**University of Asia Pacific**

**Department of Computer Science & Engineering**

**CSE 430: Compiler Design**

**Lab Problem 4: Three Address Code Generation**

**Three Address Code:**

**Three-address code** (often abbreviated to TAC or 3AC) is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations. Three address code is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in a temporary variable generated by the compiler. The compiler decides the order of operation given by three address code.

**General representation:**

**a = b op c**

Where a, b, or c represents operands like names, constants, or compiler-generated temporaries and op represents the operator.

**Example:**

x = (-b + sqrt(b^2 - 4\*a\*c)) / (2\*a)

**TAC:**

t1 := b \* b

t2 := 4 \* a

t3 := t2 \* c

t4 := t1 - t3

t5 := sqrt(t4)

t6 := 0 - b

t7 := t5 + t6

t8 := 2 \* a

t9 := t7 / t8

x := t9

**Implementation Details:**

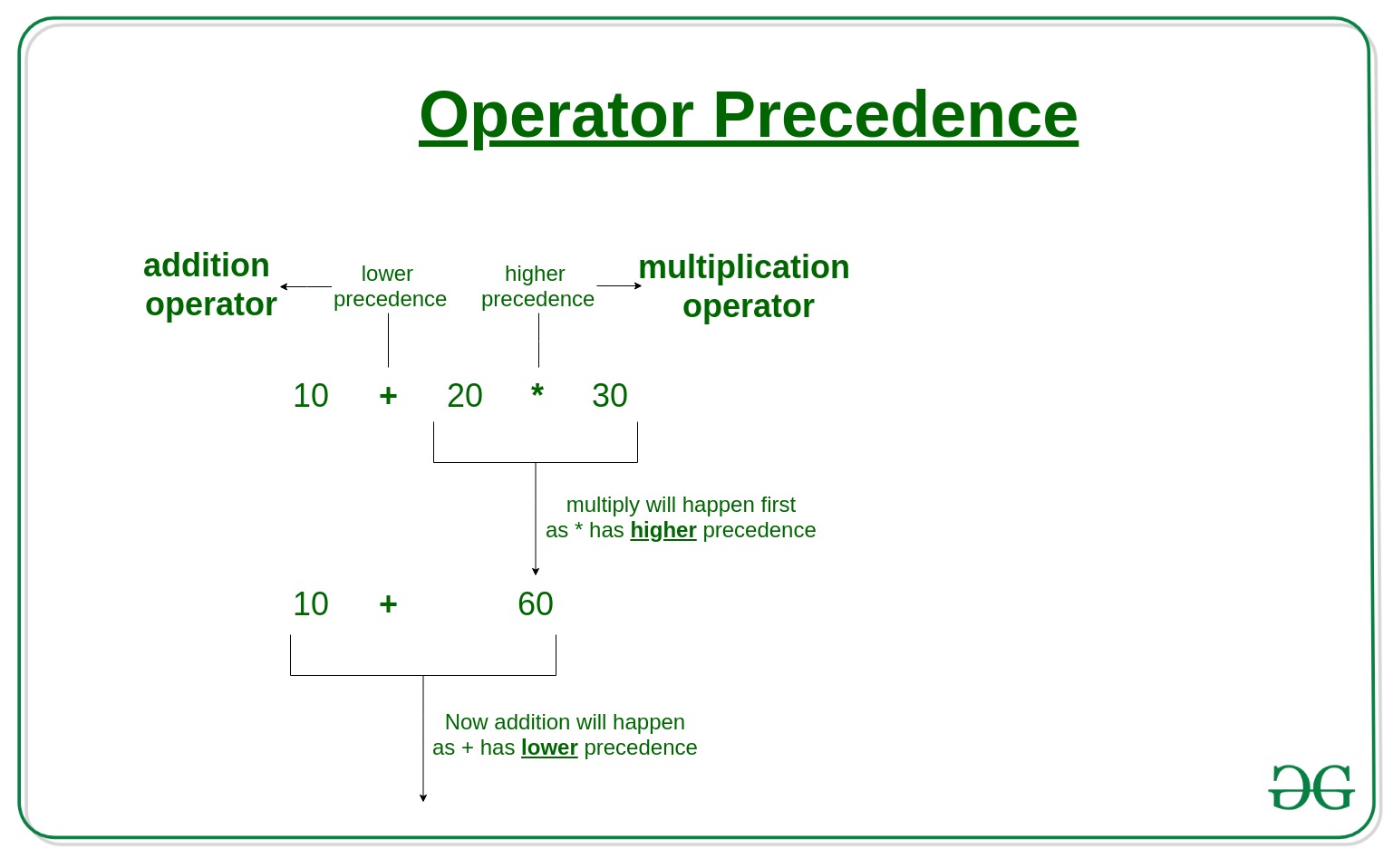
**i) Precedence:**

While implementing the TAC, the first thing to keep in mind is precedence. **Precedence**/ **order of operations** is a collection of rules that reflect conventions about which procedures to perform first in order to evaluate a given mathematical expression.

The precedence of an operator specifies how "tightly" it binds two expressions together. For example, in the expression 1 + 5 \* 3, the answer is 16 and not 18 because the multiplication ("\*") operator has higher precedence than the addition ("+") operator. Parentheses may be used to force precedence, if necessary. For instance: (1 + 5) \* 3 evaluates to 18.

**Example:**

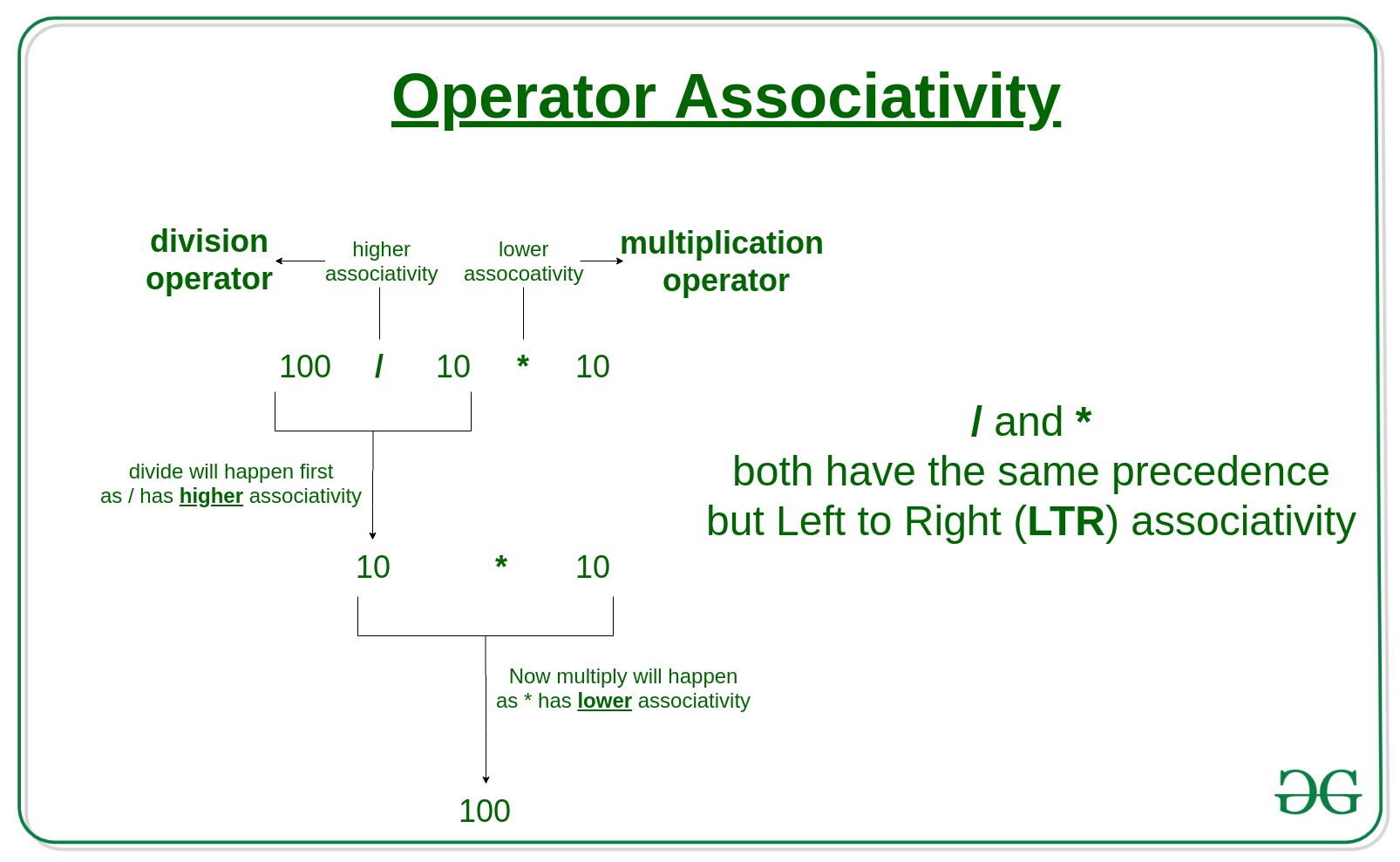
**Solve: 10 + 20 \* 30**

****

**ii) Left associativity:**

Operators Associativity is used when two operators of the same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.

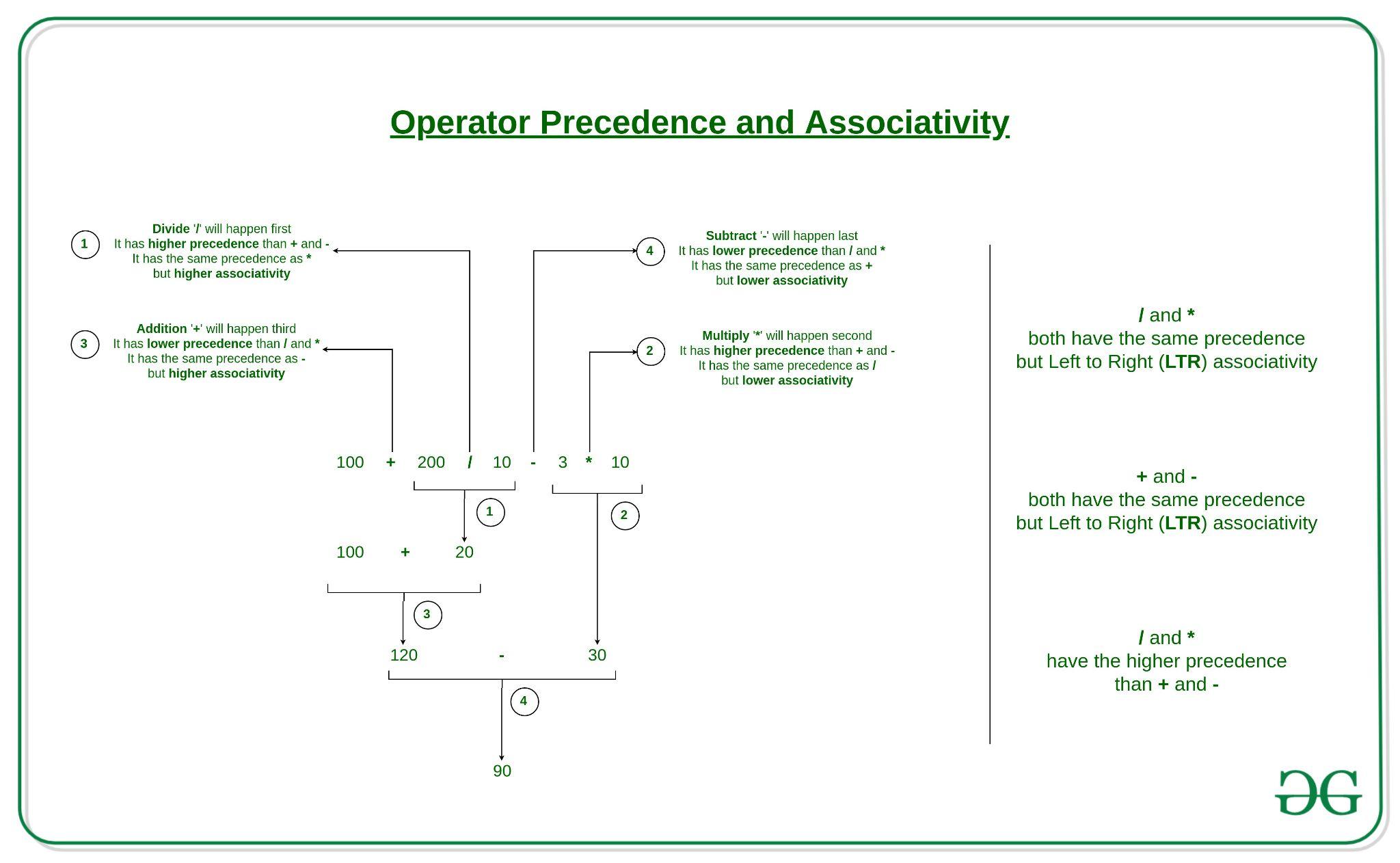
For example: ‘\*’ and ‘/’ have the same precedence and their associativity is Left to Right, so the expression “100 / 10 \* 10” is treated as “(100 / 10) \* 10”.



Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets.

**Example:**

**100 + 200 / 10 - 3 \* 10**

****

**Things to Remember:**

1) Associativity is only used when there are two or more operators of the same precedence.

2) All operators with the same precedence have the same associativity.

3) Precedence and associativity of postfix ++ and prefix ++ are different.

| **Precedence** | **Operator** | **Description** | **Associativity** |
| --- | --- | --- | --- |
| **1** | ++ -- | Suffix/postfix increment and decrement | Left-to-right |
| () | Function call |
| [] | Array subscripting |
| . | Structure and union member access |
| -> | Structure and union member access through pointer |
| (*type*){*list*} | Compound literal |
| **2** | ++ -- | Prefix increment and decrement[[note 1]](https://en.cppreference.com/w/c/language/operator_precedence#cite_note-1) | Right-to-left |
| + - | Unary plus and minus |
| ! ~ | Logical NOT and bitwise NOT |
| (*type*) | Cast |
| \* | Indirection (dereference) |
| & | Address-of |
| sizeof | Size-of[[note 2]](https://en.cppreference.com/w/c/language/operator_precedence#cite_note-2) |
| \_Alignof | Alignment requirement(C11) |
| **3** | \* / % | Multiplication, division, and remainder | Left-to-right |
| **4** | + - | Addition and subtraction |
| **5** | << >> | Bitwise left shift and right shift |
| **6** | < <= | For relational operators < and ≤ respectively |
| > >= | For relational operators > and ≥ respectively |
| **7** | == != | For relational = and ≠ respectively |
| **8** | & | Bitwise AND |
| **9** | ^ | Bitwise XOR (exclusive or) |
| **10** | | | Bitwise OR (inclusive or) |
| **11** | && | Logical AND |
| **12** | || | Logical OR |
| **13** | ?: | Ternary conditional | Right-to-left |
| **14** | = | Simple assignment |
| += -= | Assignment by sum and difference |
| \*= /= %= | Assignment by-product, quotient, and remainder |
| <<= >>= | Assignment by bitwise left shift and right shift |
| &= ^= |= | Assignment by bitwise AND, XOR, and OR |
| **15** | , | Comma | Left-to-right |

It is good to know precedence and associativity rules, but the best thing is to use brackets, especially for less commonly used operators (operators other than +, -, \*.. etc). Brackets increase the readability of the code as the reader doesn’t have to see the table to find out the order.

For simplicity in this experiment, use only first brackets and arithmetic operators (^, \*,/,+,-, %, =) and some unary operators.

**Sample Input/Output:**

You should take inputs from a file/console.

**Input:**

-(a x b) + (c + d) – (a + b + c + d)

**Output:**

(1) T1 = a x b

(2) T2 = uminus T1

(3) T3 = c + d

(4) T4 = T2 + T3

(5) T5 = a + b

(6) T6 = T3 + T5

(7) T7 = T4 – T6