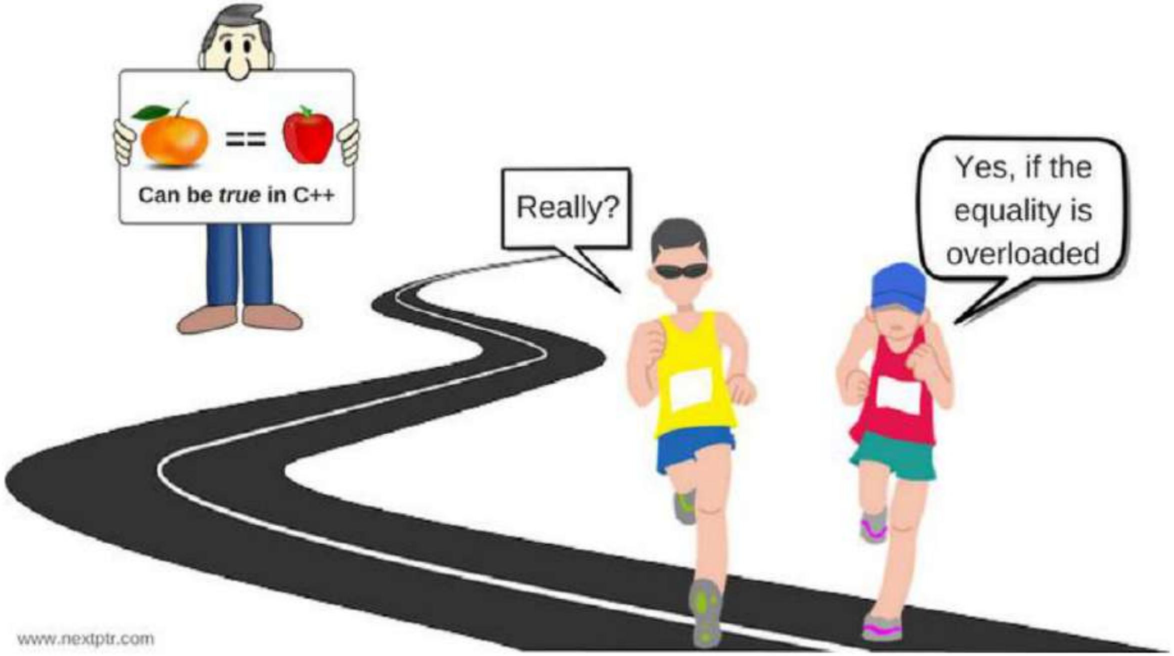
CS341 CONCEPT OF PROGRAMMING LANGUAGES

Ch7 : Expressions and Assignment

Dr. Nada Mobark



GOAL

Understand the semantics of operators, expression evaluation, type conversions, and assignment.

TOPICS

7.1 Introduction

7.2 Arithmetic Expressions

7.3 Overloaded Operators

7.4 Type conversions

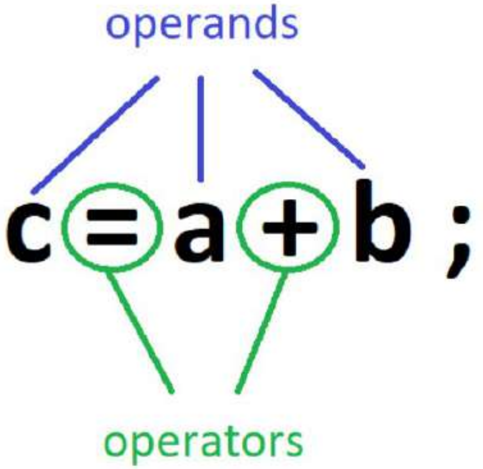
7.5 Relational and Boolean Expressions

7.6 Short-Circuit Evaluation

7.7 Assignment Statements

7.1 INTRODUCTION

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Expressions are the fundamental means of specifying computations in a programming language
- To understand expression evaluation, need to be familiar with the orders of <u>operator</u> and <u>operand</u> evaluation



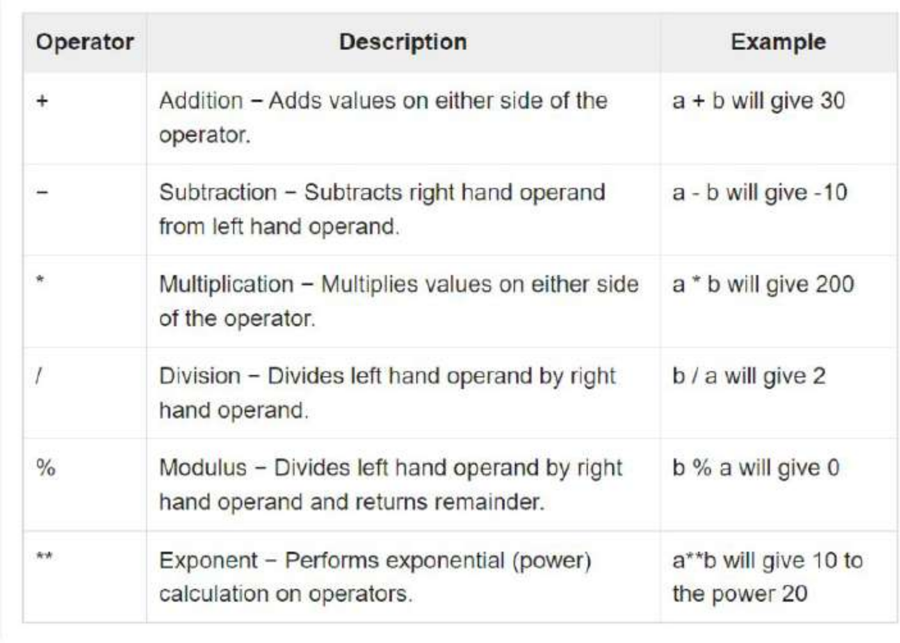
Dr. Nada Mobark, 2023

7.2 ARITHMETIC EXPRESSIONS

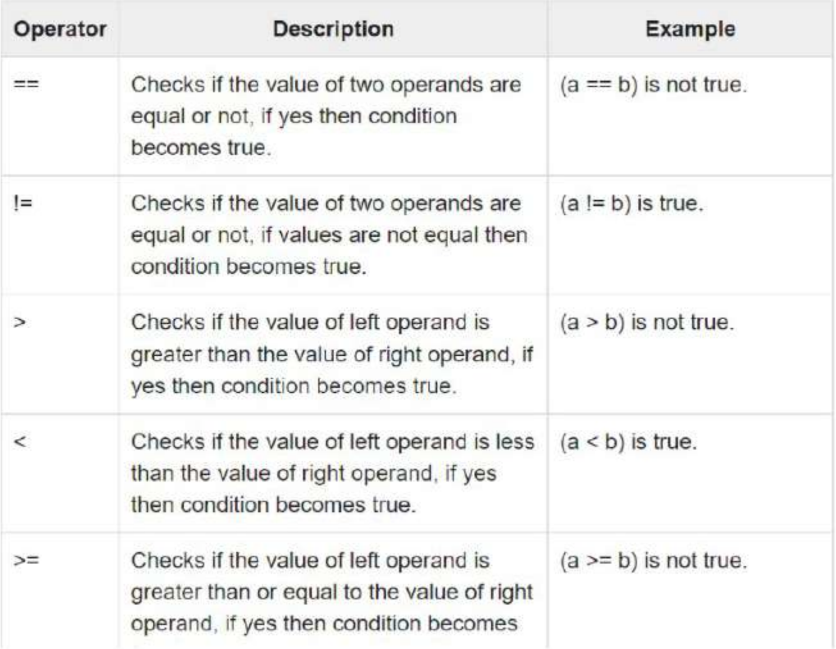
- Similar to mathematics, arithmetic expressions consist of operators, operands, parentheses, and function calls
- Design issues:
  - Types of operators?
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

OPERATORS, CATEGORIES

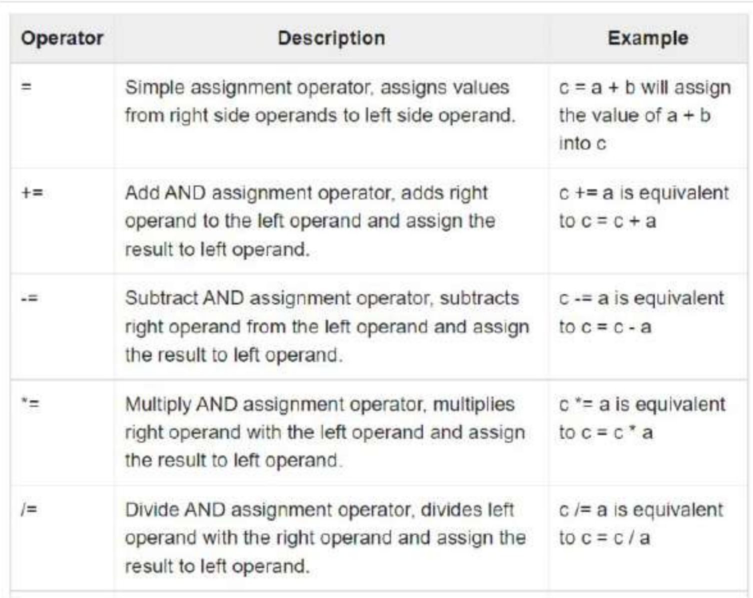
- Example, Ruby



Operator	Description	Example
+	Addition – Adds values on either side of the operator.	a + b will give 30
−	Subtraction – Subtracts right hand operand from left hand operand.	a - b will give -10
*	Multiplication – Multiplies values on either side of the operator.	a * b will give 200
/	Division – Divides left hand operand by right hand operand.	b / a will give 2
%	Modulus – Divides left hand operand by right hand operand and returns remainder.	b % a will give 0
**	Exponent – Performs exponential (power) calculation on operators.	a**b will give 10 to the power 20



Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes	(a >= b) is not true.

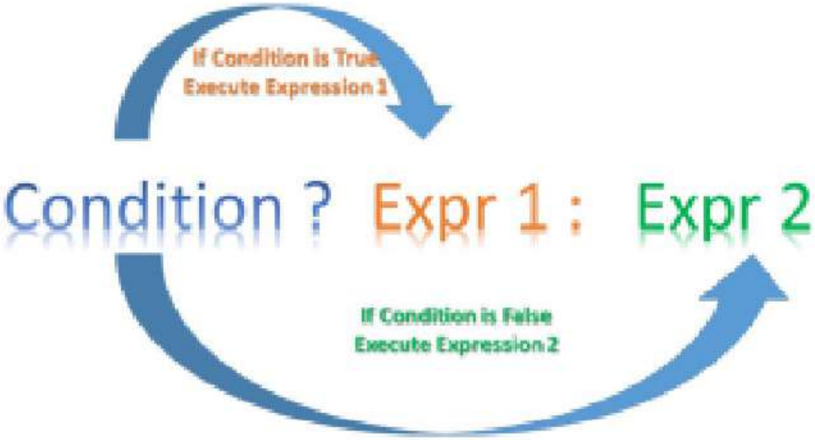


Operator	Description	Example
=	Simple assignment operator, assigns values from right side operands to left side operand.	c = a + b will assign the value of a + b into c
+=	Add AND assignment operator, adds right operand to the left operand and assign the result to left operand.	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, subtracts right operand from the left operand and assign the result to left operand.	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, multiplies right operand with the left operand and assign the result to left operand.	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, divides left operand with the right operand and assign the result to left operand.	c /= a is equivalent to c = c / a

OPERATORS, NUMBER OF OPERANDS

- A unary operator has one operand
- A binary operator has two operands
- A ternary operator has three operands

```
average = (count == 0)? 0 : sum / count
```



If Condition is True
Execute Expression 1

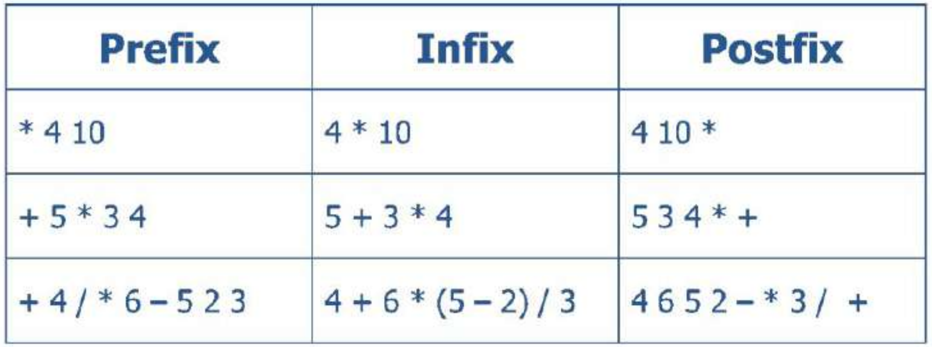
Condition ? Expr 1 : Expr 2

If Condition is False
Execute Expression 2

OPERATORS, NOTATION

- In most languages, binary operators are infix, except in Scheme and LISP, in which they are prefix; Perl also has some prefix binary operators

(Infix) a + b * c → **(prefix)** ??

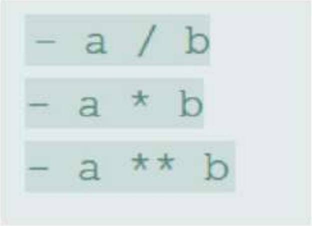


Prefix	Infix	Postfix
* 4 10	4 * 10	4 10 *
+ 5 * 3 4	5 + 3 * 4	5 3 4 * +
+ 4 / * 6 - 5 2 3	4 + 6 * (5 - 2) / 3	4 6 5 2 - * 3 / +

- Most unary operators are prefix, but the ++ and - - operators in C-based languages can be either prefix or postfix

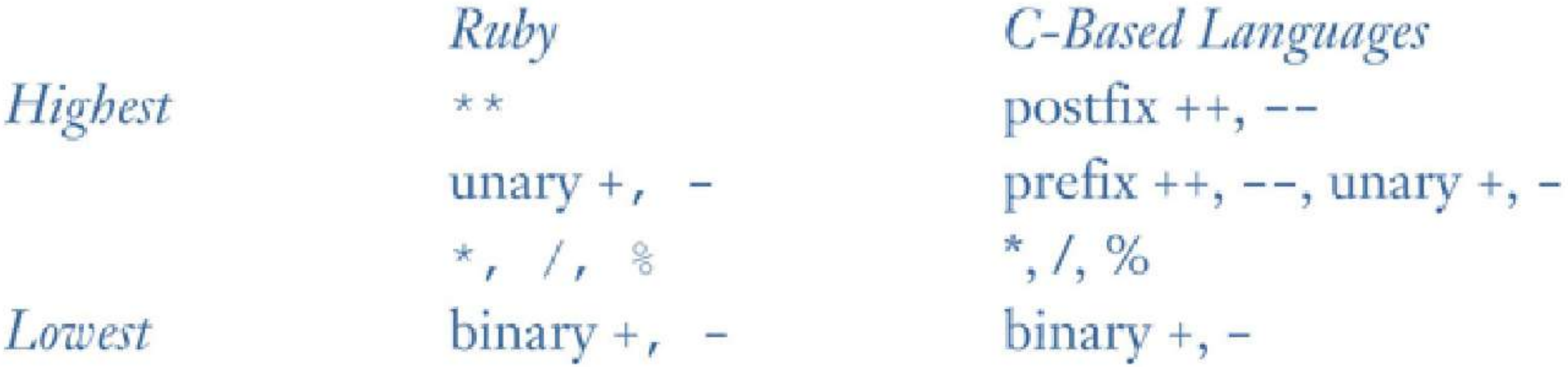
OPERATORS, PRECEDENCE

- The operator precedence rules for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated
- Example:



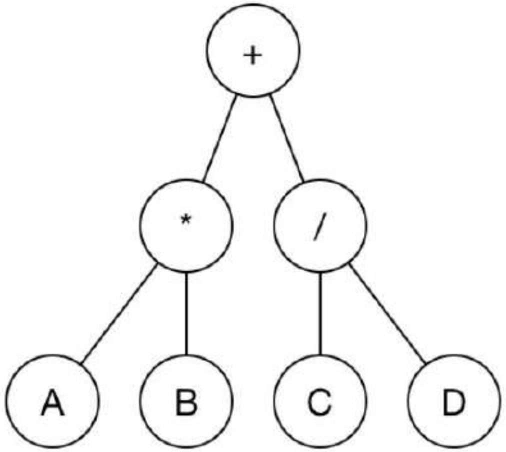
```
- a / b
- a * b
- a ** b
```

What is the relative precedence of the unary minus ??

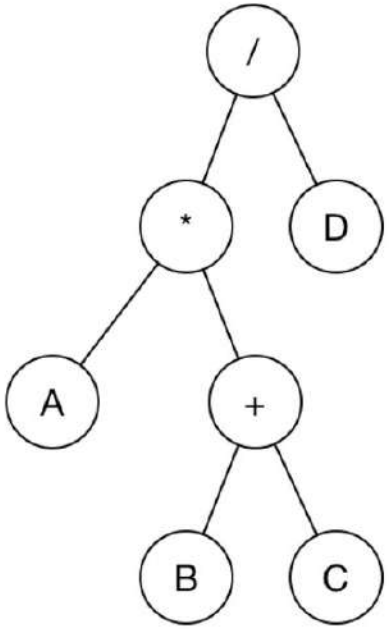


	Ruby	*C-Based Languages*
Highest	**	postfix ++, --
	unary +, -	prefix ++, --, unary +, -
	*, /, %	*, /, %
Lowest	binary +, -	binary +, -

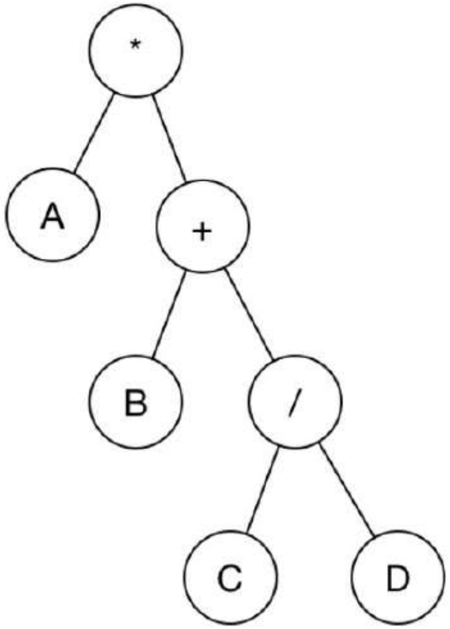
EXAMPLES



((A * B) + (C / D))



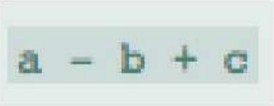
((A * (B + C)) / D)



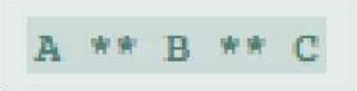
(A * (B + (C / D)))

OPERATORS, ASSOCIATIVITY

- The operator associativity rules for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left
  - In APL; all operators have equal precedence and all operators associate right to left

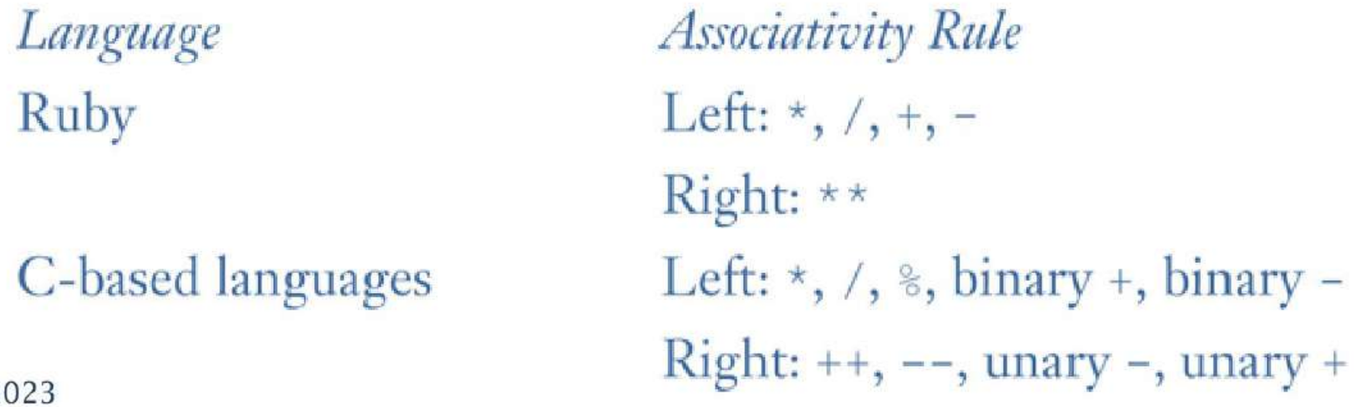


```
a - b + c
```



```
A ** B ** C
```

A + B * C



Language	*Associativity Rule*
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +

OPERATORS, ASSOCIATIVITY

- In case of floating point numbers, some associativity options may cause overflow!!
  - A & C → very large +ve values, B & D → very large –ve values

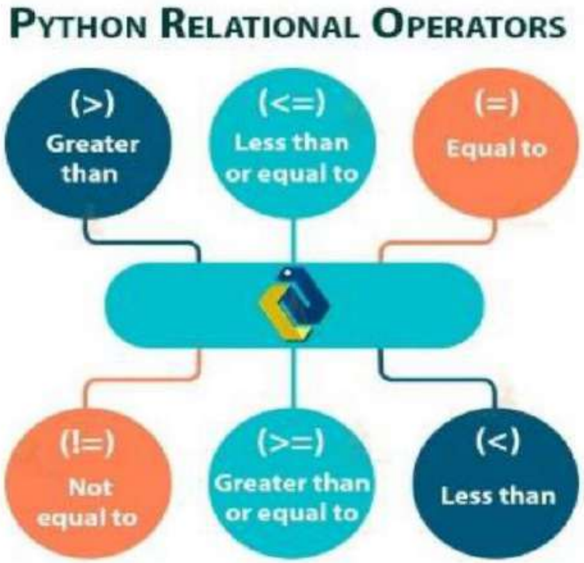
A + C + B + D

  - Programmers can alter the precedence and associativity rules by placing parentheses

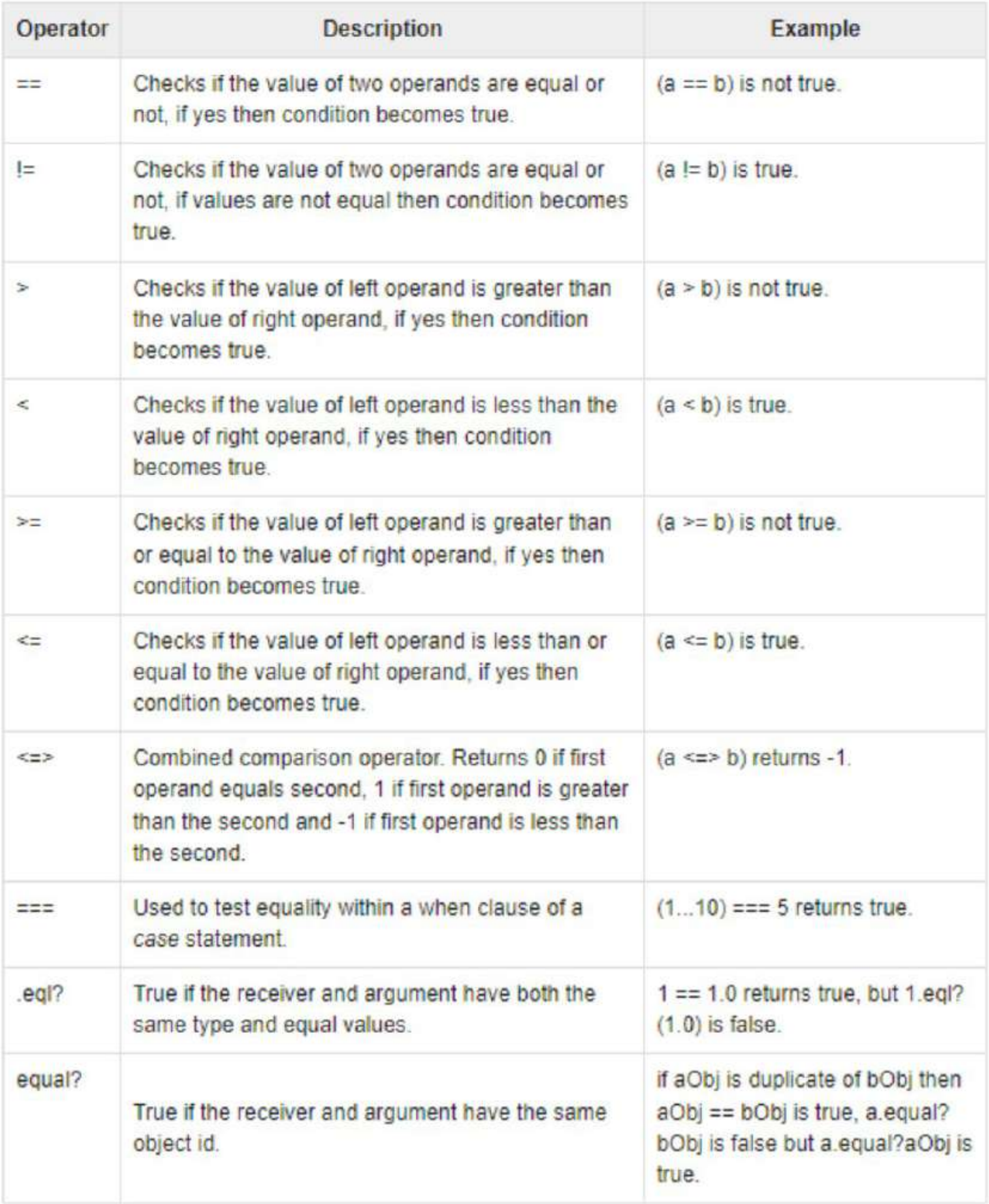
(A + B) + (C + D)

7.5 RELATIONAL AND BOOLEAN EXPRESSIONS

- A relational operator is an operator that compares the values of its two operands
- Relational Expressions
  - Use relational operators and operands of various types
  - Evaluate to some Boolean representation
- Operator symbols used vary somewhat among languages (!=, /=, ~=, .NE., <>, #)


PYTHON RELATIONAL OPERATORS

EXAMPLE, RUBY



Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.	(a <=> b) returns -1.
===	Used to test equality within a when clause of a *case* statement.	(1...10) === 5 returns true.
.eql?	True if the receiver and argument have both the same type and equal values.	1 == 1.0 returns true, but 1.eql?(1.0) is false.
equal?	True if the receiver and argument have the same object id.	if aObj is duplicate of bObj then aObj == bObj is true, a.equal?bObj is false but a.equal?aObj is true.

BOOLEAN EXPRESSIONS

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators : AND, OR, &&, ||

not

x	not x
False	True
True	False

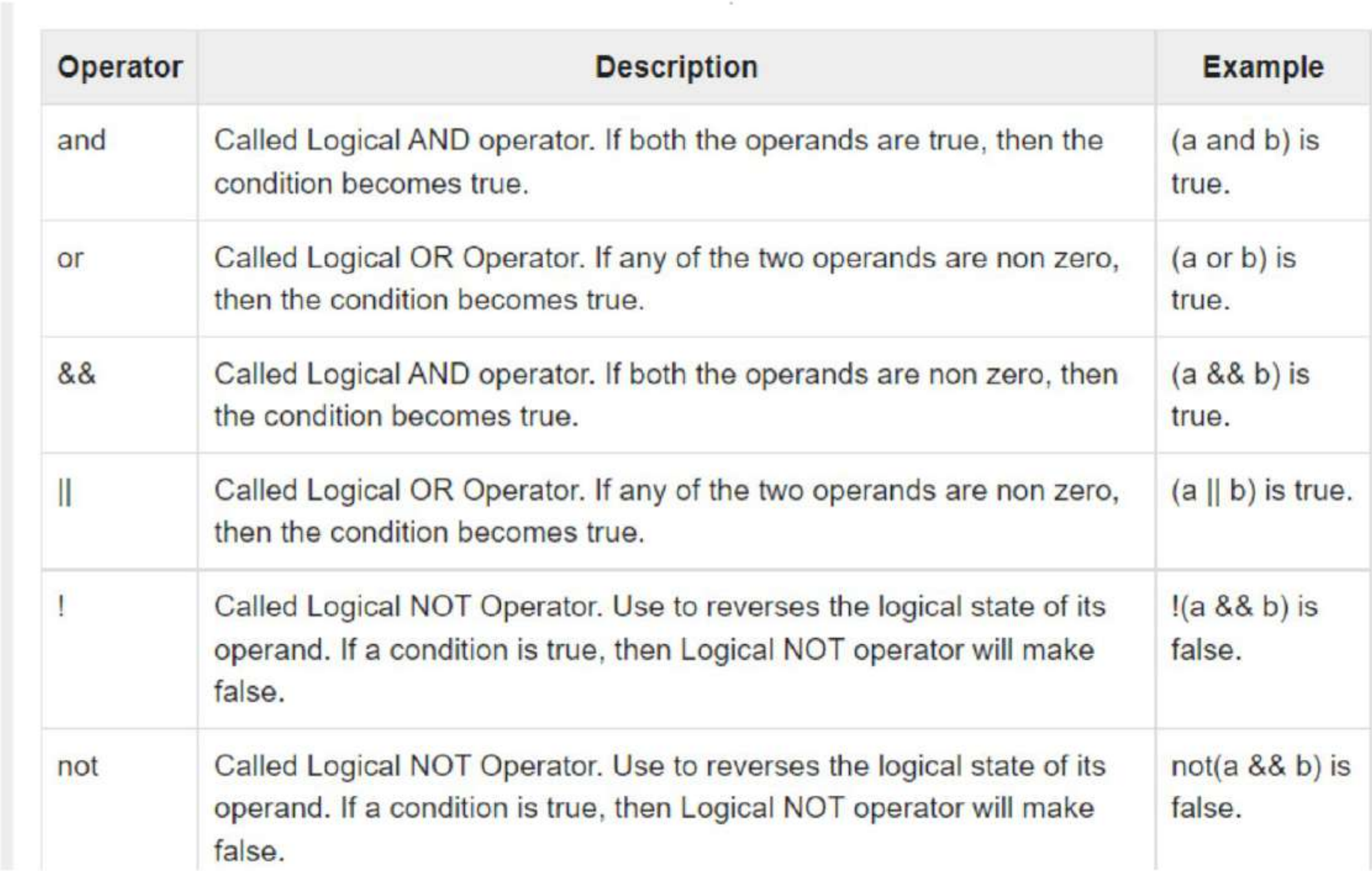
and

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

or

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

EXAMPLE, RUBY



Operator	Description	Example
and	Called Logical AND operator. If both the operands are true, then the condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a or b) is true.
&&	Called Logical AND operator. If both the operands are non zero, then the condition becomes true.	(a && b) is true.
||	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a || b) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(a && b) is false.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	not(a && b) is false.

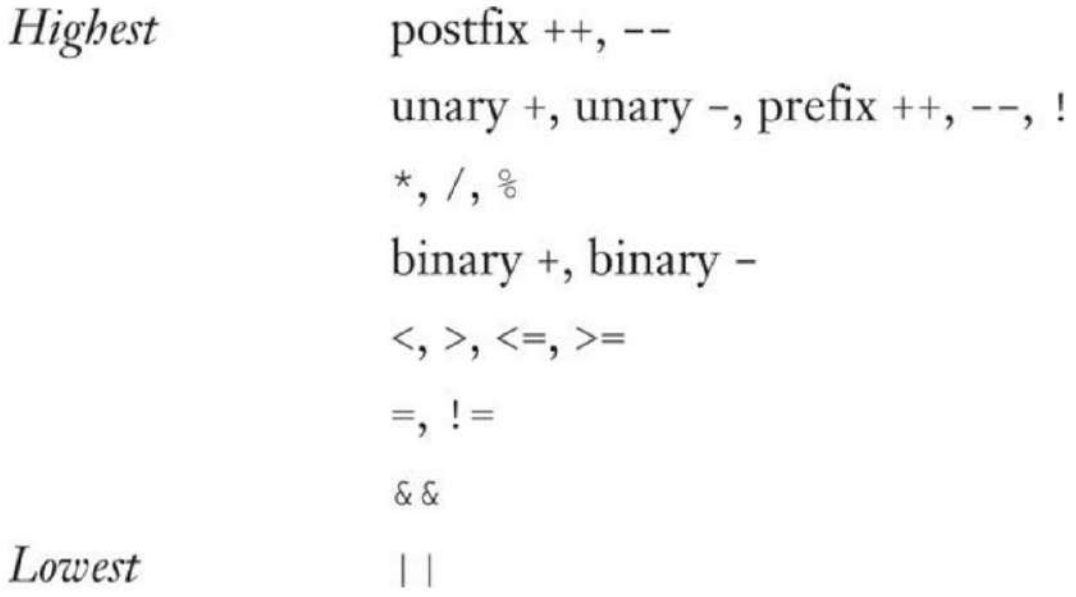
BOOLEAN EXPRESSIONS

- C89 has no Boolean type -- it uses `int` type with 0 for false and nonzero for true
  - eg,

    `a < b < c` is a legal expression:
    - Left operator is evaluated, producing 0 or 1
    - The evaluation result is then compared with the third operand (i.e., c)
    - b is never compared with c

PRECEDENCE

- Arithmetic expressions can be the operands of relational expressions, and relational expressions can be the operands of Boolean expressions → different precedence levels



Highest	postfix ++, --
	unary +, unary -, prefix ++, --, !
	*, /, %
	binary +, binary -
	<, >, <=, >=
	=, !=
	&&
Lowest	||

SHORT CIRCUIT EVALUATION

- An expression in which the result is determined without evaluating all of the operands and/or operators
- Examples;

```
(13 * a) * (b / 13 - 1)
```

  - If `a` is zero, there is no need to evaluate `(b /13 - 1)`

```
(a > b) || (b++ / 3)
```

  - B may not be incremented

SHORT CIRCUIT EVALUATION

- AND operation does not short circuit in
  - FORTRAN (1956)
  - BASIC (1964) and VB
  - Pascal (1970)
  - SQL (1974)
- Problem with non-short-circuit evaluation

```
index = 0;
while (index <= length) && (LIST[index] != value)
      index++;
```

  - When `index=length`, `LIST[index]` will cause an indexing problem