-parser

Evaluating Attributes

- Attribute `A.s` is said to *depend* on attribute `B.t` if `B.t` is ever passed to a semantic function that returns a value for `A.s`

- *L-attributed grammar*:
  - each synthesized attribute of a LHS symbol depends only on that symbol's own inherited attributes or on attributes (synthesized or inherited) of the RHS symbols
  - each inherited attribute of a RHS symbol depends only on inherited attributes of the LHS symbol or on attributes (synthesized or inherited) of symbols to its left in the RHS

- L-attributed grammar
  - attributes can be evaluated by a single left-to-right depth-first traversal

Evaluating Attributes

- S-attributed implies L-attributed (but not vice versa)
- S-attributed grammar: the most general class of attribute grammars for which evaluation can be implemented on the fly during an LR parse
- L-attributed grammar: the most general class of attribute grammars for which evaluation can be implemented on the fly during an LL parse
- If semantic analysis interleaved with parsing:
  - bottom-up parser paired with S-attribute translation scheme
  - top-down parser paired with L-attributed translation scheme

Syntax Tree

- *One-pass compiler*
  - interleaved: parsing, semantic analysis, code generation
  - saves space (older computers)
    - no need to build parse tree or syntax tree
- *Multi-pass compiler*
  - possible due to increases in speed and memory
  - more flexible
  - better code improvement
    - Example: forward references
    - declaration before use no longer necessary

Syntax Tree

- *Syntax Tree*
  - separate parsing and semantics analysis
  - attribute rules for CFG are used to build the syntax tree
  - semantics easier on syntax tree
    - syntax tree reflects semantic structure better
    - can pass the tree in different order than that of parser

Syntax Trees Construction

- Bottom-up (S-attributed) attribute grammar to construct syntax tree

$E_1 \longrightarrow E_2 + T$

▷ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)

$E_1 \longrightarrow E_2 - T$

▷ E_1.ptr := make_bin_op("−", E_2.ptr, T.ptr)

$E \longrightarrow T$

▷ E.ptr := T.ptr

$T_1 \longrightarrow T_2 * F$

▷ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)

$T_1 \longrightarrow T_2 / F$

▷ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)

Syntax Trees Construction

- Bottom-up (S-attributed) attribute grammar to construct syntax tree (cont'd)

$T \longrightarrow F$
▷ T.ptr := F.ptr

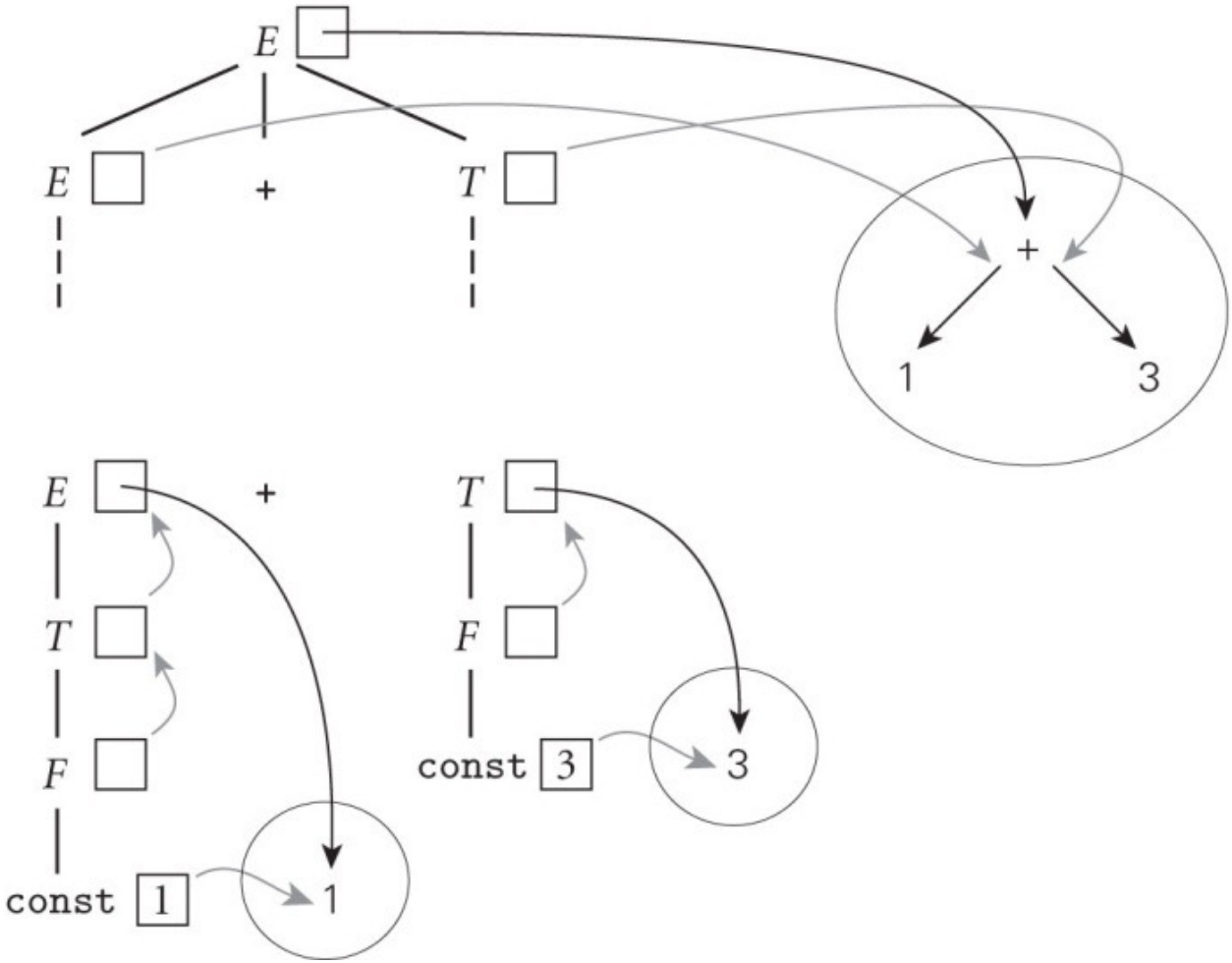
$F_1 \longrightarrow$ - F_2
▷ F_1.ptr := make_un_op("+/_", F_2.ptr)

$F \longrightarrow$ (E)
▷ F.ptr := E.ptr

$F \longrightarrow$ `const`
▷ F.ptr := make_leaf(const.val)

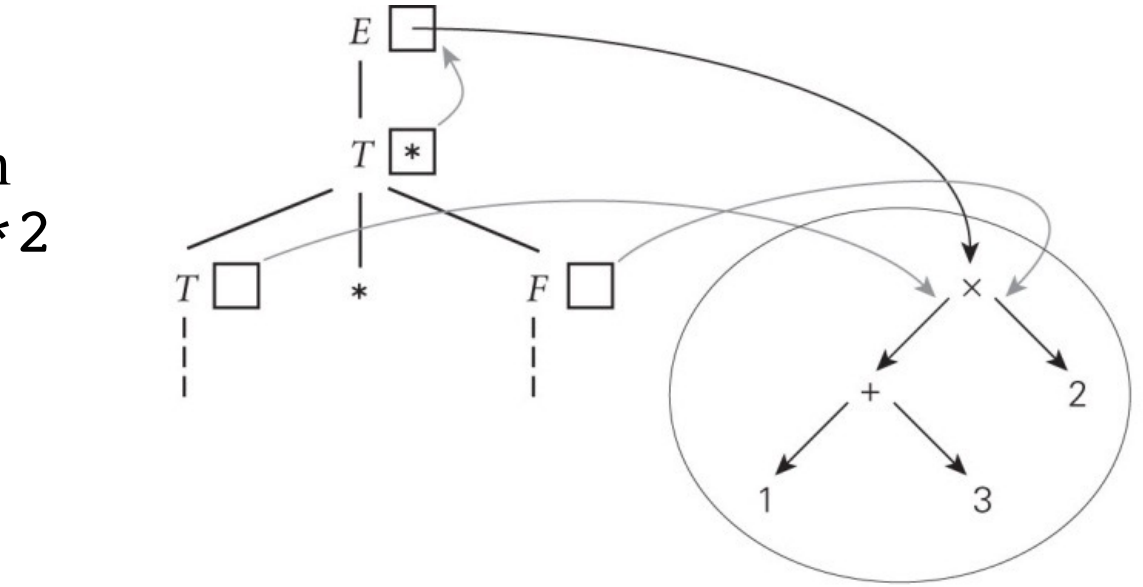
Syntax Trees Construction

- Syntax tree construction for `(1+3)*2`

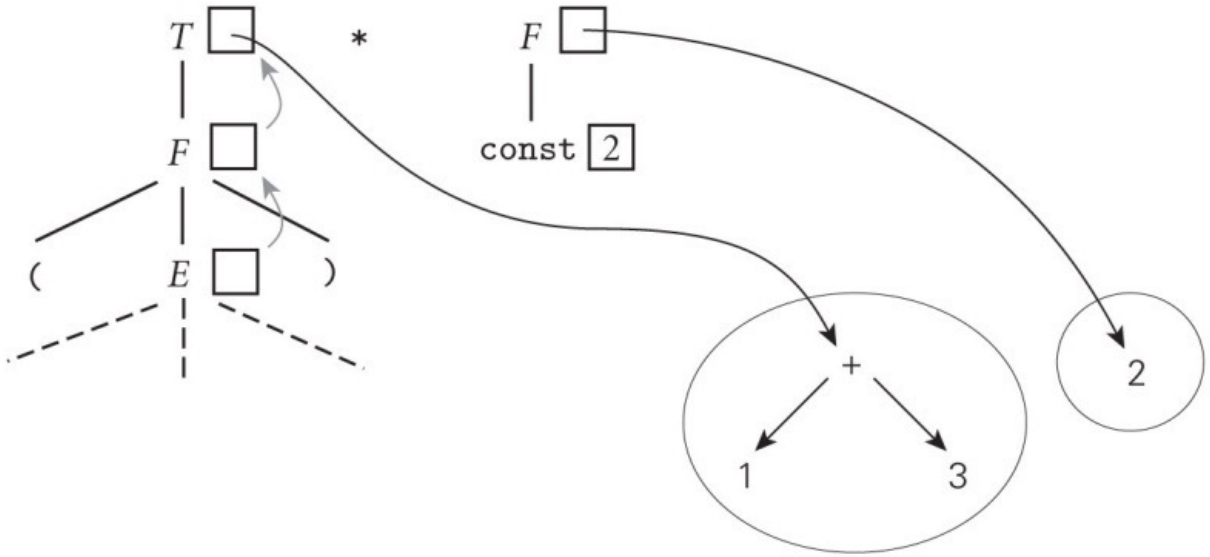


Syntax Trees Construction

- Syntax tree construction for `(1+3)*2` (cont'd)



(d)



(c)

Syntax Trees Construction

- Top-down (L-attributed) attribute grammar to construct syntax tree

$E \longrightarrow T\ TT$
- ▷ 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

Syntax Trees Construction

- Top-down (L-attributed) attribute grammar to construct syntax tree (cont'd)

$FT_1 \longrightarrow$ * 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)

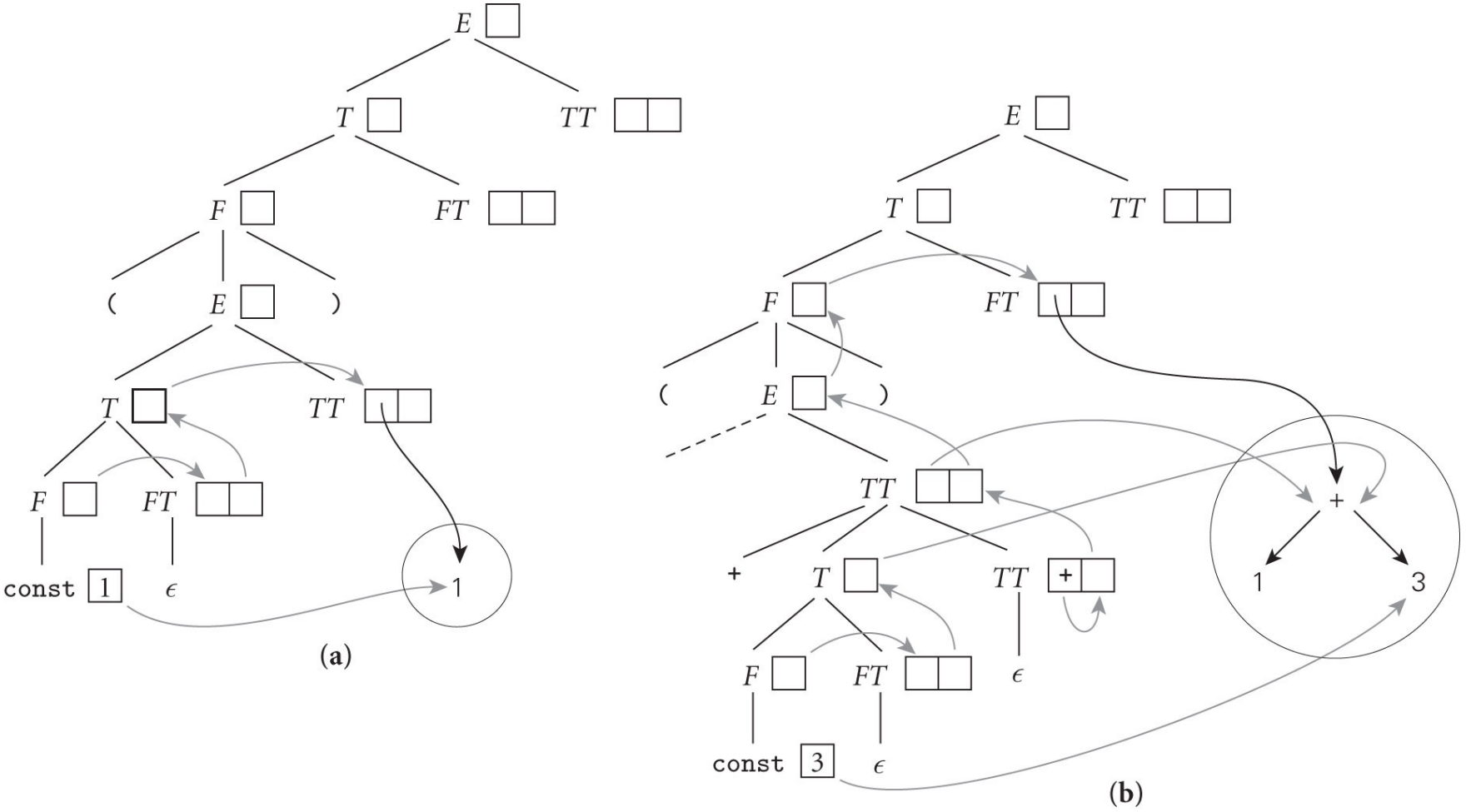
▷ F.ptr := E.ptr

$F \longrightarrow$ `const`

▷ F.ptr := make_leaf(const.val)

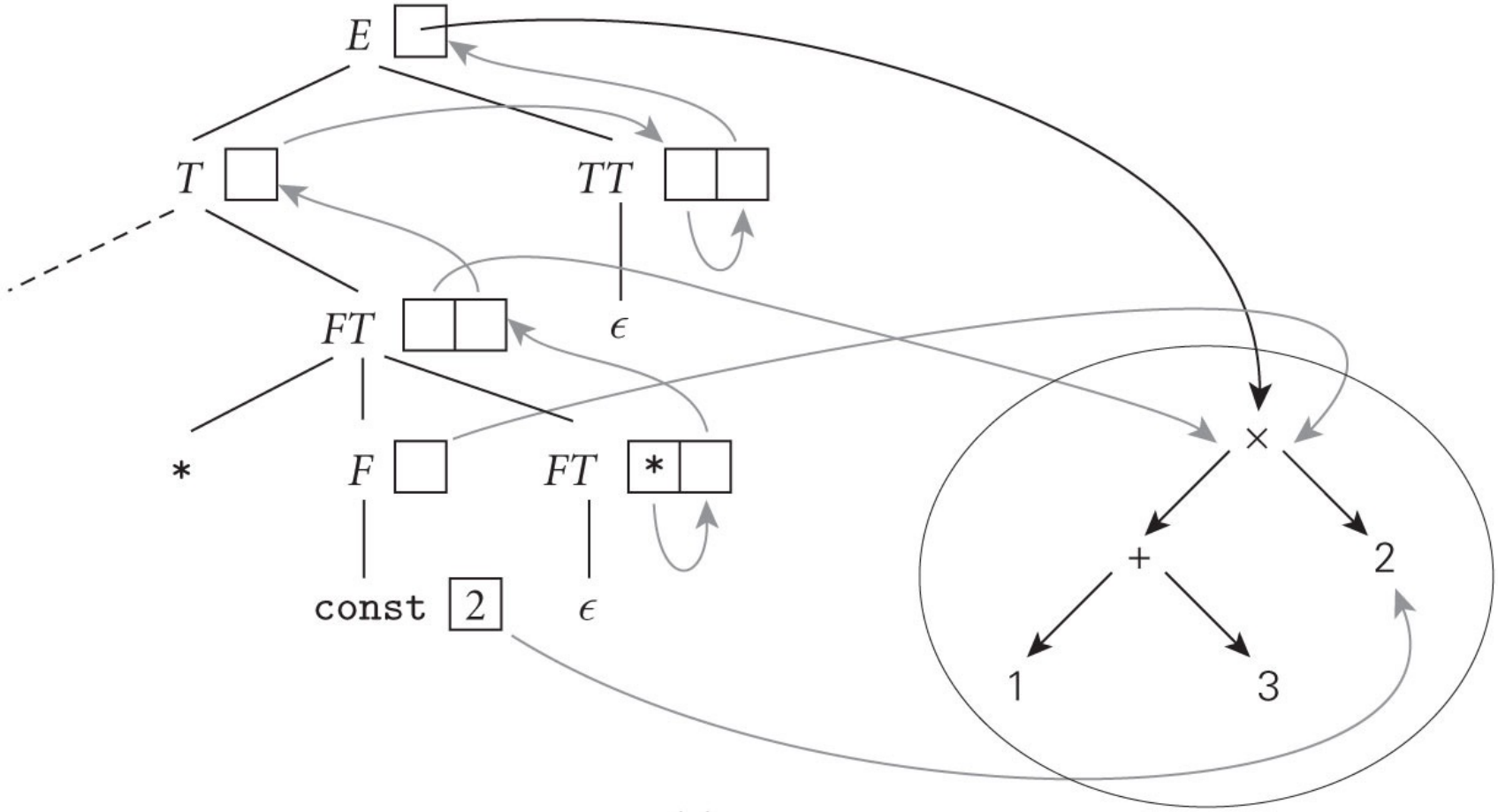
Syntax Trees Construction

- Syntax tree for `(1+3)*2`



Syntax Trees Construction

- Syntax tree for `(1+3)*2` (cont'd)



(c)

Action Routines

- There are automatic tools for:
  - Context-free grammar ⟹ parser
  - Attribute grammar ⟹ semantic analyzer (attrib. eval.)
- *Action routines*
  - ad-hoc approach; most ordinary compilers use (!)
  - Interleave parsing, syntax tree construction, other aspects of semantic analysis, code generation
  - Action routine: Semantic function that the programmer (grammar writer) instructs the compiler to execute at some point in the parse
  - In an LL grammar, can appear anywhere in the RHS; called as soon as the parser matched the (yield of the) symbol to the left

Action Routines - Example

- LL(1) grammar for expressions
  - with action routines for building the syntax tree
  - only difference from before: actions embedded in RHS

$E \longrightarrow T$ { 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) }

Action Routines - Example

- Recursive descent parsing with embedded action routines:

```
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
```

- does the same job as productions 2-4:

$TT_1 \longrightarrow$ + 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 }

Action Routines - Example

- Bottom-up evaluation
  - In LR-parser action routines cannot be embedded at arbitrary places in the RHS
  - the parser needs to see enough to identify the production, i.e., the RHS suffix that identifies the production uniquely
  - Previous bottom-up examples are identical with the action routine versions