3.2 ABSTRACT SYNTAX TREES FOR EXPRESSIONS

An **Abstract Syntax Tree (AST)** is a condensed, hierarchical representation of the logical structure of source code, primarily used as an intermediate representation in compiler design. For expressions, an AST abstracts away superficial syntax details like parentheses, commas, and punctuation, preserving only the operator precedence and operands needed for evaluation.

Core Anatomy of an Expression AST

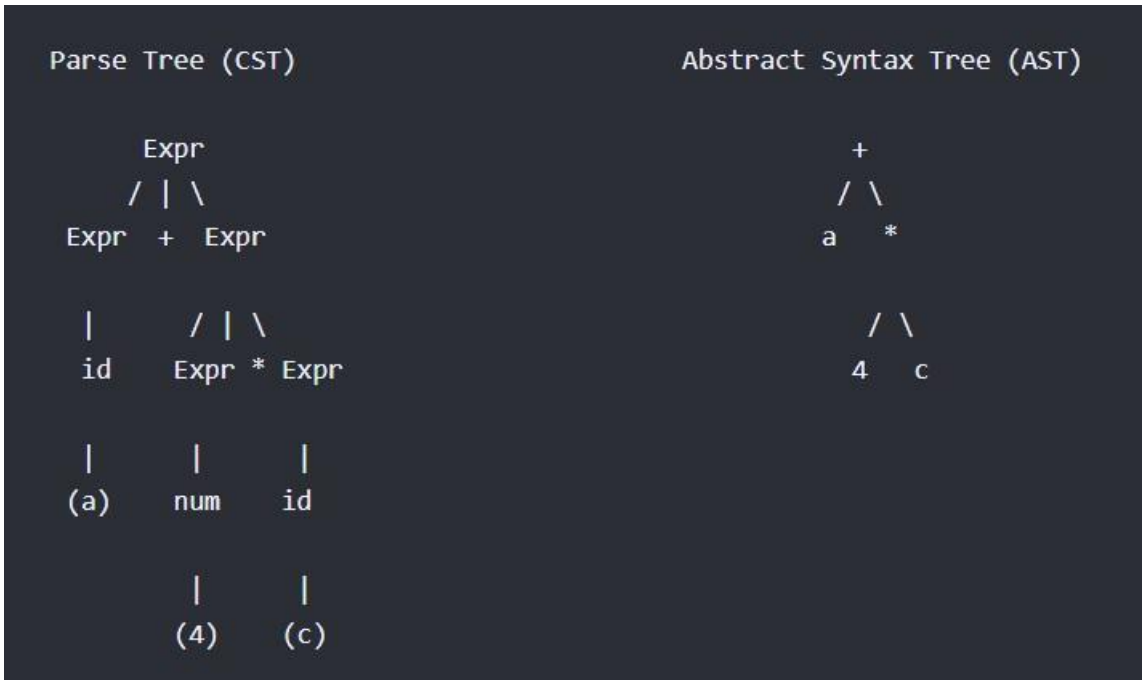
In an expression AST, nodes are mapped based on their grammatical function:

[1, 2] **Interior Nodes**: Represent **operators** (e.g., +, -, *, /).

Leaf Nodes: Represent **operands** such as constants, numeric literals, or variable identifiers. **3.2.1 Step-by-Step Construction Example** Consider the arithmetic expression:

a + 4 * c

1. Parse Tree (Concrete Syntax) vs. AST (Abstract Syntax)A Parse Tree tracks every grammar rule production step explicitly. An AST collapses those chains to focus purely on execution order.



2. **Construction Functions**

Compilers use a set of primitive constructors during the syntax analysis phase to generate these nodes:

mkleaf(id, entry): Creates an identifier leaf node pointing to its symbol table index.

mkleaf(num, val): Creates a literal number leaf node containing the value.

mknode(op, left, right): Allocates an internal operator node linking to left and right sub-trees **3. Semantic Execution Steps**

Following a bottom-up parser traversal, the tree is constructed via these steps:

1. p1 = mkleaf(id, entry_a) → Creates node for a
2. p2 = mkleaf(num, 4) → Creates node for 4
3. p3 = mkleaf(id, entry_c) → Creates node for c
4. p4 = mknode('*', p2, p3) → Links 4 and c under *
5. p5 = mknode('+', p1, p4) → Links a and the result of the * subtree under

Key Differences: Parse Tree vs. AST

Feature	Parse Tree (Concrete Syntax Tree)	Abstract Syntax Tree (AST)
Granularity	Includes every syntactic detail (brackets, semicolons, keywords).	Contains only the structural logic of the operation.
Node Types	Non-terminals serve as internal nodes.	Operators serve as internal nodes.
Memory footprint	High; contains redundant chains of production rules.	Low; highly optimized and compact.
Compiler Phase	Generated during parsing / syntactic analysis.	Used in semantic analysis, type-checking, and optimization.

3.2.2 Implementation in Compiler Code

In an object-oriented compiler frontend, an AST for an expression language is typically implemented using class inheritance hierarchies:

Java Code:

```
// Base class for all expression
nodes abstract class Expr {}


// Leaf node for
numeric constants
class NumLiteral
extends Expr { int
value;
```

```
    NumLiteral(int val) { this.value = val; }
}

// Leaf node for
variables class
Identifier extends
Expr {
    String name;
    Identifier(String name) { this.name = name; }
}
// Interior node for
binary operators class
BinaryOp extends
Expr { char operator;
Expr left;
    Expr right;
    BinaryOp(char op, Expr
        lhs, Expr rhs) {
        this.operator = op;
        this.left = lhs; this.right =
        rhs;
    }
}
```

3.2.3 Downstream Uses of Expression ASTs

Once built, the compiler relies on the expression AST for three major procedures:

1. **Type Checking**: The compiler traverses the tree to ensure operators are receiving compatible types (e.g., verifying you aren't adding a string to an int).
2. **Optimization**: Techniques like constant folding evaluate static expressions (e.g., simplifying 3 + 5 directly to 8) at compile-time by modifying the tree.
3. **Code Generation**: The tree structure directly mirrors stack machine operations or linearized intermediate code formats like three-address code.