1. **Three-address code (TAC) is an intermediate representation that simplifies complex expressions into statements with at most one operator on the RHS, aiding optimization and code generation. The procedure involves tokenizing the input using LEX, parsing it with YACC, and applying semantic actions to generate TAC for each expression. The code includes LEX rules for identifiers, numbers, and operators, and YACC grammar to implement TAC generation. For example, the expression a = b + c \* d produces TAC: t1 = c \* d; t2 = b + t1; a = t2. The output displays the generated TAC, validating correct intermediate representation.**

**Three-Address Code (TAC) Generation**

Three-address code (TAC) is an intermediate representation in compiler design.  
It breaks down complex expressions into simple statements, where the right-hand side contains at most one operator.  
This makes it suitable for optimization and code generation.

**Example**  
Input expression:

a = b + c \* d

TAC:

t1 = c \* d

t2 = b + t1

a = t2

**Purpose of TAC**

* Provides a machine-independent representation.
* Simplifies the process of optimization.
* Makes code generation easier.

**Steps in TAC generation**

**a) Lexical analysis (LEX)**

Lexical analysis scans the program and converts it into tokens.

[a-zA-Z][a-zA-Z0-9]\* return ID; // identifier

[0-9]+ return NUM; // number

"+" return PLUS; // addition

"-" return MINUS; // subtraction

"\*" return MUL; // multiplication

"/" return DIV; // division

"=" return ASSIGN; // assignment

**Example tokens for** a = b + c \* d:

ID(a), =, ID(b), +, ID(c), \*, ID(d)

**b) Syntax analysis (YACC)**

Syntax analysis checks grammar and builds a parse tree or abstract syntax tree (AST).  
It ensures operator precedence and associativity.

**Grammar rules example**

E → E + T | T

T → T \* F | F

F → ID | NUM

stmt : ID ASSIGN expr { $$ = generate\_code($1, $3); };

expr : expr PLUS term { $$ = new\_temp(); emit($$, $1, '+', $3); }

| expr MINUS term { $$ = new\_temp(); emit($$, $1, '-', $3); }

| term { $$ = $1; };

term : term MUL factor { $$ = new\_temp(); emit($$, $1, '\*', $3); }

| term DIV factor { $$ = new\_temp(); emit($$, $1, '/', $3); }

| factor { $$ = $1; };

factor : ID { $$ = $1; }

| NUM { $$ = $1; };

* Each operation generates a **temporary variable** for intermediate results.
* The emit function outputs the TAC instruction.

**c) Semantic actions**

Semantic actions are applied during parsing to generate TAC.  
They introduce temporary variables (t1, t2, …) for intermediate results.

**Example actions**

* Multiplication → t1 = c \* d
* Addition → t2 = b + t1
* Assignment → a = t2

**Expression tree diagram**

For input a = b + c \* d:

=

/ \

a +

/ \

b \*

/ \

c d

**Flowchart of TAC generation**

Input Expression

│

▼

Tokenization

(LEX scans input)

│

▼

Parsing

(YACC checks grammar)

│

▼

Semantic Actions

(generate TAC using temporaries)

│

▼

Output TAC

1. **Type checking ensures that operations are applied to compatible types, detecting mismatches at compile time and preventing semantic errors. The procedure includes tokenizing identifiers and data types using LEX, parsing statements with YACC, and checking type compatibility through semantic actions. The code defines LEX tokens for types and identifiers, and YACC grammar implements type rules for assignments and expressions. For example, input int a; float b; a = b + 5; results in a type mismatch. The output shows an error message indicating incompatibility, ensuring semantic correctness.**

**Type Checking**   
Type checking is a **semantic analysis process** in compiler design that ensures **operations are applied to compatible data types**. It detects **type mismatches at compile time**, preventing semantic errors and ensuring **program correctness**.

**Purpose:**

1. Detect **incompatible operations** (e.g., adding a float to an int).
2. Enforce **type rules** for assignments and expressions.
3. Enable **early error detection** before code generation.
4. Facilitate **optimized and safe code generation**.

**Importance:**

1. **Early Error Detection:** Detects errors like assigning a float to an int before runtime.
2. **Semantic Correctness:** Ensures that operations are meaningful according to the language rules.
3. **Optimization Support:** Knowing operand types allows type-specific optimizations.
4. **Supports Overloading and Polymorphism:** Helps resolve correct function/operator usage.

**Types of Type Checking:**

* **Static Type Checking:** Done at compile time (e.g., C, Java).
* **Dynamic Type Checking:** Done at runtime (e.g., Python, JavaScript).

**Type System Concepts:**

* **Primitive Types:** int, float, char, boolean.
* **Composite Types:** arrays, structures, records.
* **Type Conversion:** implicit (coercion) or explicit (casting).
* **Type Compatibility:** Rules defining which types can be combined or assigned.

**Procedure of Type Checking**

**1. Tokenization (LEX)**

* Identify **keywords, identifiers, data types, numbers, and operators**.

**Sample LEX rules:**

int return INT; // integer type

float return FLOAT; // float type

[a-zA-Z][a-zA-Z0-9]\* return ID; // identifiers

[0-9]+ return NUM; // numbers

"+" return PLUS; // addition

"=" return ASSIGN; // assignment

**2. Parsing (YACC)**

* Use grammar rules to parse **declarations, assignments, and expressions**.
* Apply **semantic actions** to check type compatibility.

**Sample YACC grammar with type rules:**

declaration : type ID { add\_symbol($2, $1); };

type : INT { $$ = "int"; }

| FLOAT { $$ = "float"; };

assignment : ID ASSIGN expr {

if (type\_of($1) != $3)

printf("Type mismatch error\n");

};

expr : expr PLUS term { $$ = compatible\_type($1, $3); }

| term { $$ = $1; };

term : ID { $$ = type\_of($1); }

| NUM { $$ = "int"; };

* add\_symbol() – stores variable and type in the **symbol table**.
* type\_of() – retrieves type of a variable.
* compatible\_type() – determines expression type or reports mismatch.

**Input Code:**

int a;

float b;

a = b + 5;

1. **Parse Declarations:**
   * int a; → add a as int in symbol table
   * float b; → add b as float in symbol table
2. **Parse Assignment:**
   * b + 5 → float + int → result is **float**
   * a = b + 5 → assigning float to int → **type mismatch**

**Output:**

Type mismatch error: cannot assign float to int 'a'

**Flowchart of Type Checking**

Input Source Code

│

▼

Tokenization (LEX)

│

▼

Parsing (YACC)

│

▼

Semantic Analysis

- Lookup types in symbol table

- Check type compatibility

│

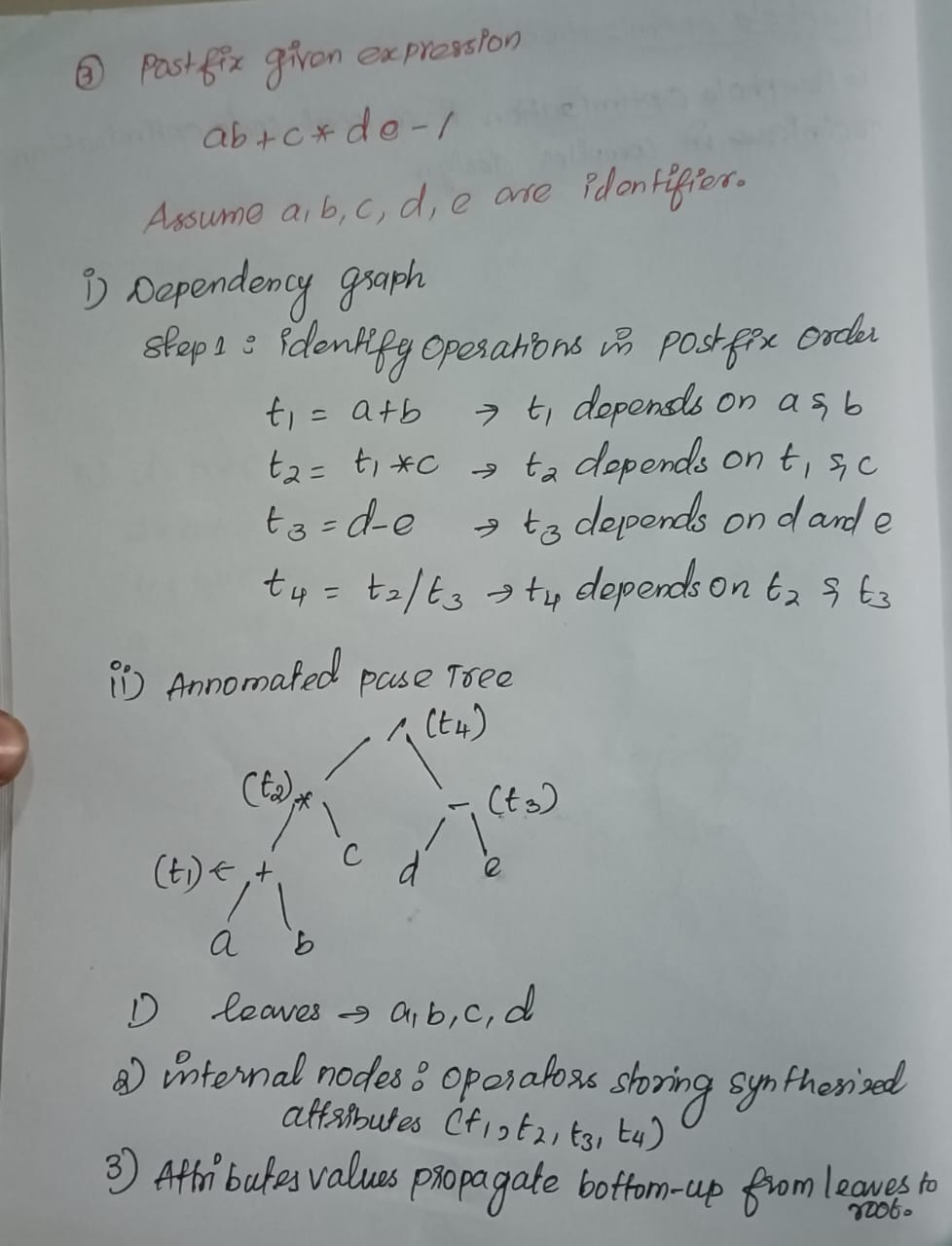
▼

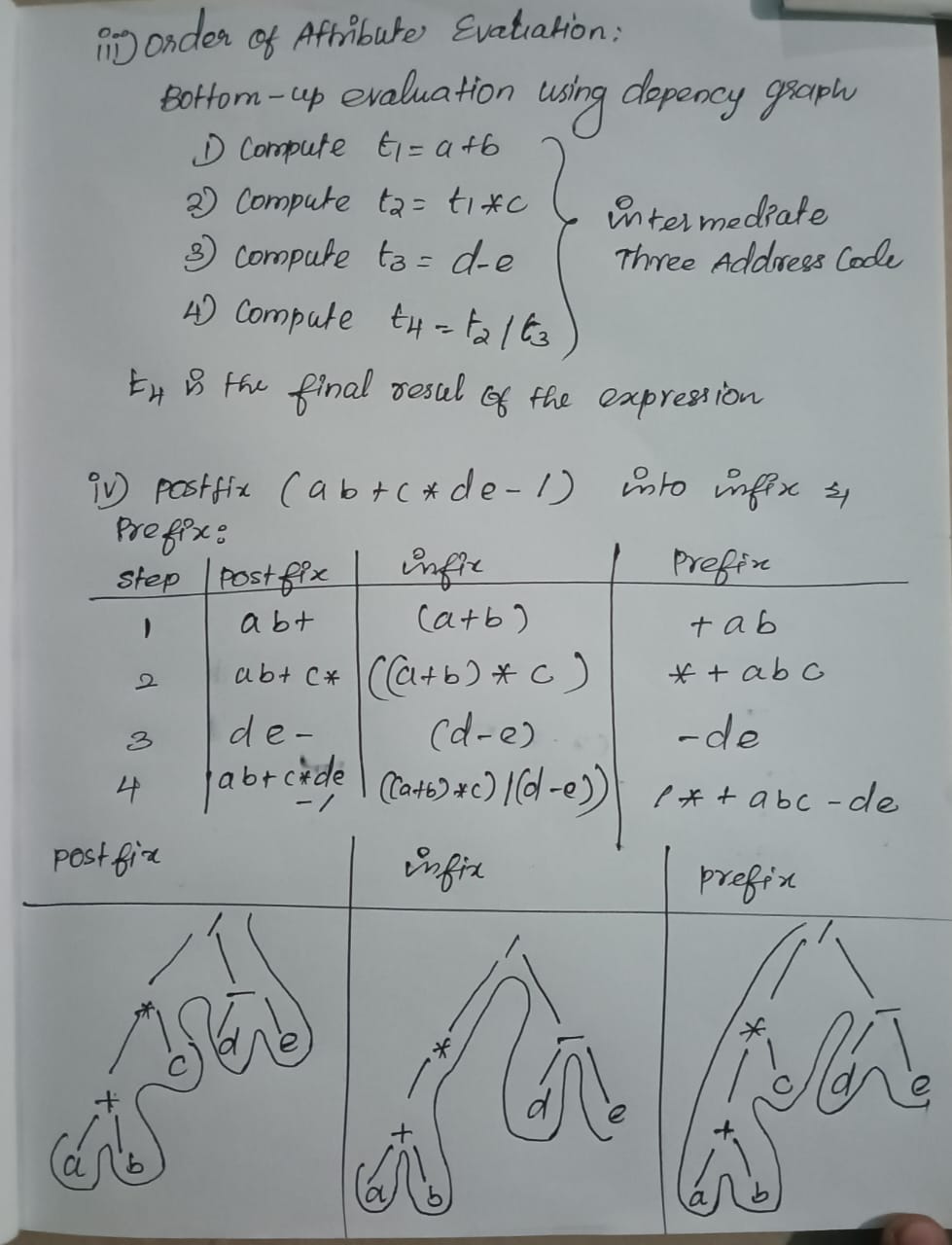
Output

- Error (if mismatch)

- Success (if types match)

1. **Demonstrate attribute evaluation for different expression notations.  
   i. Construct the dependency graph for the postfix expression ab+c\*de-/.  
   ii. Draw the annotated parse tree showing synthesized attributes.**
   * 1. **Show the order of attribute evaluation using the dependency graph.  
        iv Generate intermediate three-address code for the expression.  
        v. Convert the same expression into infix and prefix forms.**

****



1. **Construct the annotated parse tree for the arithmetic expression:**

**(3 + 4) \* (5 + 6)**

**using the following grammar / productions:**

**L → E**

**E → E + T | T**

**T → T \* F | F**

**F → (E) | DIGIT**

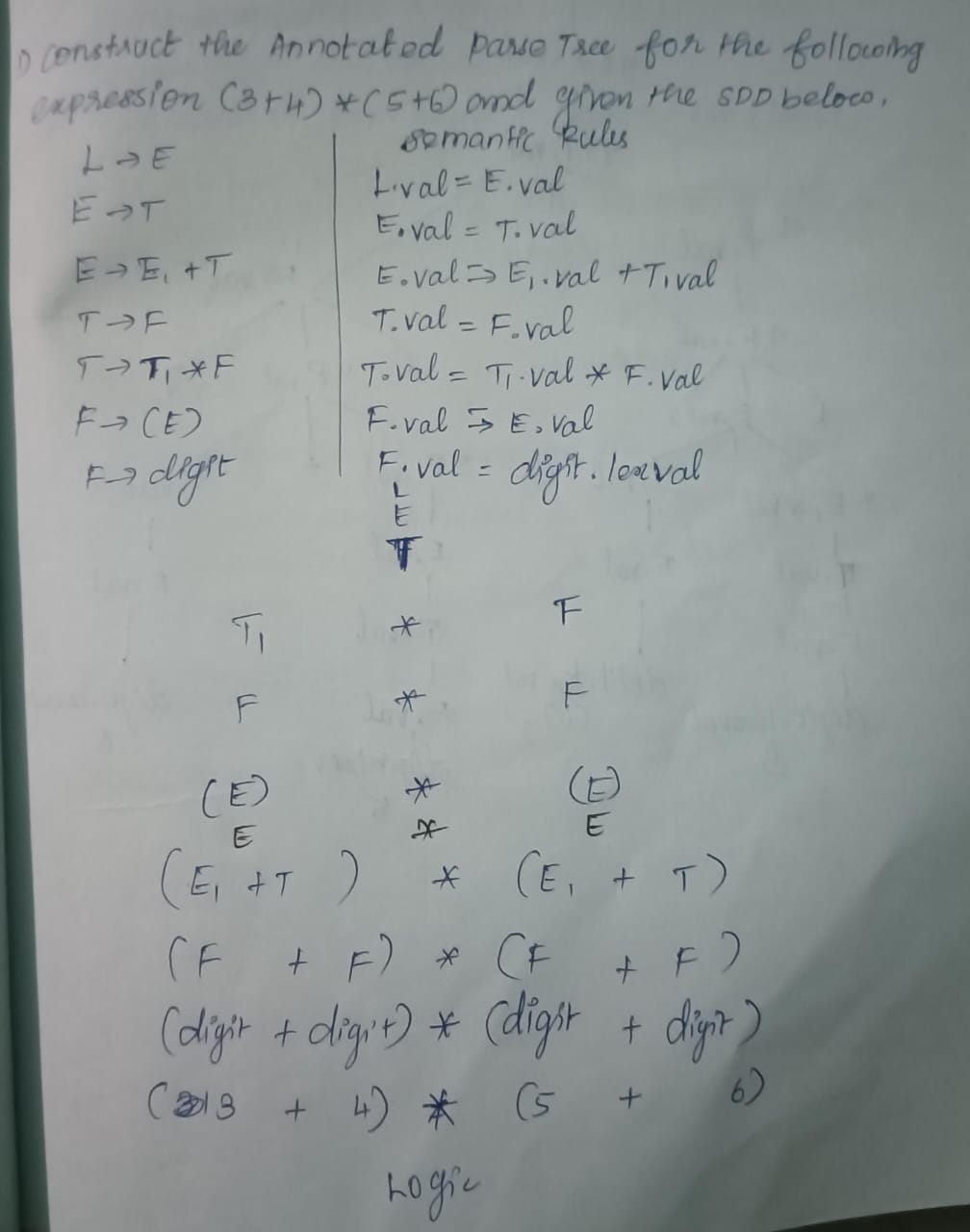
**i. Draw the parse tree for (3 + 4) \* (5 + 6) with top node L → E.**

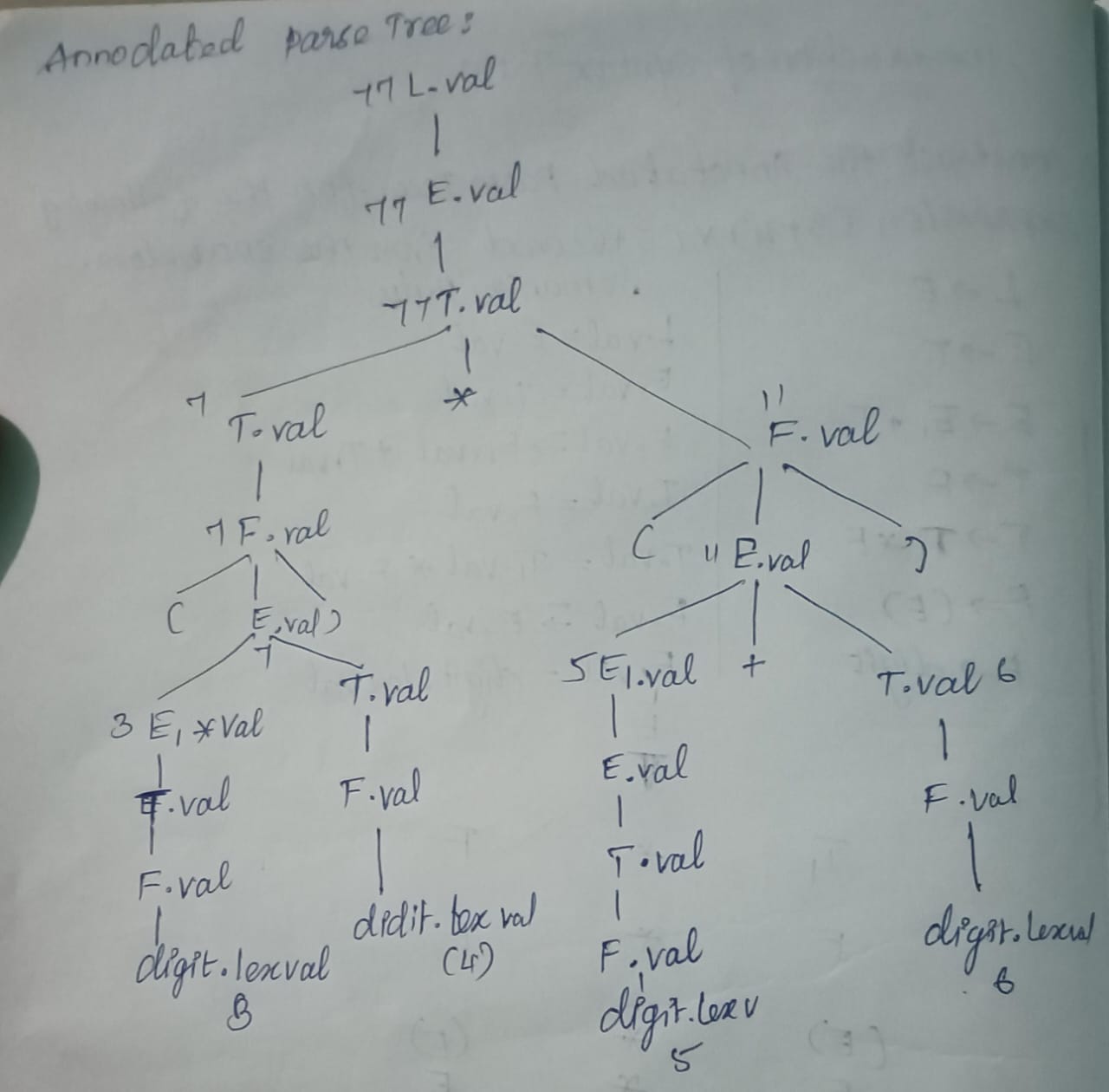
**ii. Annotate each node of the parse tree with computed values using bottom-up evaluation.**

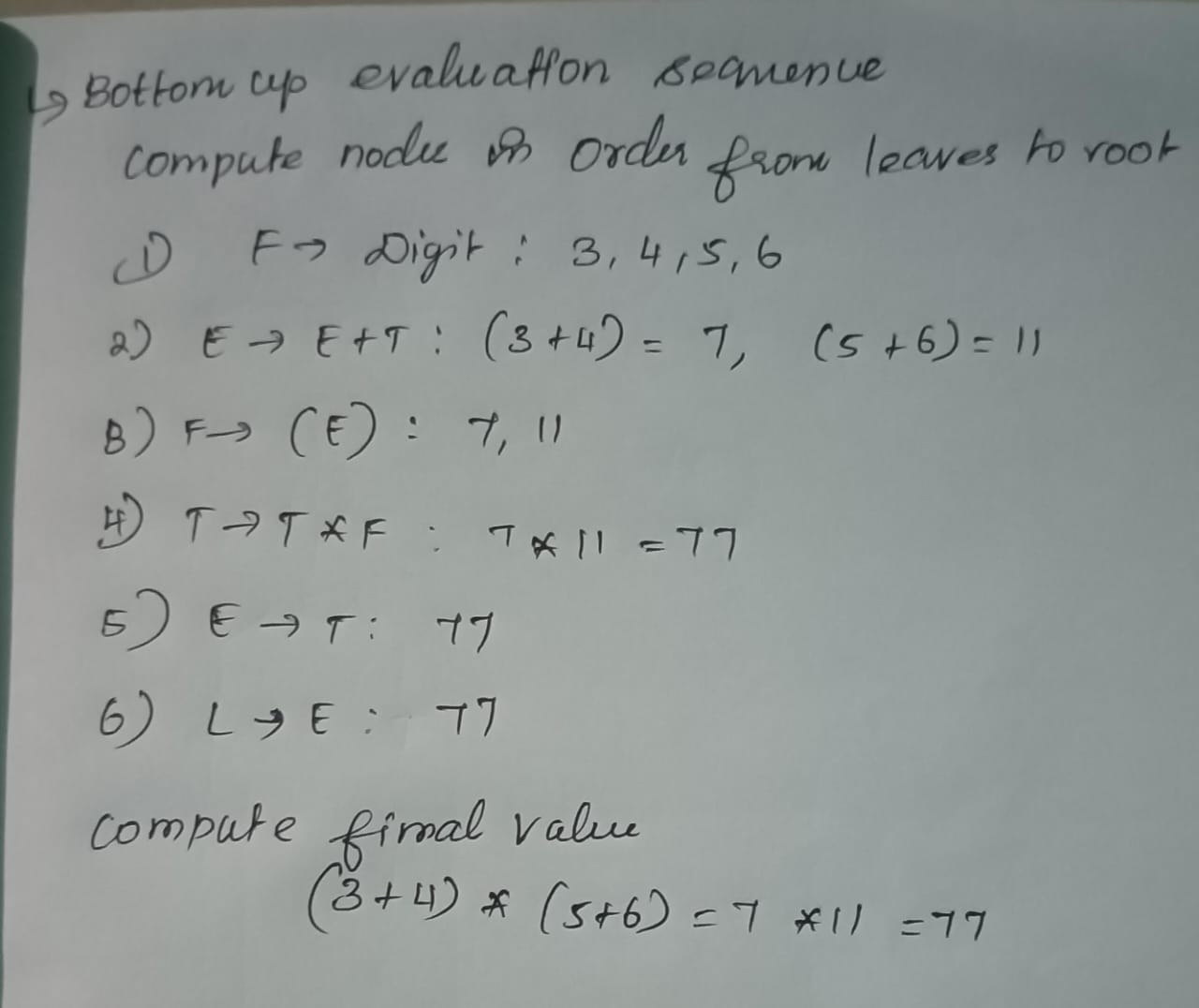
**iii. Construct the dependency graph showing the order of attribute evaluation.**

**iv. Show the bottom-up evaluation sequence of all nodes.**

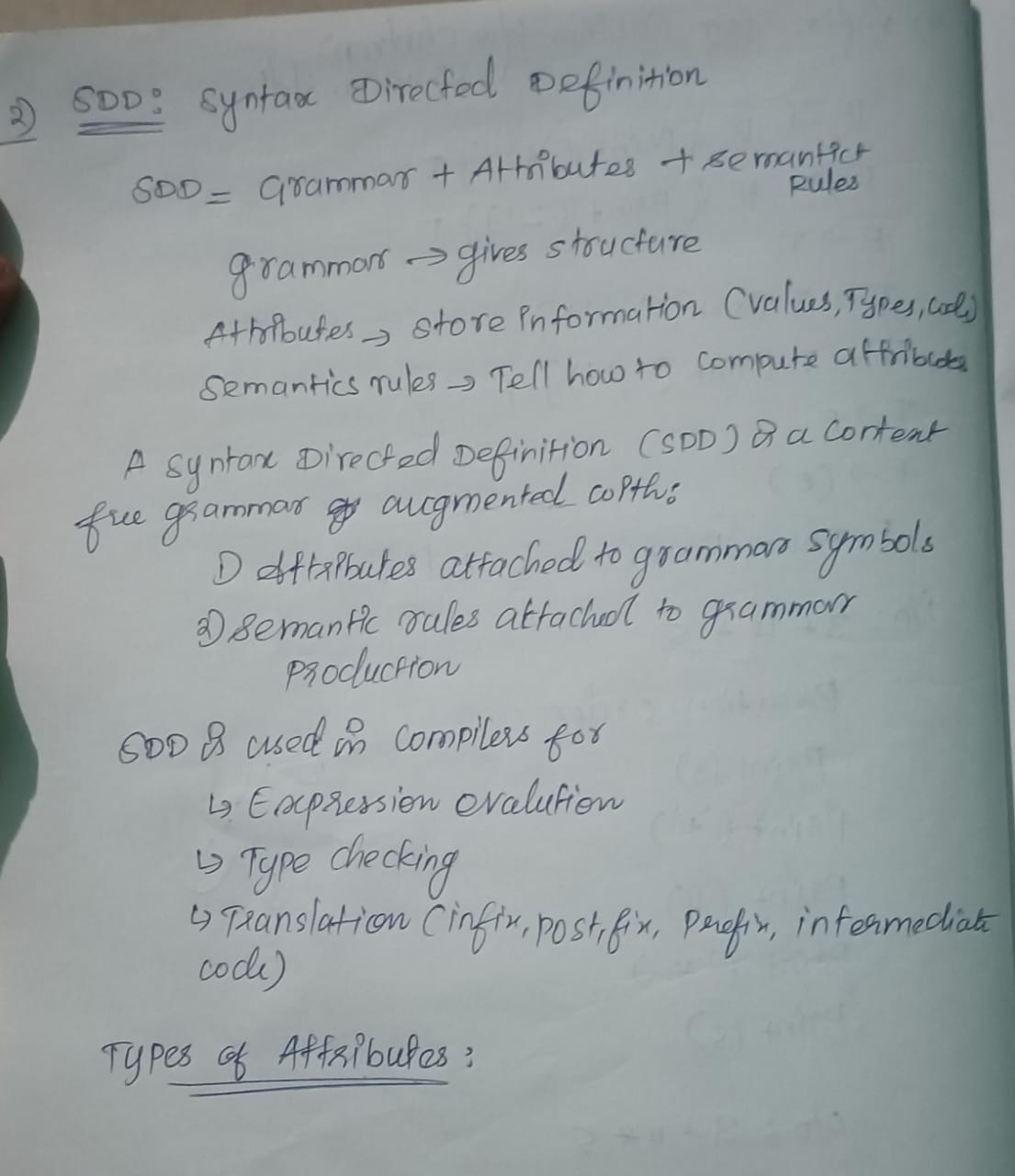
**v. Compute the final value of the expression (3 + 4) \* (5 + 6) using the annotated parse tree.**

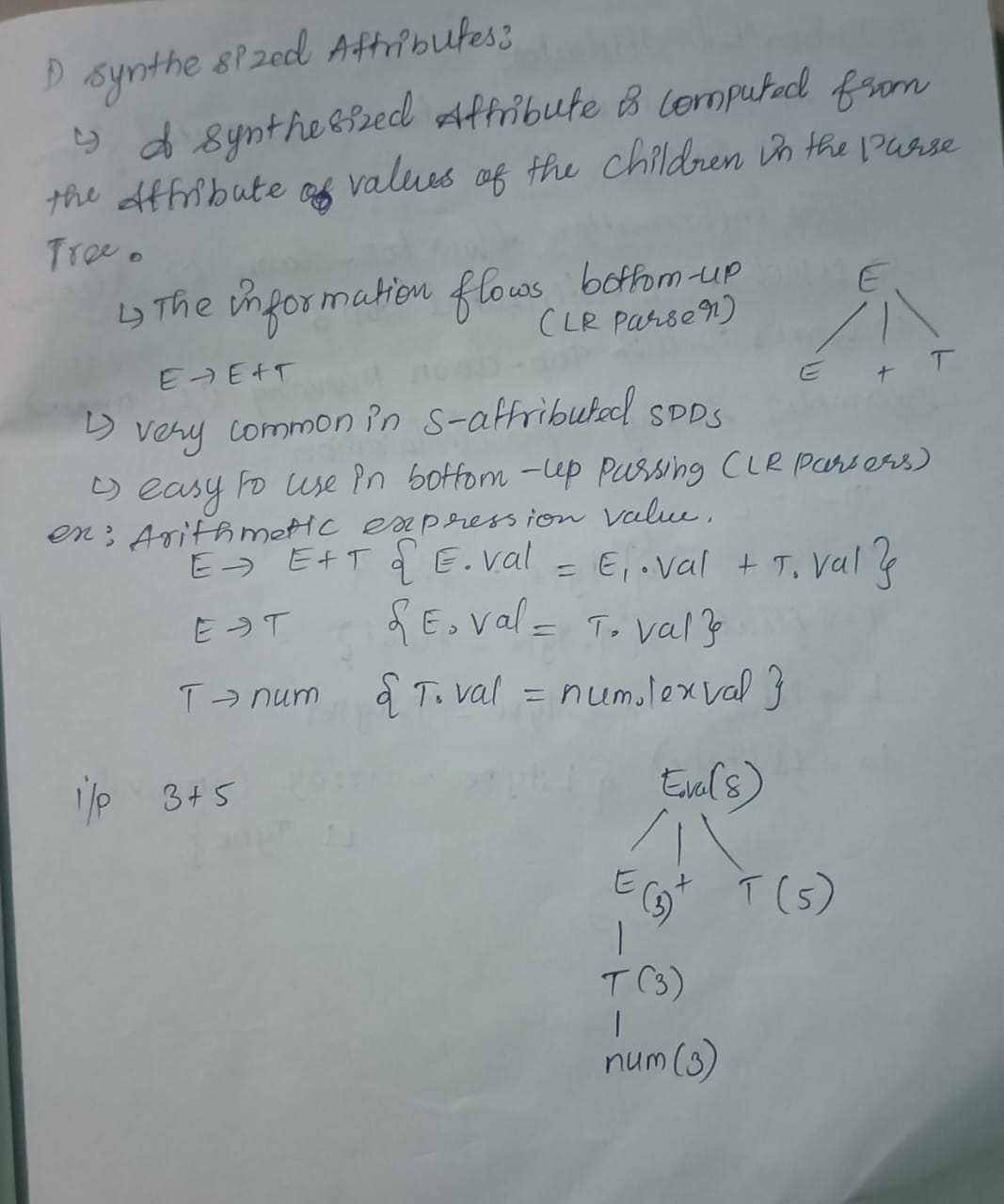
****

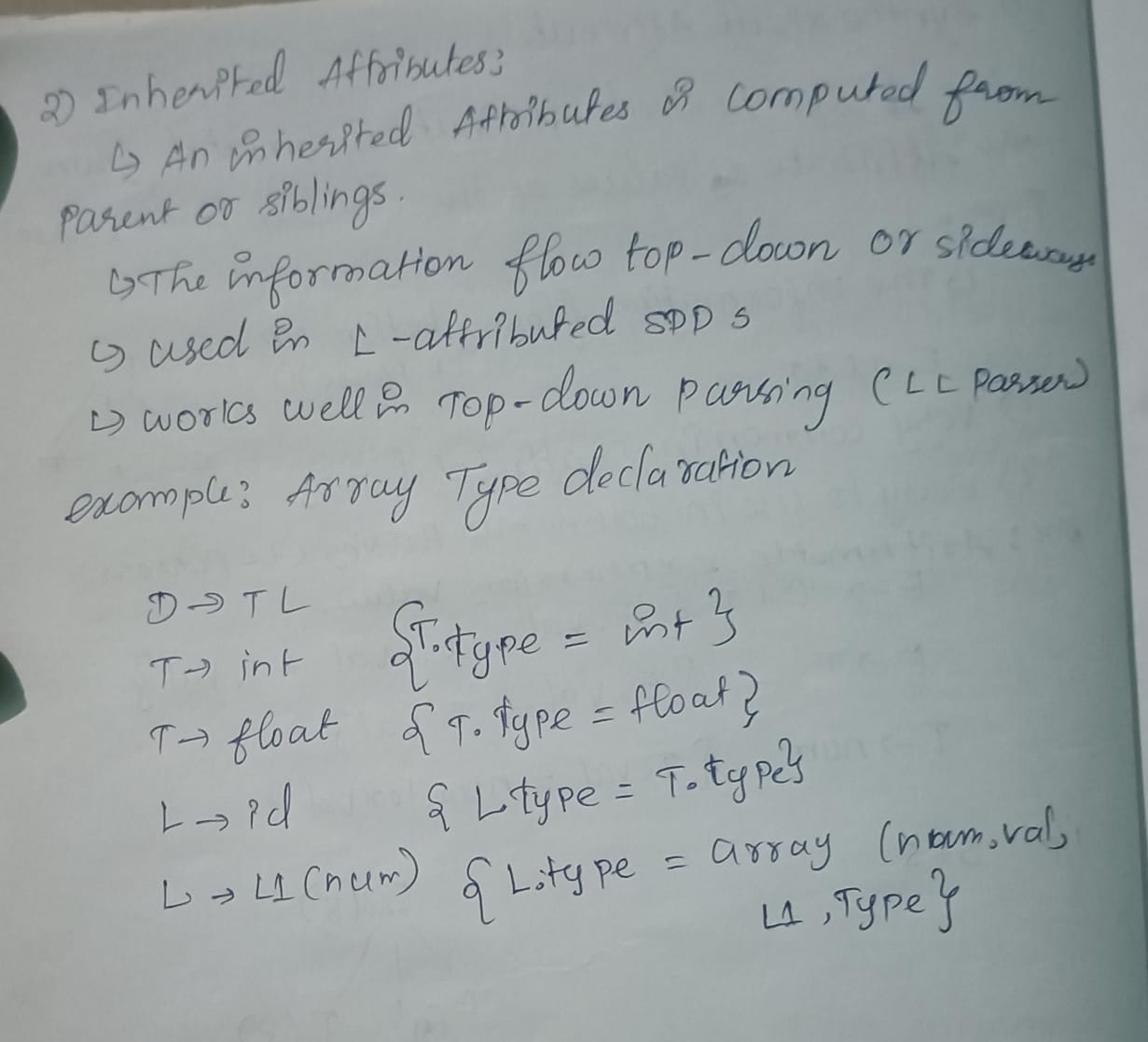
****

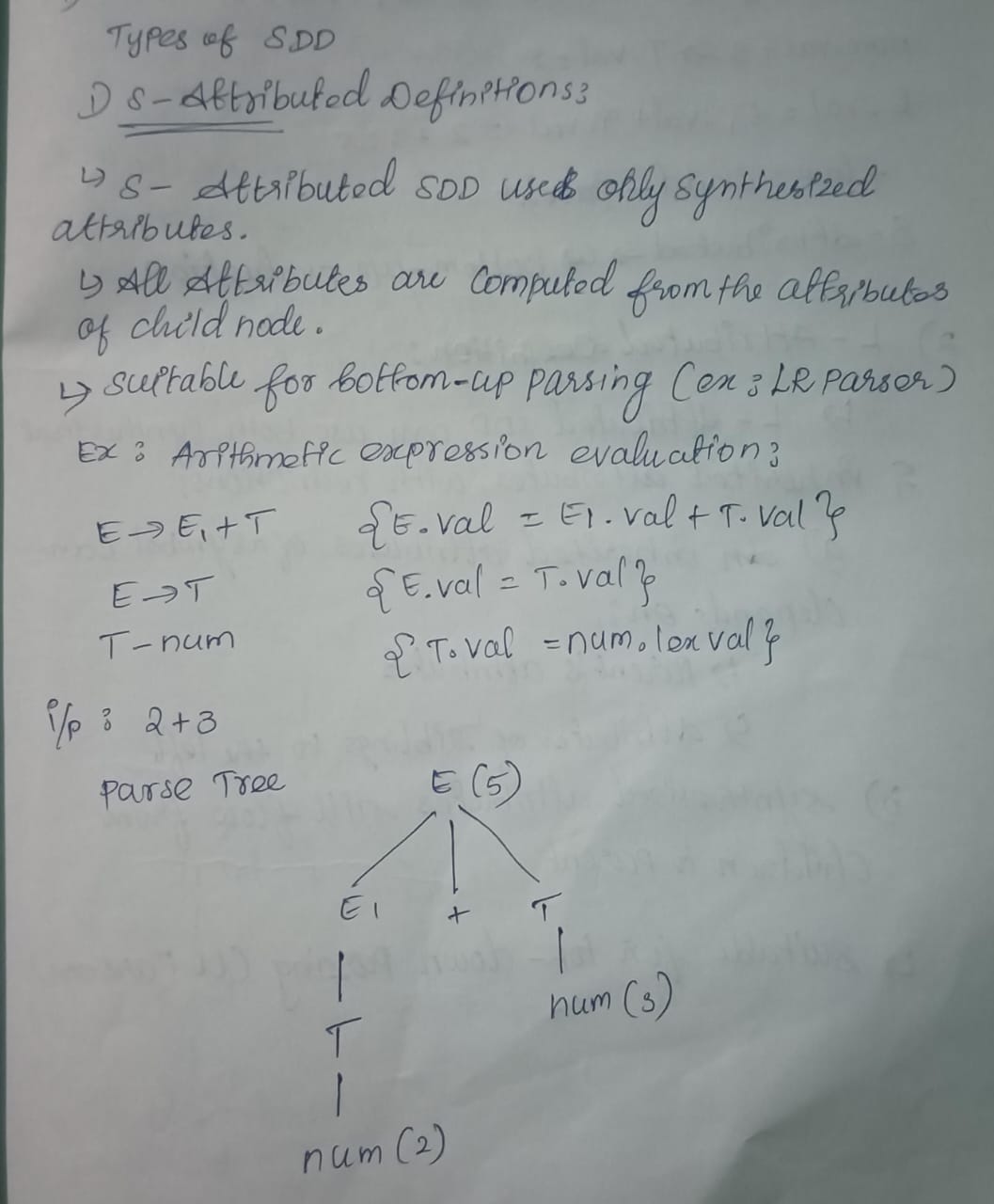
****

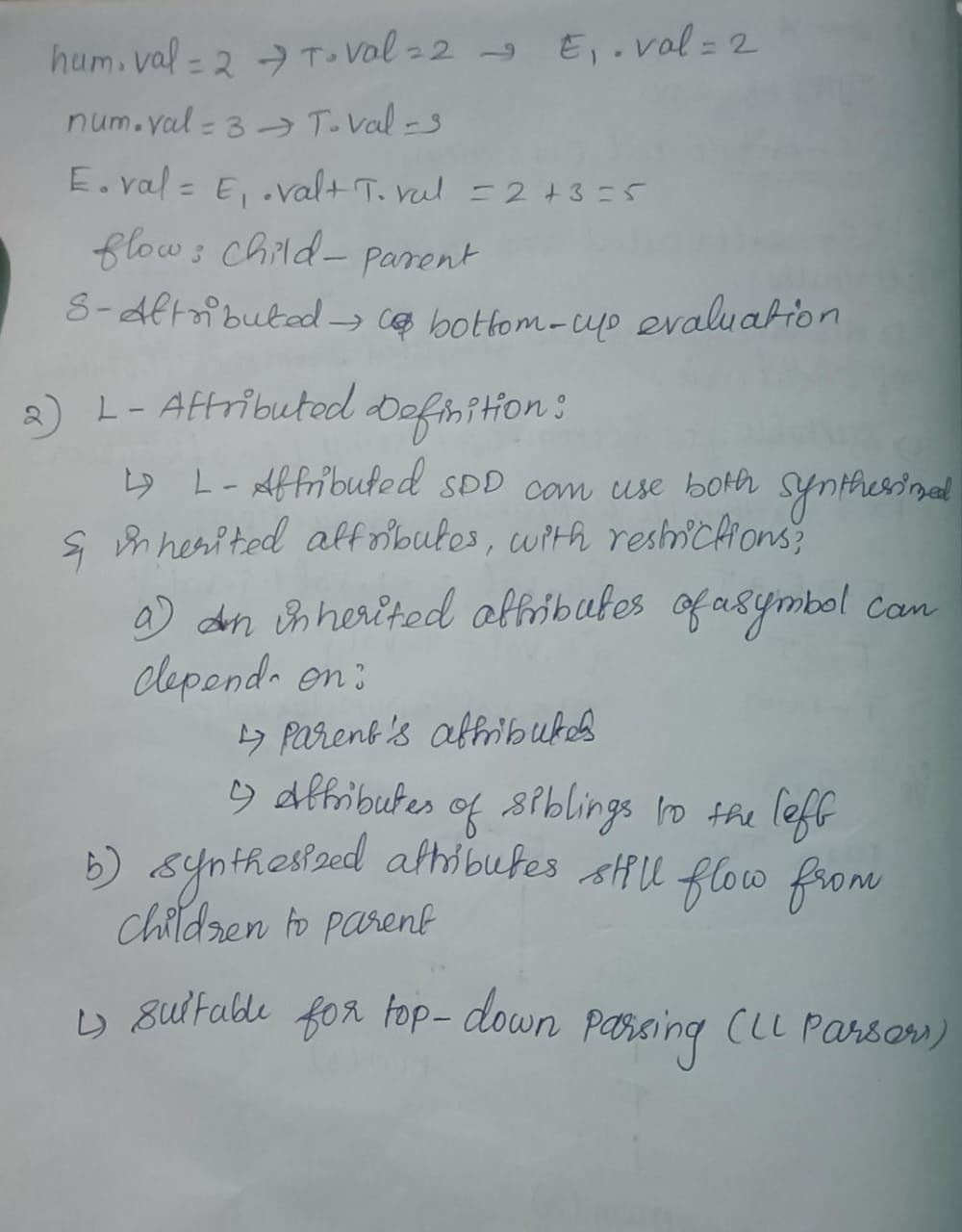
1. **Elucidate the concept of Syntax Directed Definitions (SDD) and the methodology for constructing a syntax tree for arithmetic expressions. Delineate the mechanism for bottom-up evaluation of S-attributes, illustrating the process with a representative expression to demonstrate attribute computation and dependency resolution.**

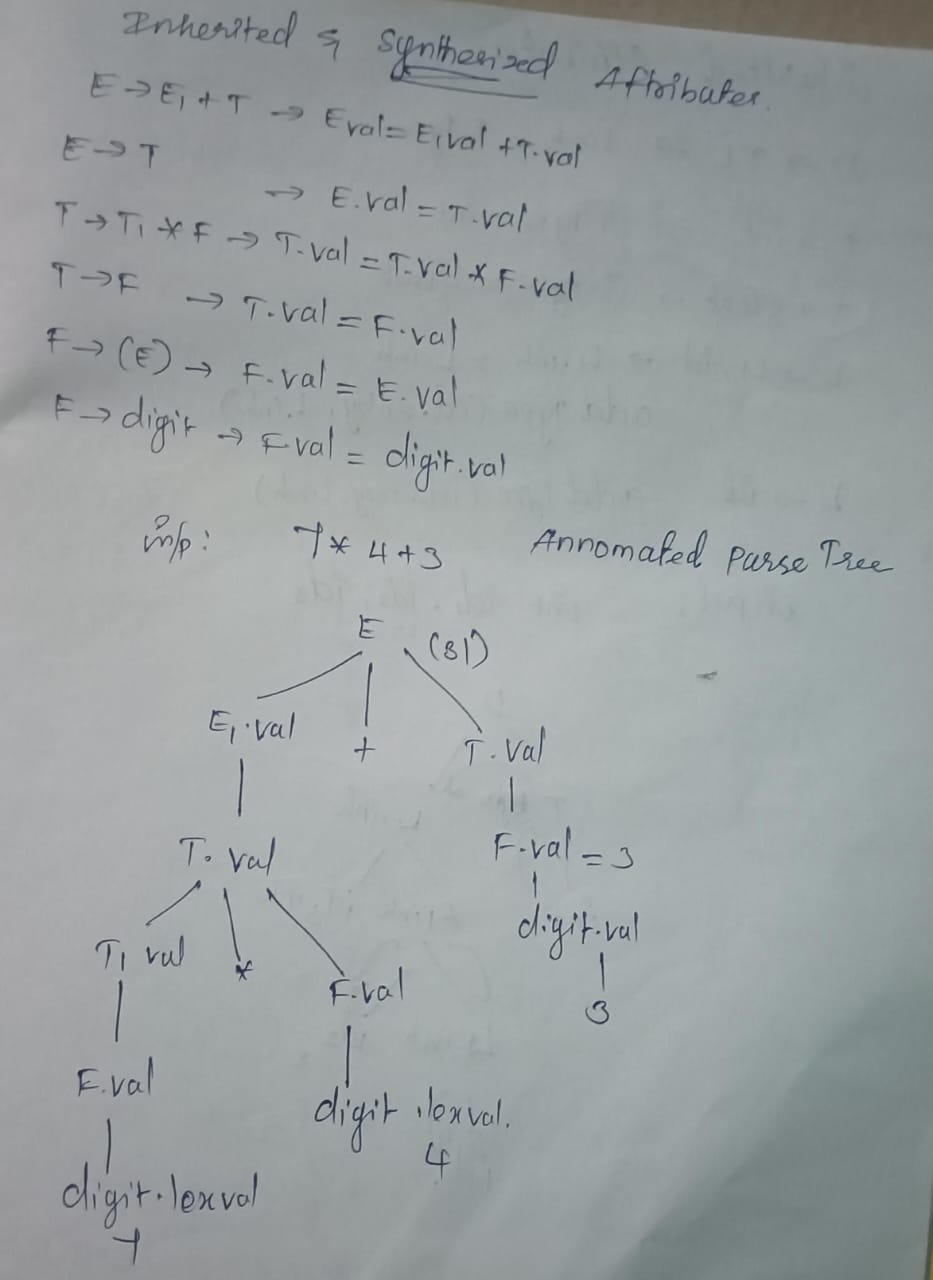
****

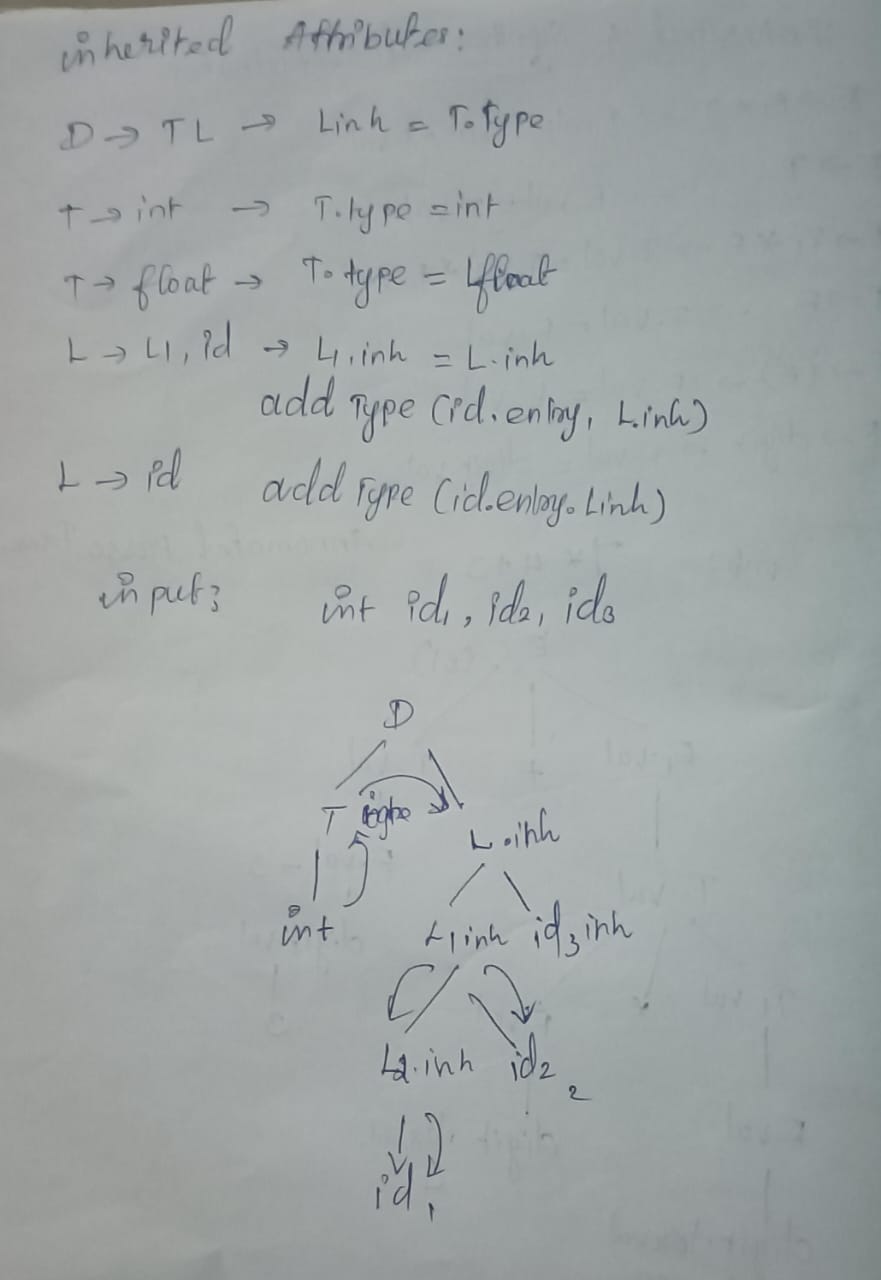
****

****

****

****

****

****