Another Example: Constant Expressions

- CFG

```
E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → - F
F → ( E )
F → const
```

Note:

- Says nothing about the meaning of any **particular** program
- Conveys only potential structured sequence of tokens

Example Attribute Grammar

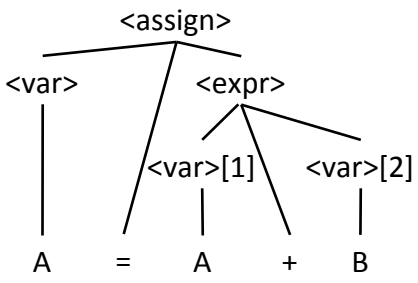
- Attribute: val
- Semantic Rules

$E_1 \rightarrow E_2 + T$	E1.val = E2.val + T.val
$E_1 \rightarrow E_2 - T$	E1.val = E2.val - T.val
$E \rightarrow T$	E.val = T.val
$T_1 \rightarrow T_2 * F$	T1.val = T2.val * F.val
$T_1 \rightarrow T_2 / F$	T1.val = T2.val / F.val
$T \rightarrow F$	T.val = F.val
$F_1 \rightarrow - F_2$	F1.val = - F2.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow const$	F.val = C.val

Evaluating Attributes

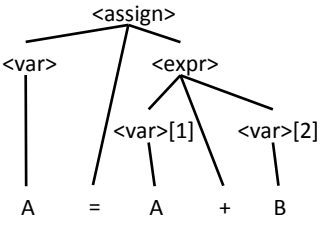
- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree
- If all attributes are inherited, the evaluation process can be done in a top-down order
- Alternatively, if all attributes are synthesized, the evaluation can proceed in a bottom-up order

An Example Parse Tree



- We have both inherited and synthesized attributes. In what direction should we proceed the computation ?

An Example Parse Tree



R1. <expr>.expected_type <- <var>.actual_type

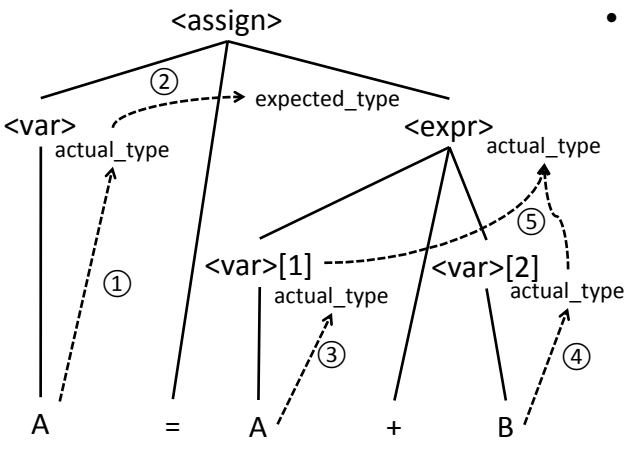
R2. <expr>.actual_type <- if (<var>[1].actual_type = int) and (<var>[2].actual_type = int)
then int
else real
end if
predicate: <expr>.actual_type == <expr>.expected_type

R3. <expr>.actual_type <- <var>.actual_type
predicate: <expr>.actual_type == <expr>.expected_type

R4. <var>.actual_type <- look-up(<var>.string)
The look-up function looks up a given variable name in the symbol table and returns the variable's type

1. <var>.actual_type <- look-up(A) (R4)
2. <expr>.expected_type <- <var>.actual_type (R1)
3. <var>[1].actual_type <- look-up(A) (R4)
   <var>[2].actual_type <- look-up(B) (R4)
4. <expr>.actual_type <- either int or real (R2)
5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE (R2)

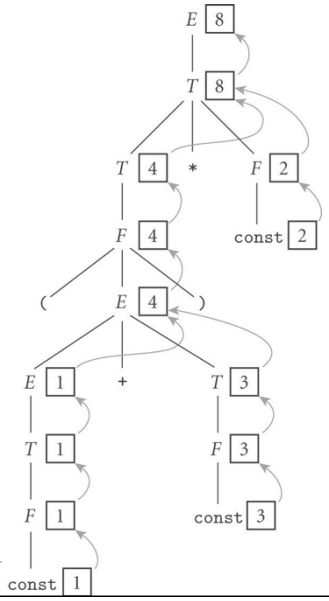
Attribute Evaluation Order



- Determining attribute evaluation order for any attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies

Decoration of a parse tree for (1 + 3) * 2

```
E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → - F
F → ( E )
F → const
```



A Third Example of Attribute Grammar

- CFG

```
expr -> const expr_tail
expr_tail -> -const expr_tail | ε
```

- What is the parse tree of 9 – 4 – 3?
- Can we accumulate the values of the overall expression into the root of the tree?

Insight

- We need to parse attribute values not only bottom-up, but also top-down and left-to-right in the tree

Attribute Grammar for Constant Expressions based on LL(1) CFG

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.va
3. $TT_1 \longrightarrow$ - $T\ TT_2$
   ▷ TT_2.st := TT_1.st − T.val ▷ TT_1.val := TT_2.va
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 × F.val ▷ FT_1.val := FT_2.va
7. $FT_1 \longrightarrow$ / $F\ FT_2$
   ▷ FT_2.st := FT_1.st ÷ F.val ▷ FT_1.val := FT_2.va
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

10

Attribute Grammar for CE LL(1) CFG

- Attributes
  - *st*: subtotal attribute to record intermediate evaluation result so far
  - *val*: value attribute to copy the right-most leaf back up to the root