PRESENTATION ON

DESIGN AND IMPLEMENTATION OF A SYNTAX –DIRECTED TRANSLATOR FOR ARITHMETIC EXPRESSIONS

Submitted to the department of

COMPUTER SCIENCE ENGINEERING

BACHELOR OF TECHNOLOGY

3RD YEAR

Submitted by:

Himanshu Negi (09318002722)
Rishi kumar Jha (09818002722)
Sudhanshu (11518002722)

Under the Guidance of

Design and Implementation of a Syntax-Directed Translator for Arithmetic Expressions

Syntax-directed translators are crucial tools in compiler design. They transform arithmetic expressions into executable code. This presentation explores their design and implementation, focusing on key components and techniques.

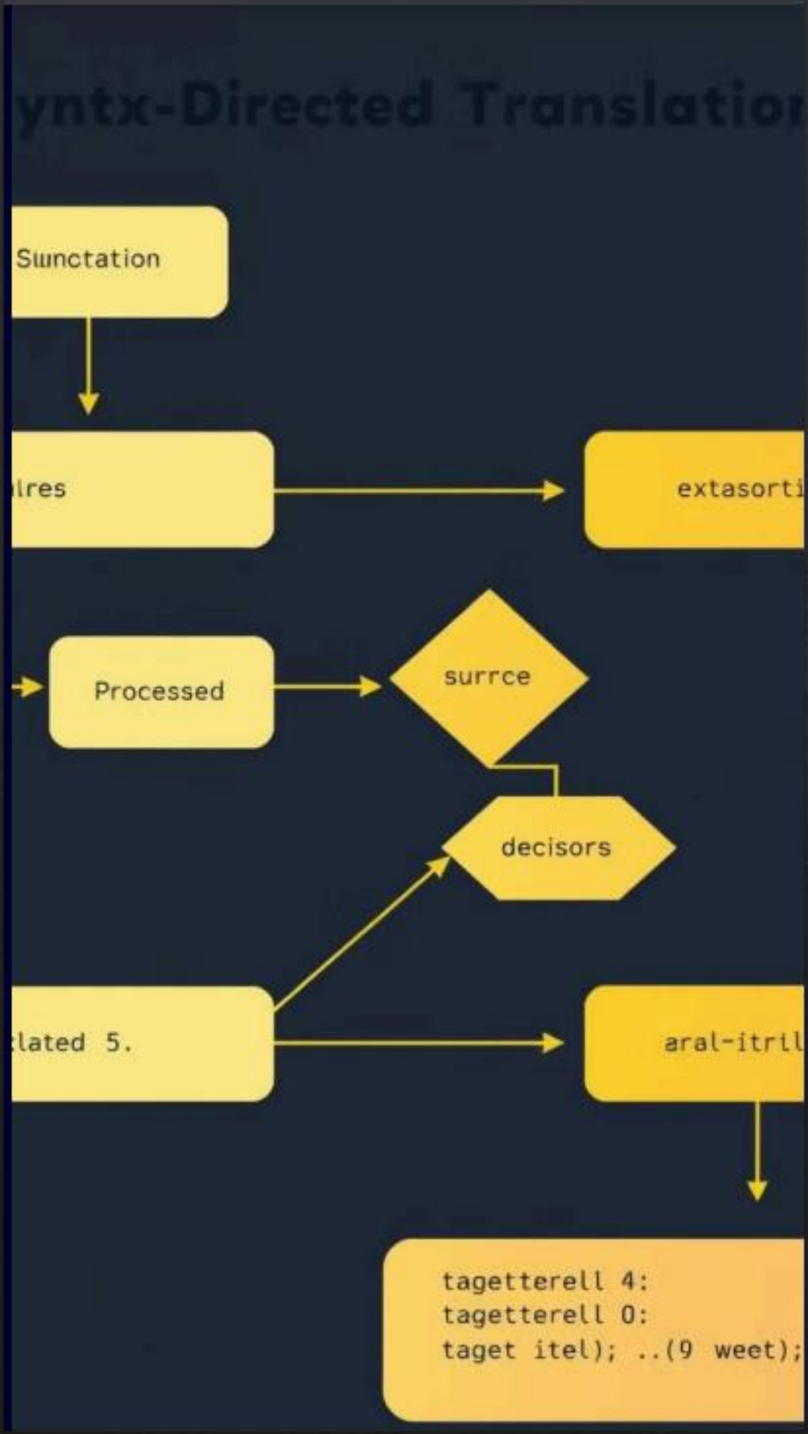
Overview of Syntax-Directed Translation

1. **Lexical Analysis**
   Tokens are generated from the input stream. This phase identifies lexemes and categorizes them

2. **Syntax Analysis**
   The parser constructs a parse tree. It ensures the expression follows grammar rules.

3. **Semantic Analysis**
   Meaning is attached to the parsed structure. Type checking and other semantic rules are applied.

4. **Code Generation**
   The final phase produces target code. It translates the analyzed expression into executable instructions.



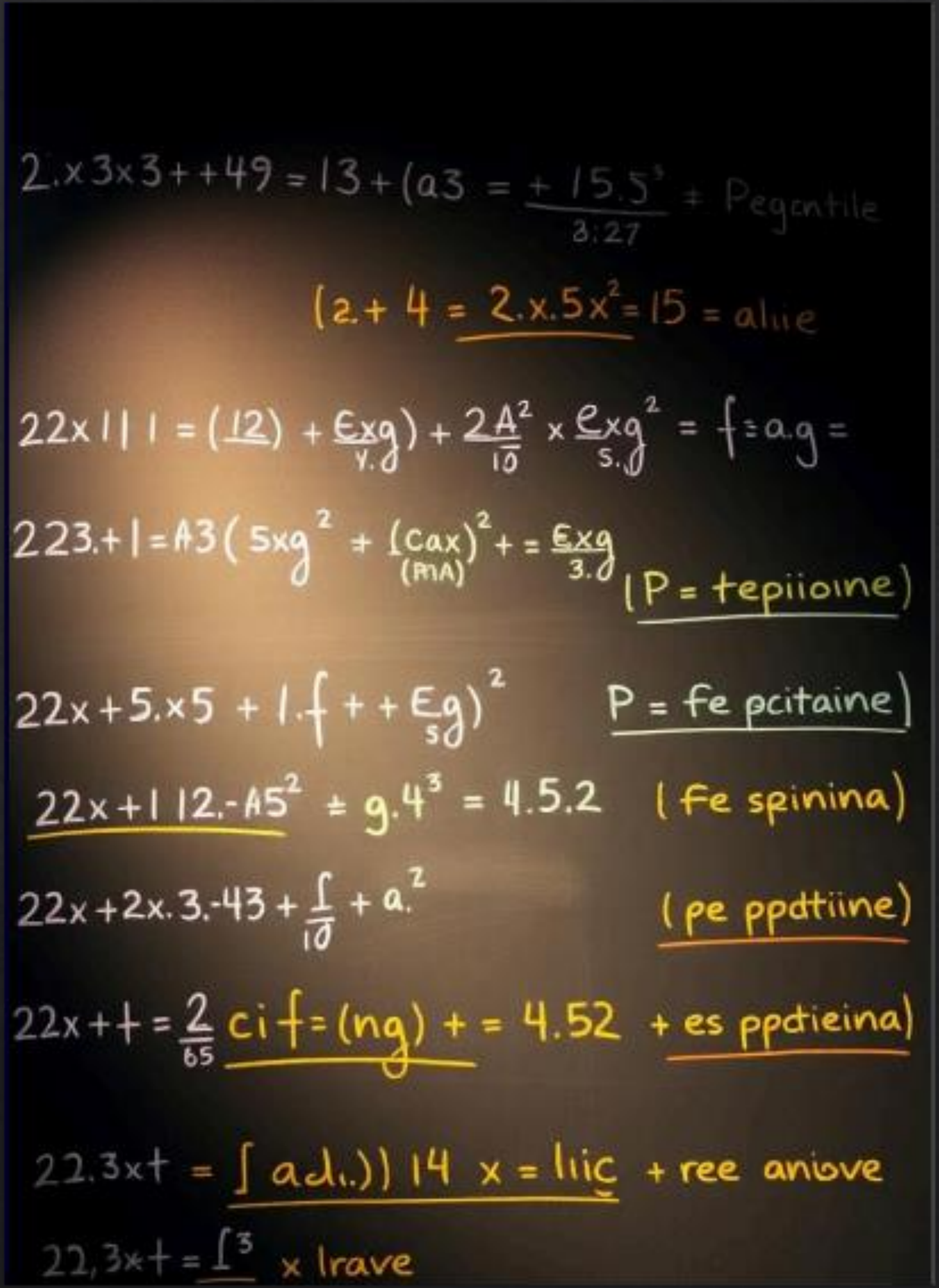
Arithmetic Expression Grammar

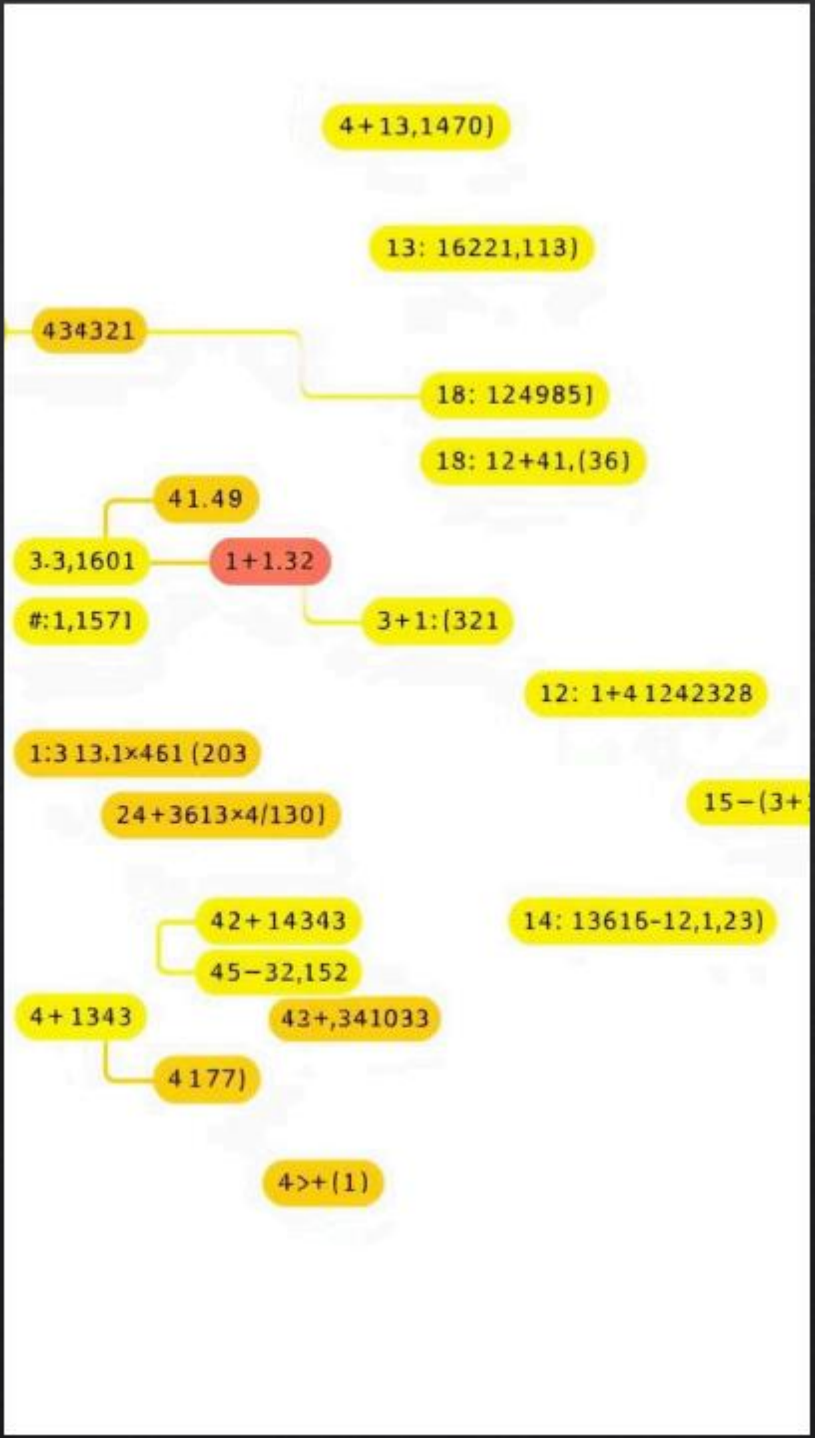
Terminal Symbols:
Includes numbers, variables, and operators. These are the basic building blocks of expressions.

Non-Terminal Symbols:
Represent higher-level constructs. Examples include 'expression', 'term', and 'factor'

Production Rules :
Define the structure of valid expressions. They specify how symbols can be combined

Precedence:
Grammar rules encode operator precedence. This ensures correct evaluation order of complex expressions.





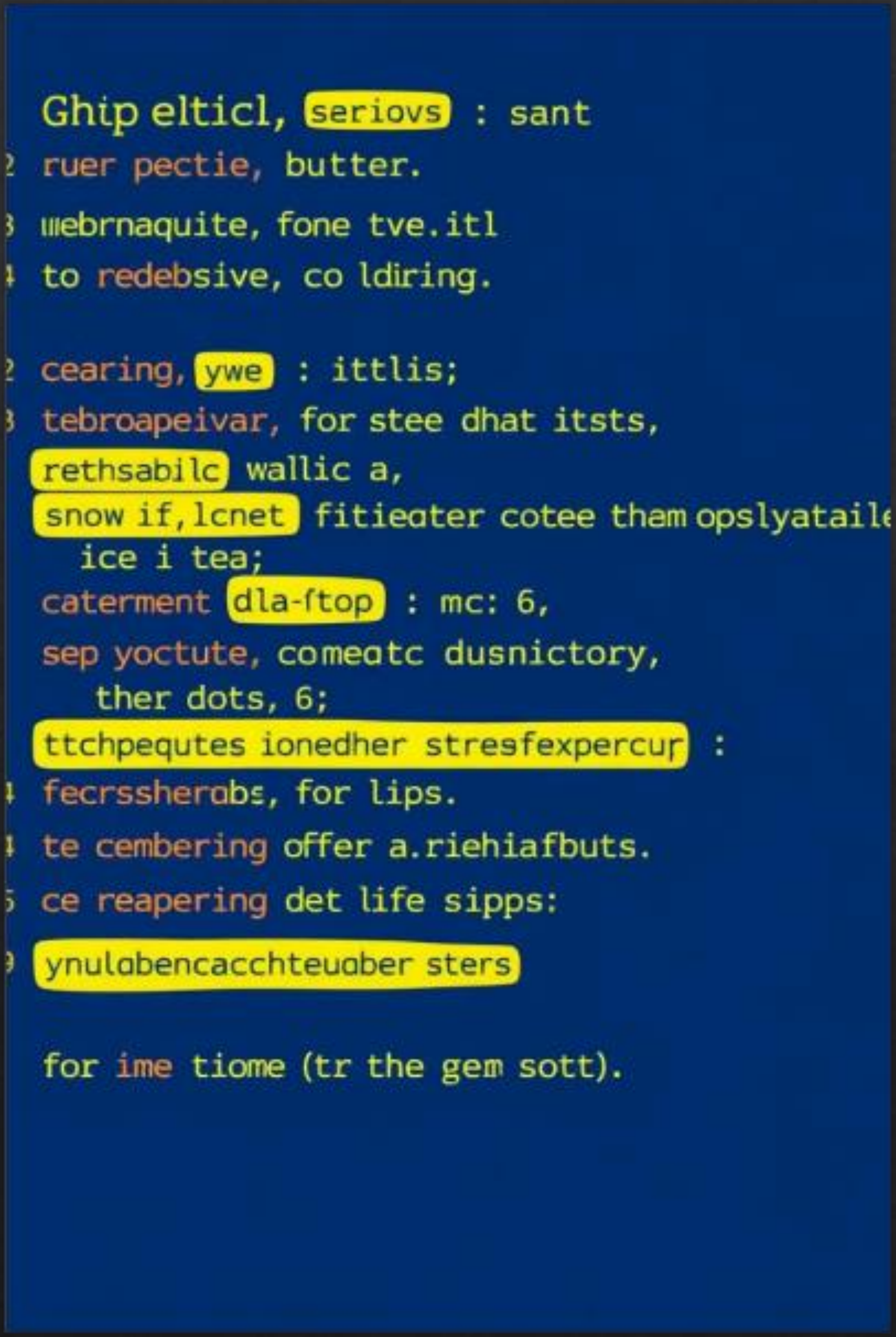
Parse Tree Representation

1. **Root Node**
   Represents the entire expression. It's the starting point for tree traversal.

2. **Internal Nodes**
   Correspond to non-terminal symbols. They represent subexpressions or operations.

3. **Leaf Nodes**
   Represent terminal symbols. These are typically numbers or variables in the expression.

4. **Edge Relationships**
   Show the hierarchical structure. They indicate how subexpressions are combined to form larger expressions.



Semantic Actions for Expression Evaluation

1

Attribute Synthesis

Information flows up the parse tree. Child nodes contribute to parent node attributes.

2

Value Calculation

Each node computes its value. This is based on operator semantics and child values.

3

Type Inference

Node types are determined. This ensures type consistency across the expression

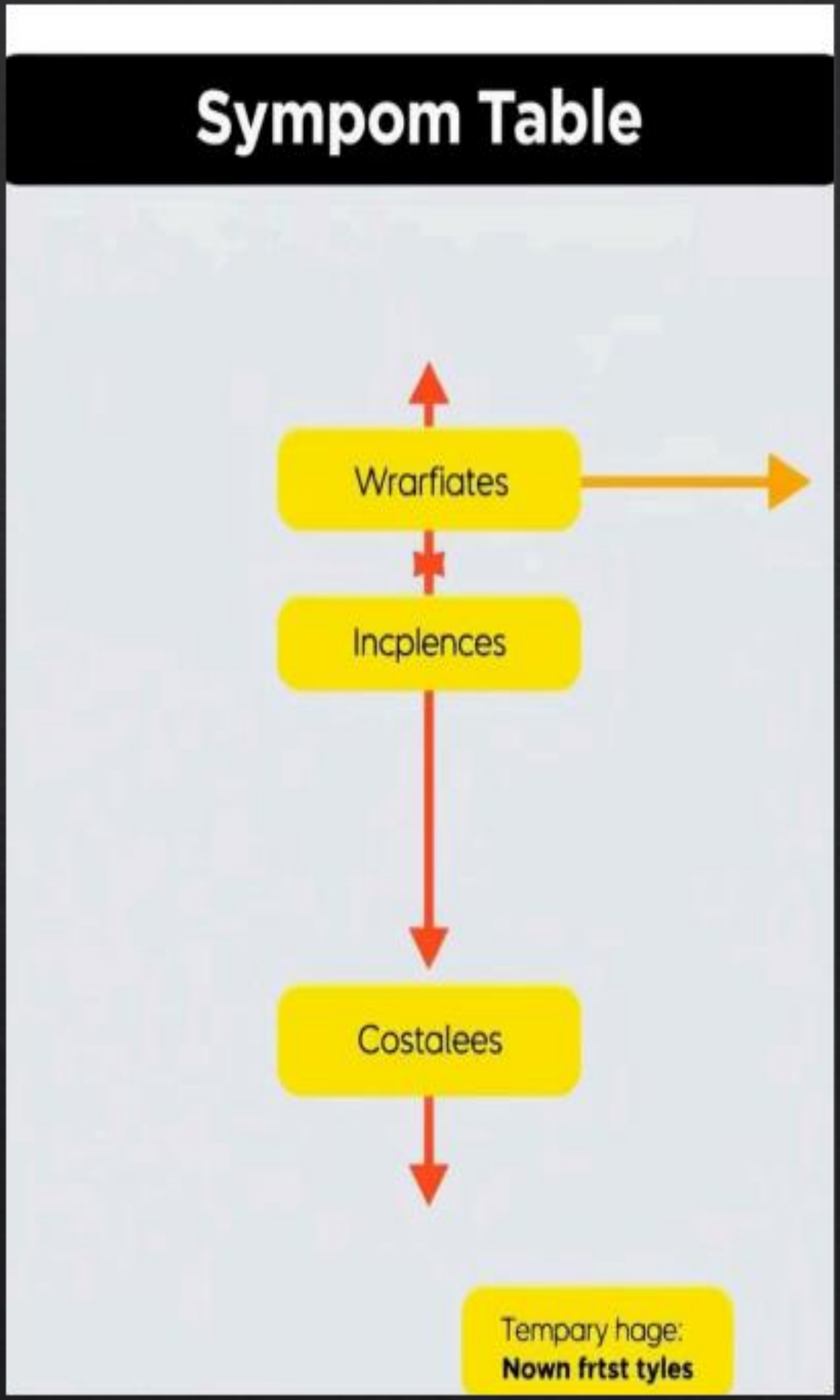
4

Error Handling

Semantic actions detect and report errors. This includes type mismatches or undefined variables.

Implementation of Symbol Table Management

Component	Purpose	Implementation
Hash Table	Fast symbol lookup	Array of linked lists
Scope Management	Handle nested scopes	Stack of symbol tables
Symbol Attributes	Store variable info	Struct or object
Collision Resolution	Handle hash conflicts	Chaining or probing



Type Checking and Implicit Type Conversions

Static Type Checking

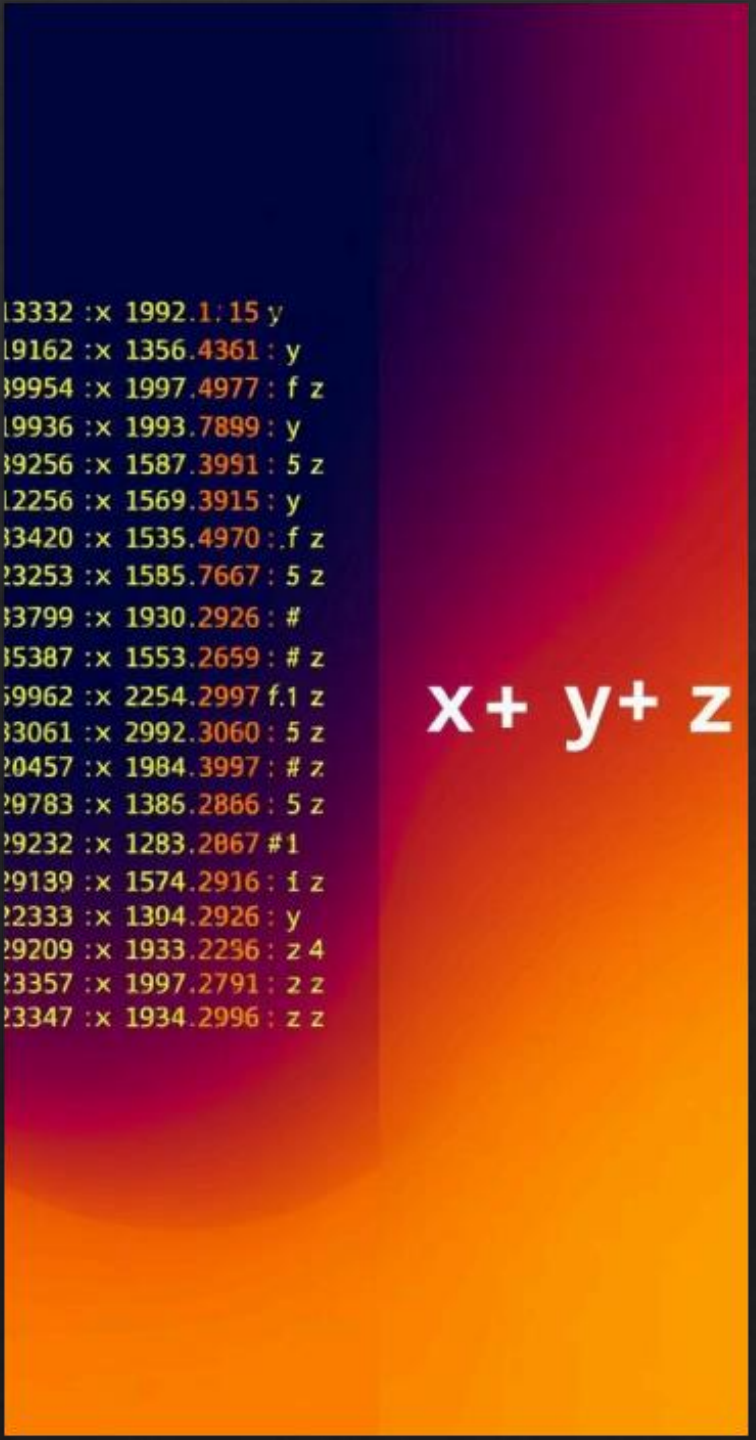
Verifies type compatibility at compiletime. Catches type errors before execution.

Dynamic Type Checking

Performs type checks at runtime. Allows for more flexible but potentially less safe code.

Implicit Conversions

Automatically converts types when safe. Examples include int to float promotion in mixed expressions.



Code Generation for Arithmetic Expressions

1. **Instruction Selection**
   Choose appropriate machine instructions. Map high-level operations to low-level code.

2. **Register Allocation**
   Assign variables to CPU registers. Optimize for efficient use of limited register space

3. **Memory Management**
   Handle stack and heap allocation. Ensure proper memory usage for variables and temporaries.

4. **Optimization**
   Apply code optimizations. Improve efficiency while preserving semantics.

Handling Operator Precedence and Associativity

Precedence Levels:

Assign priority to operators. Higher precedence operators are evaluated first.

Left Associativity

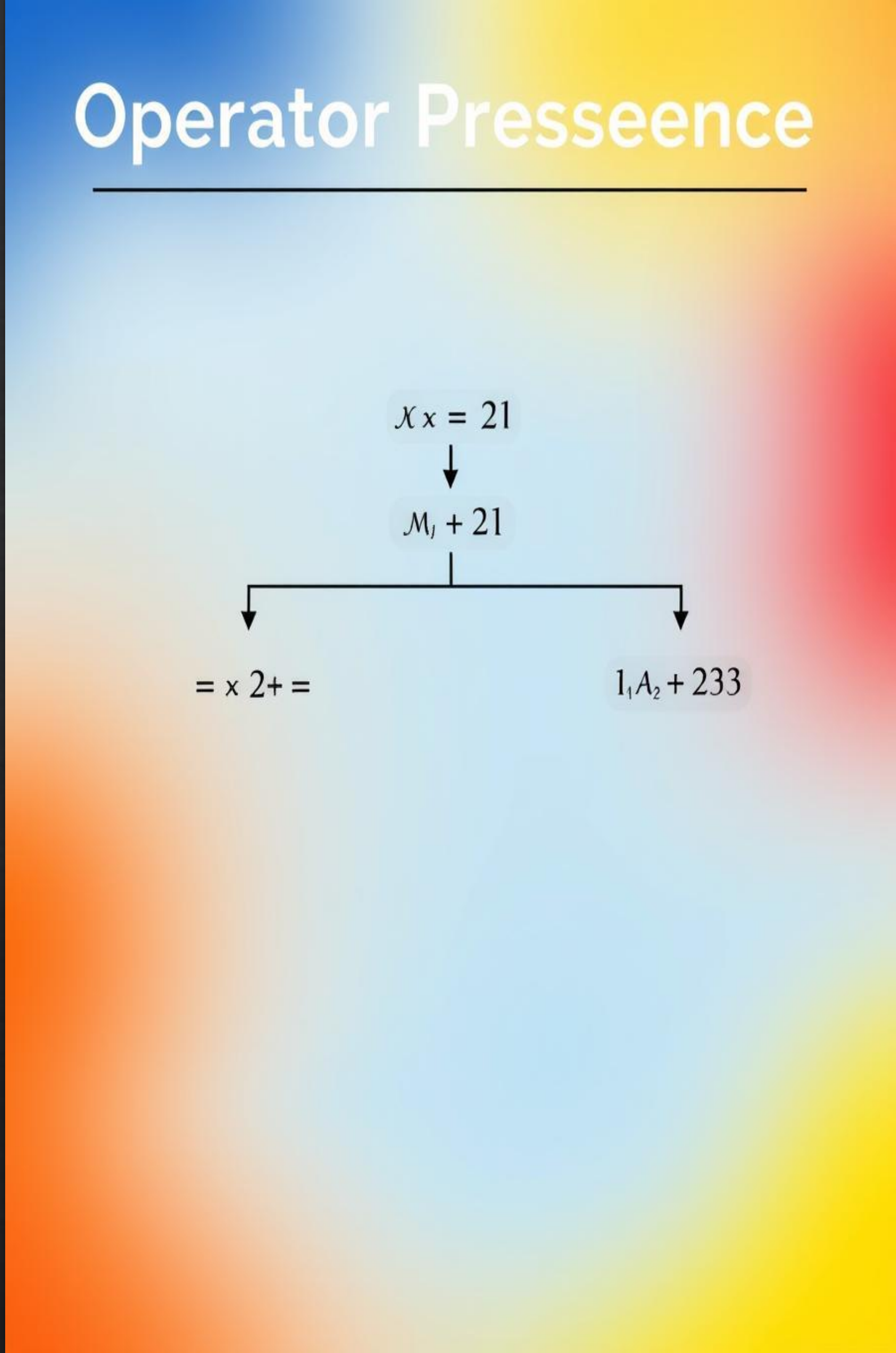
Most binary operators associate left-to-right. Example: a - b - c is (a - b) - c.

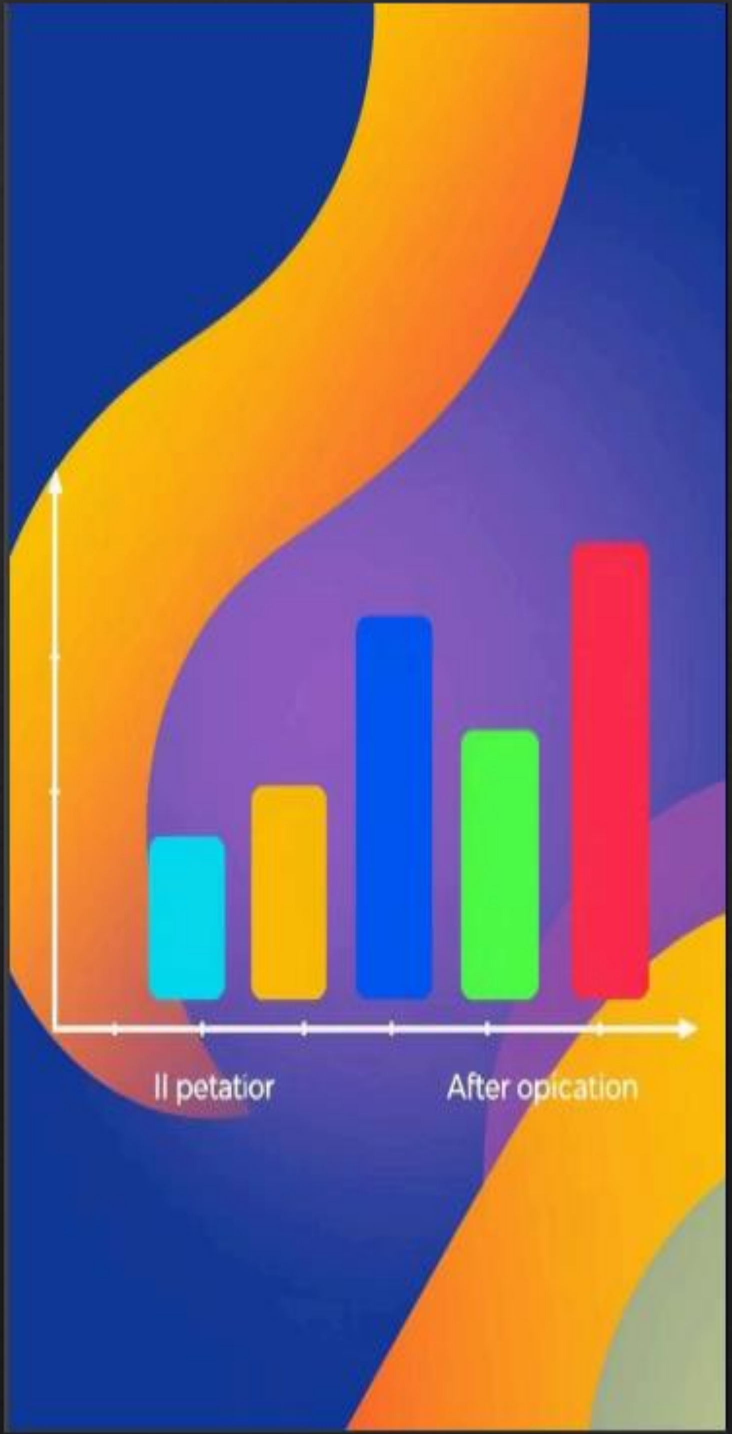
Right Associativity

Some operators, like exponentiation, associate right-toleft. Example: a^b^c is a^(b^c).

Parse Tree Structure

Reflect precedence in tree shape. Higher precedence operations are deeper in the tree.





Optimizations and Improvements

Constant Folding

Evaluate constant expressions at compile-time. Reduces runtime computations.

Common Subexpression Elimination

Identify and reuse repeated subexpressions. Avoids redundant calculations.

Dead Code Elimination

Remove code that doesn't affect the output. Improves code size and execution speed.

Strength Reduction

Replace expensive operations with simpler ones. Example: multiply by 2 becomes left shift

THANK YOU