### **Q1.1. Context-Free Grammar (CFG) for Arithmetic Expressions**

In this task, we need to define a **CFG** that can correctly parse arithmetic expressions with addition, subtraction, multiplication, and division. The grammar should enforce the precedence and associativity of these operators, with multiplication and division having higher precedence than addition and subtraction.

#### **Precedence and Associativity Rules:**

* **Addition (+)** and **subtraction (-)** are of lower precedence.
* **Multiplication (\*)** and **division (/)** are of higher precedence.
* All operators are **left-associative** (i.e., operations are evaluated from left to right).

#### **CFG Rules:**

Let’s define the following CFG for arithmetic expressions:

1. **E → E + T**This rule states that an expression (E) can be another expression (E) followed by the addition of a term (T).
2. **E → E - T**This rule represents an expression (E) followed by subtraction of a term (T).
3. **E → T**This rule states that an expression (E) can be just a term (T).
4. **T → T \* F**This rule represents a term (T) followed by multiplication of a factor (F).
5. **T → T / F**This rule represents a term (T) followed by division of a factor (F).
6. **T → F**This rule states that a term (T) can also just be a factor (F).
7. **F → ( E )**This rule represents a factor that is an expression (E) inside parentheses.
8. **F → id**This rule handles a factor (F) being an identifier or a literal value (number).

#### 

#### **LR Parsability:**

The above grammar is suitable for **LR parsing** for the following reasons:

* **Operator precedence** is maintained by splitting the rules for expressions (E), terms (T), and factors (F). This ensures that addition and subtraction are handled at the top level (E), and multiplication and division are handled at a lower level (T), giving them higher precedence.
* The grammar is **unambiguous** because:
  + It does not lead to **shift-reduce** conflicts, meaning the LR parser can always decide whether to shift or reduce at any point.
  + There are no **reduce-reduce** conflicts, meaning the parser can always choose one valid reduction.
  + The grammar properly distinguishes between different operator precedences and associativity, making it ideal for **LR parsing**.

### 

### 

### 

### **Q1.2. Attribute Grammar for Evaluating Arithmetic Expressions**

To compute the value of an arithmetic expression during parsing, we extend the CFG with **attributes** and **semantic rules**. In this case, we will use **synthesized attributes** to propagate the computed values upward from the leaves of the parse tree.

#### **Attribute Definitions:**

* For each non-terminal (E, T, F), we define a synthesized attribute val that will store the result of the evaluated expression.

#### **Attribute Grammar Rules:**

1. **E → E₁ + T**
   * **Semantic Rule:** E.val = E₁.val + T.val  
     This rule states that the value of E is the sum of E₁ and T.
2. **E → E₁ - T**
   * **Semantic Rule:** E.val = E₁.val - T.val  
     This rule states that the value of E is the result of E₁ minus T.
3. **E → T**
   * **Semantic Rule:** E.val = T.val  
     This rule propagates the value of T to E when E consists only of T.
4. **T → T₁ \* F**
   * **Semantic Rule:** T.val = T₁.val \* F.val  
     This rule states that the value of T is the product of T₁ and F.
5. **T → T₁ / F**
   * **Semantic Rule:** T.val = T₁.val / F.val  
     This rule states that the value of T is the result of T₁ divided by F.
6. **T → F**
   * **Semantic Rule:** T.val = F.val  
     This rule propagates the value of F to T when T consists only of F.
7. **F → ( E )**
   * **Semantic Rule:** F.val = E.val  
     This rule states that the value of F is the same as the value of the expression inside the parentheses.
8. **F → id**
   * **Semantic Rule:** F.val = id.lexval  
     This rule assigns the lexical value of the identifier (or number) to F.

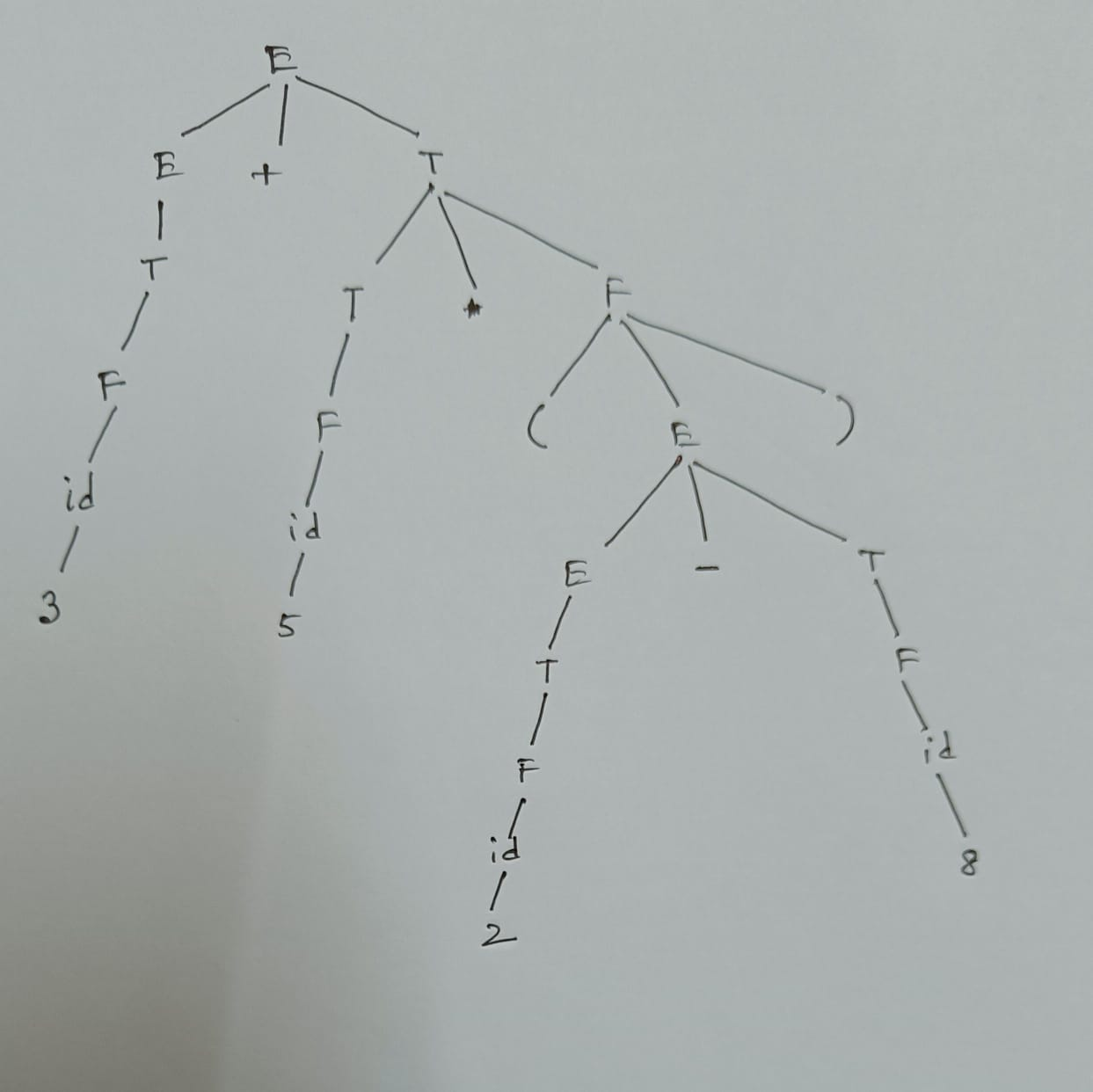
#### **Explanation:**

* In this **attribute grammar**, each non-terminal symbol carries a value (val) that is computed based on its children. These values are propagated upwards from the leaves (literals and identifiers) to the root (the complete expression).
* The **semantic rules** define how these values are computed for each production.

### **Example: Evaluating 3 + 5 \* (2 - 8)**

Let’s step through how the attribute grammar works on the expression 3 + 5 \* (2 - 8).

#### **Parse Tree:**

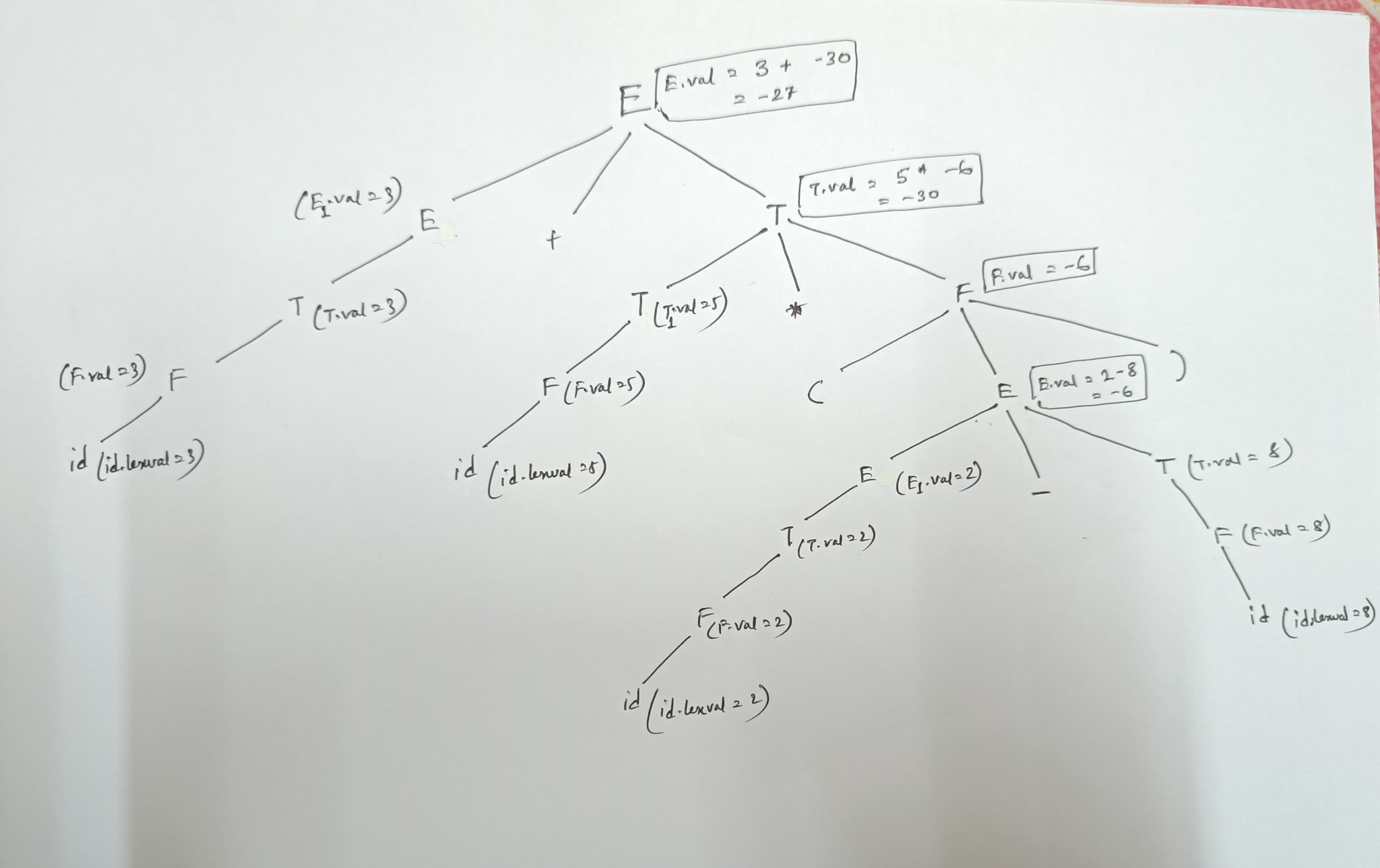


#### **Semantic Evaluation:**

1. **Bottom-up Evaluation**: Start from the leaves:
   * id(3).lexval = 3
   * id(5).lexval = 5
   * id(2).lexval = 2
   * id(8).lexval = 8
2. **Evaluate Inner Expression**:
   * Evaluate the expression inside the parentheses:  
     E.val = 2 - 8 = -6
3. **Multiply**:
   * Now, calculate the multiplication:  
     T.val = 5 \* -6 = -30
4. **Add**:
   * Finally, add 3 + (-30):  
     E.val = 3 + (-30) = -27

So, the result of 3 + 5 \* (2 - 8) is **-27**.

### **Parse Tree with Attributes:**



### **Dependency Graph:**

