STATIC SEMANTICS

Attribute Grammar

- A device used to describe more of the structure of a programming language than can be described with a context-free grammar
- It provides a formal framework for decorating parse trees
- An attribute grammar is an extension to a context-free grammar

Attribute Grammar

- The extension includes
  - Attributes
  - Attribute computation functions
  - Predicate functions

A Running Example

- Context-Free Grammar (CFG)

```
<assign> -> <var> = <expr>
<expr>   -> <var> + <var>
<expr>   -> <var>
<var>    -> A | B | C
```

- Note:
  - It only focuses on potential structured sequence of tokens
  - It says nothing about the meaning of any particular program

Attributes

- Associated with each grammar symbol X is a set of attributes A(X). The set A(X) consists of two disjoint sets: S(X) and I(X)

Attributes

- S(X): synthesized attributes, which are used to pass semantic information up a parse tree

Attributes

- I(X): inherited attributes, which pass semantic information down or across a tree. They are similar to variables because they can also have values assigned to them

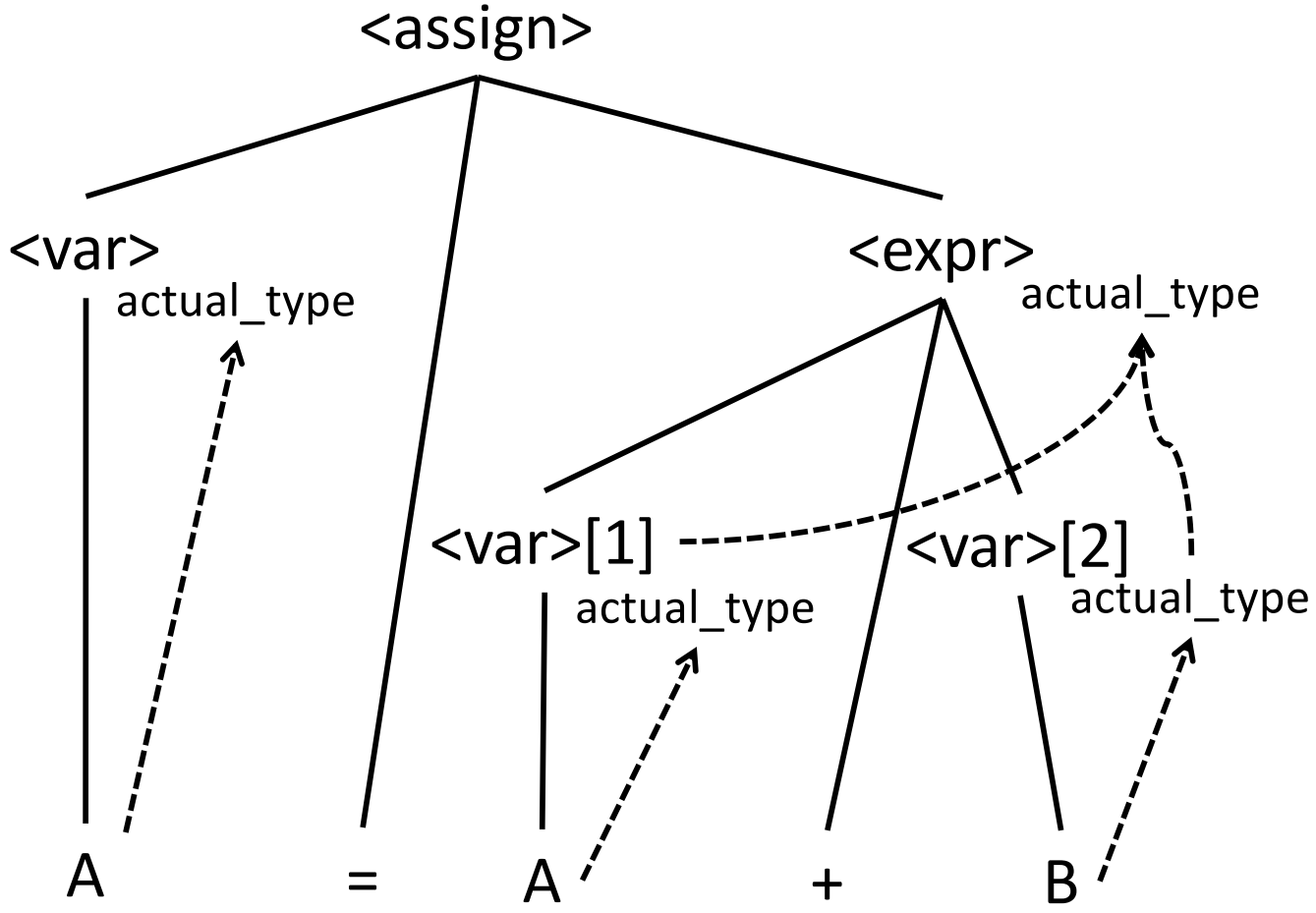
Intrinsic Attributes

- Synthesized attributes of leaf nodes whose values are determined outside the parse tree
  - E.g., the type of a variable can come from the symbol table
  - Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values

Example Synthesized Attribute

- *actual_type*
  - A synthesized attribute associated with nonterminals: <var> and <expr>
  - It is used to store the actual type, int or real, of a variable or expression
  - The attribute is determined by the actual types of children nodes

Evaluation Order of Synthesized Attribute *actual_type*

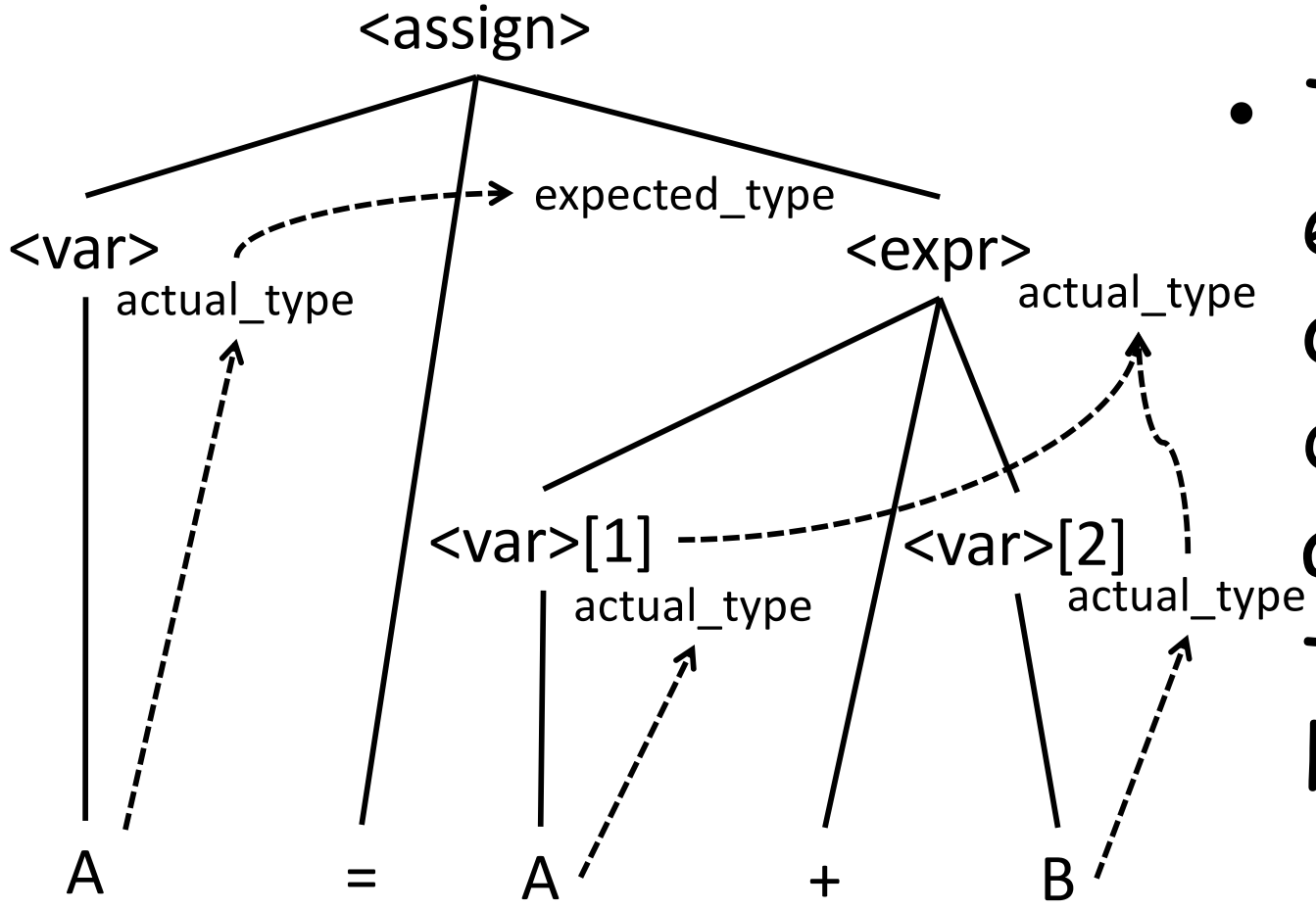


- Parser tree of A = A + B
- A and B have type "real" or "int" according to the symbol table

Example Inherited Attribute

- *expected_type*
  - An inherited attribute associated with the nonterminal <expr>
  - It is used to store the expected type, either int or real
  - It is determined by the type of the variable on the left side of the assignment statement

Evaluation Order of Inherited Attribute *expected_type*



- The expected_type of <expr> is decided by the actual_type of the assignment's left side

Attribute Grammar

- Defines the attributes, and attribute evaluation rules mentioned in the example

Example Attribute Grammar

Syntax Rule	Semantic Rule
<assign> -> <var> = <expr>	R1. <expr>.expected_type <- <var>.actual_type
<expr> -> <var>[1] + var[2]	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
<expr> -> <var>	R3. <expr>.actual_type <- <var>.actual_type predicate: <expr>.actual_type == <expr>.expected_type
<var> -> A | B | C	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

Semantic Functions

- Specify how attribute values are computed for S(X) and I(X)

Semantic Functions

- For a rule $X_0 \rightarrow X_1 \ldots X_n$, the synthesized attributes of X_0 are computed with semantic functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$

Semantic Functions

- Inherited attributes of symbols X_j, $1 \leq j \leq n$, are computed with a semantic function of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$
- To avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_{j-1}))$

Revisit the Semantic Functions

Syntax Rule	Semantic Rule
<assign> -> <var> = <expr>	1. **<expr>.expected_type <- <var>.actual_type**
<expr> -> <var>[1] + var[2]	2. **<expr>.actual_type <- if (<var>[1].actual_type = int) and (<var>[2].actual_type = int) then int else real end if**
<expr> -> <var>	3. **<expr>.actual_type <- <var>.actual_type**
<var> -> A | B | C	4. **<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**

Predicate Function

- A predicate function has the form of a Boolean expression on the union of the attribute set {$A(X_0)$, …, $A(X_n)$}, and a set of literal attribute values
- A false predicate function value indicates a violation of the syntax or static semantic rules

Example Semantic Rules & Predicates

Syntax Rule	Semantic Rule
<assign> -> <var> = <expr>	R1. <expr>.expected_type <- <var>.actual_type
<expr> -> <var>[1] + var[2]	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**
<expr> -> <var>	R3. <expr>.actual_type <- <var>.actual_type **predicate: <expr>.actual_type == <expr>.expected_type**
<var> -> A | B | C	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

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

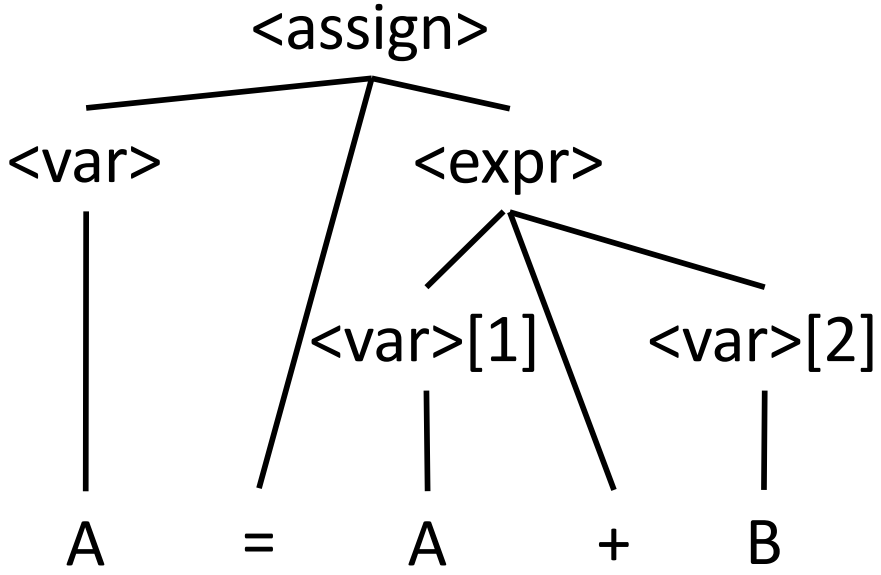
- Attribute: val
- Semantic Rules

$E_1 \rightarrow E_2 + T$	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

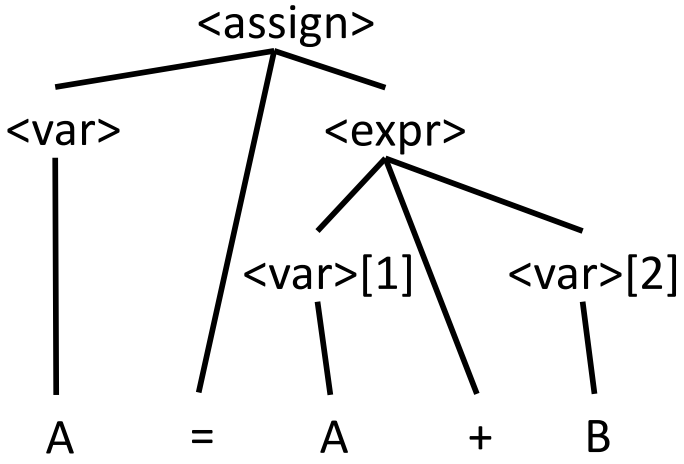
- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree
- If all attributes are inherited, the evaluation process can be done in a top-down order
- Alternatively, if all attributes are synthesized, the evaluation can proceed in a bottom-up order

An Example Parse Tree



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

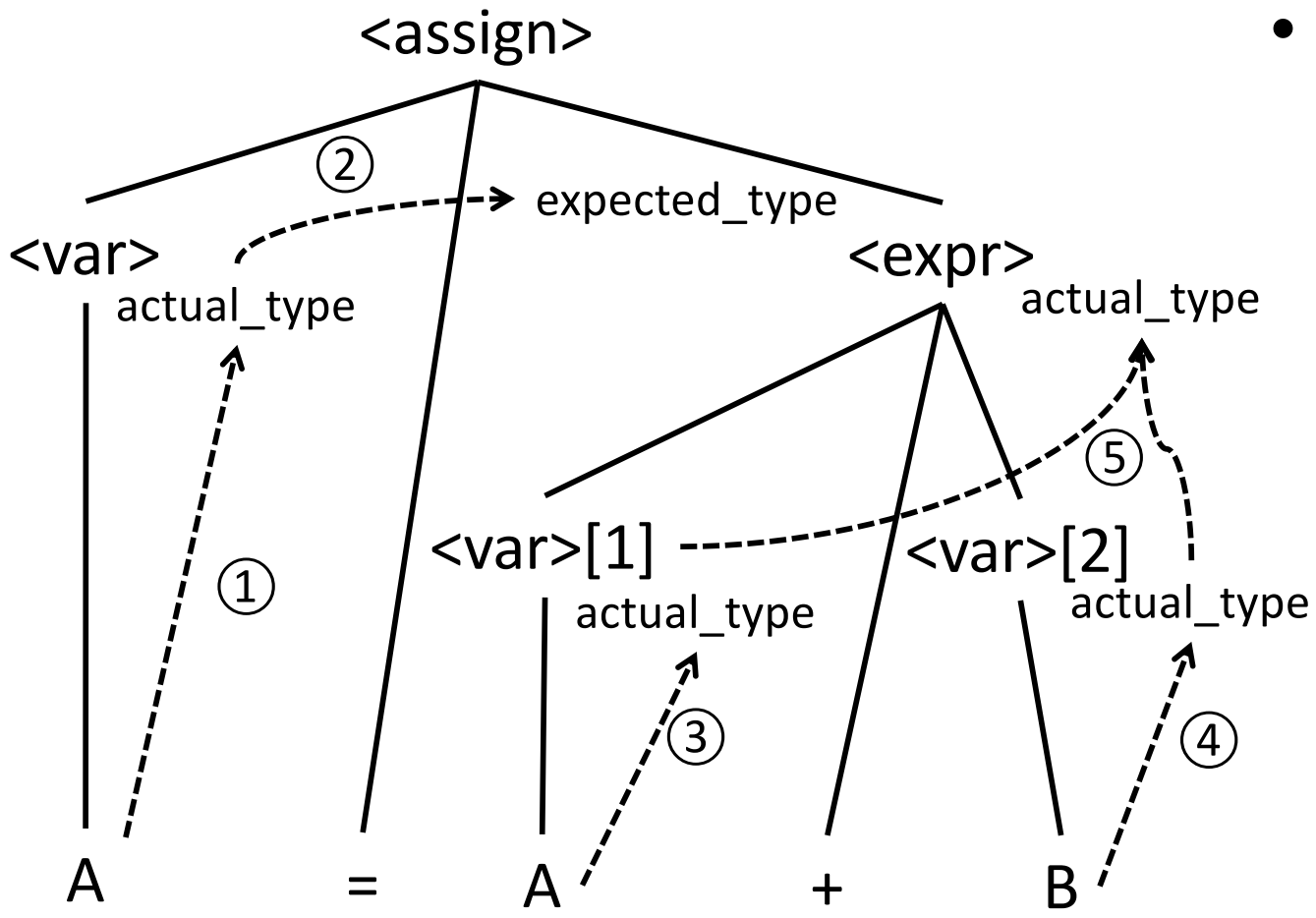
An Example Parse Tree



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)

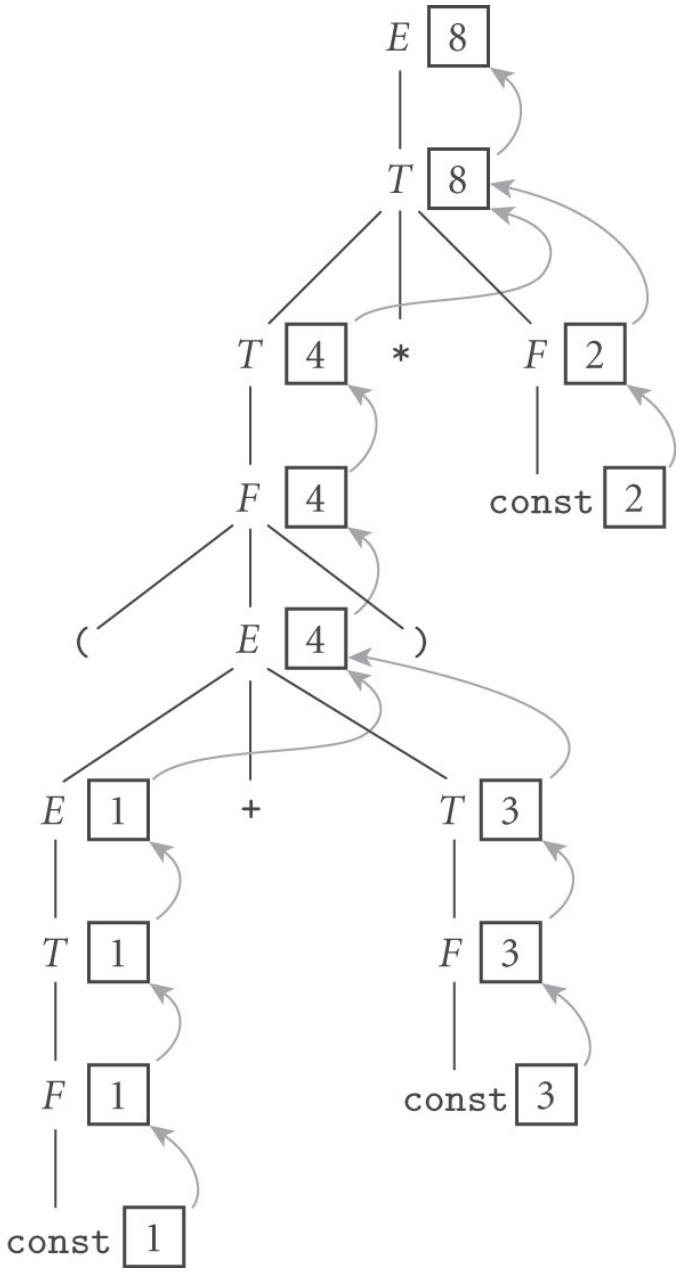
Attribute Evaluation Order



- 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.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
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

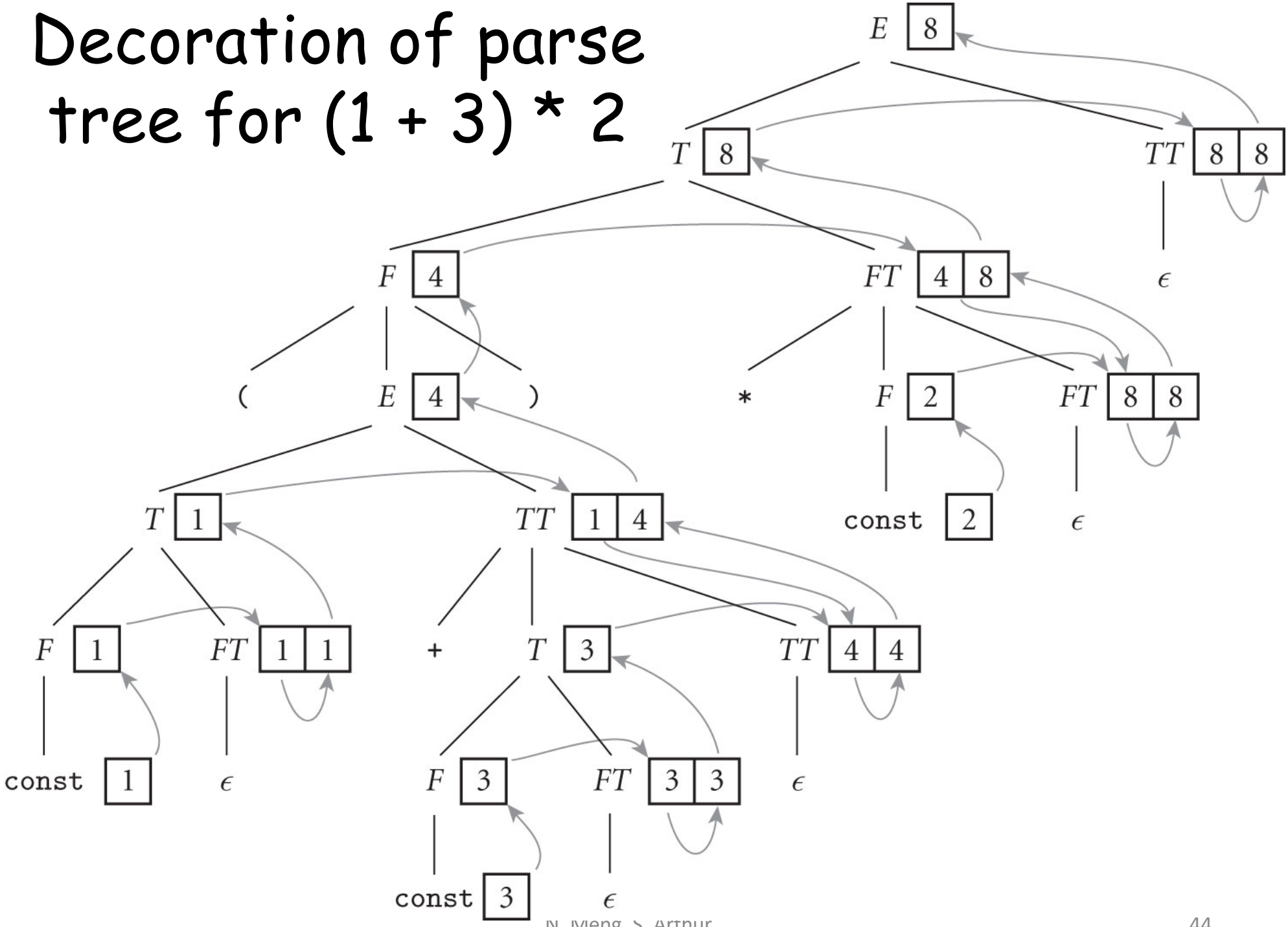
Attribute Grammar for CE LL(1) CFG

- 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

Decoration of (1 + 3) * 2

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
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

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



Program Assignment 2

- Bitwise Manipulation of Hexidecimal Numbers
- CFG

```
E → E "|" A     bitwise OR
E → A
A → A "^" B     bitwise XOR
A → B
B → B "&" C     bitwise AND
B → C
C → "<" C    bitwise shift left 1
C → ">" C    bitwise shift right 1
C → "~" C    bitwise NOT
C → "(" E ")"
C → hex
```

LL(1) Attribute Grammar

$E \rightarrow A\ EE$
$EE.st = A.val \qquad E.val = EE.val$

$EE_1 \rightarrow |\ A\ EE_2$
$EE_2.st = EE_1.st \mid A.val$ ("|" bitwise OR)
$EE_1.val = EE_2.val$

$EE \rightarrow \varepsilon$
$EE.val = EE.st$

$A \rightarrow B\ AA$
$AA.st = B.val \qquad A.val = AA.val$

$AA_1 \rightarrow \hat{}\ B\ AA_2$
$AA_2.st = AA_1.st \hat{}\ B.val$ ("^" bitwise XOR)
$AA_1.val = AA_2.val$

$AA \rightarrow \varepsilon$
$AA.val = AA.st$

$B \rightarrow C\ BB$
$BB.st = C.val \qquad B.val = BB.val$

$BB_1 \rightarrow \&\ C\ BB_2$
$BB_2.st = BB_1.st \ \&\ C.val$ ("&" bitwise AND)
$BB_1.val = BB_2.val$

$BB \rightarrow \varepsilon$
$BB.val = BB.st$

$C_1 \rightarrow <C_2$
$C_1.val = C_2.val << 1$ ("<<" bitwise shift left one)

$C_1 \rightarrow >C_2$
$C_1.val = C_2.val >> 1$ (">>" bitwise shift right one)

$C_1 \rightarrow \sim C_2$
$C_1.val = \sim C_2.val$ ("~" bitwise NOT)

$C \rightarrow (\ E\ )$
$C.val = E.val$

$C \rightarrow hex$
$C.val = hex.val$

Program Requirement

- Write a C program using recursive descent parser w/ lexical analyzer to implement the designated inherited and synthesized attributes. The program evaluates the expressions in a file input.txt, and outputs the results to console
- E.g., input: f&a

  output: f&a = a

Program Requirements

- You cannot use more than 2 global/non-local variables, and they should be to hold the Operator and HexNumber as detected by the lexical analyzer

Hints

- To solve the problems, you should take the following steps:
  - Write a lexical analyzer
  - Write a recursive-descent parser
  - Attributes are processed as either pass-in parameters or return value of functions

Hints

- Write a lexical analyzer
  - You may need to define an enum type for all possible tokens your scanner can generate
  - E.g., when reading hexadecimial numbers 0-9 or a-f, the recognized token is HEX, and the value is saved in HexNumber

Hints

- Write a recursive-descent parser
  - Parse the program by defining and invoking functions
  - E.g., **E → A EE**

    EE.st = A.val E.val = EE.val

```
int E() {
   int val = A();
   return EE(val);
}
```

Hints

- There are parameters passed in or returned when invoking functions. When invoking a function, the synthesized attribute is the return value, while the inherited attribute is the passing-in parameter

Hints

- Sample code of main()

```
int main() {
    int val;
    symbol = getNextToken();
    while (symbol != EOF_) {
        if (symbol != NEW_LINE) {
            val = E();
             printf(" = %x\n", val & 0xf);
        }
        if (symbol == EOF_) break;
        symbol = getNextToken();
    }
    return 1;
}
```

Submission Requirements

- Pack the following files into a .tar file:
  - Source file: parser.c
  - Executable file: parser
  - Input file: input.txt
  - Output file: output.txt (copy all your console outputs to this file)
  - README file (optional, used if you have any additional comments/explanations about the files)