CHAPTER 7

Expressions & Assignments

Hannah Ruth Labana
Marc Nathaniel Valeros

Topic Outline

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Introduction

- **Expressions** are the fundamental means of specifying computations in a programming language

- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation

- Essence of imperative languages is dominant role of assignment statements

Arithmetic Expressions

- **Automatic evaluation** of arithmetic expressions.

- Most of its characteristics were inherited from conventions that had evolved in mathematics.

- In programming languages, Arithmetic Expressions consists of: **Operators, Operands,Parenthesis, Function Calls**

Arithmetic Expressions

- In most programming language their **binary operators** are *infix* (operators appear between operands).

    - In **Perl**, some of its operators are prefix (precedes their operands)
    - In **Scheme and Lisp**, most operators are prefix
    - In **C**, most unary operators are prefix, -- and ++ operators can either be prefix or postfix

Arithmetic Expressions

- What is the purpose of arithmetic expression?
  - To **specify an arithmetic computation**.

- **Two actions** in the implementation of such computation:
  - **Fetch** the operands from memory
  - **Execute** arithmetic operations on those operands

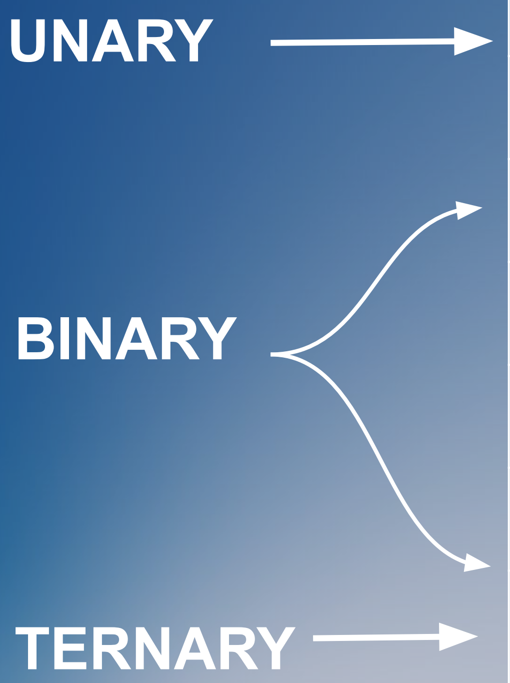
Arithmetic Expressions: Operators

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

Arithmetic Expressions: Design Issues

Design issues for arithmetic expressions

- Operator precedence rules?
- Operator associativity rules?
- Order of operand evaluation?
- Operand evaluation side effects?
- Operator overloading?
- Type mixing in expressions?


UNARY

BINARY

TERNARY

Operator	Type
++, --	Unary Operator
+, - , * , / , %, **, //	Arithmetic Operator
< , <=, >, >=, ==, !=	Relational Operator
&&, ||, !	Logical Operator
&, |, <<, >>, ~, ^	Bitwise Operator
=, +=, -=, *=, /=, %=	Assignment Operator
?:	Ternary or Conditional Operator

Operator Evaluation Order

- The **operator precedence** and **associativity rules** *dictate the order of evaluation of its operators*

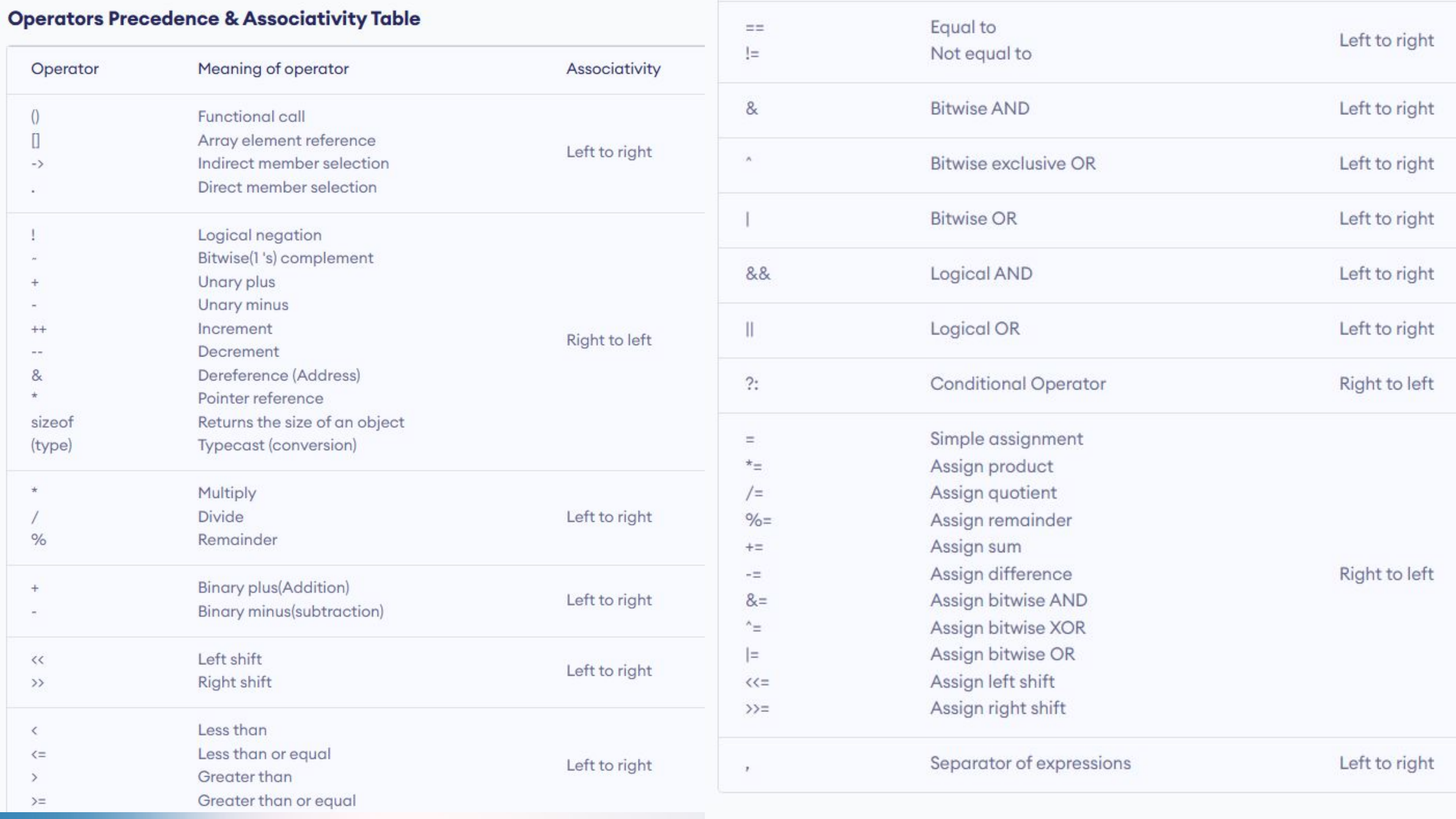
Arithmetic Expressions: Precedence

- The **value** of an expression depends on the order of evaluation of the operators in the expression

- Given:
  - **a + b * c**
  - where a = 3, b = 4, c = 5
- What is the value if evaluated from **left to right** and **right to left** ? **35, 23**

Arithmetic Expressions: Operator Precedence Rules

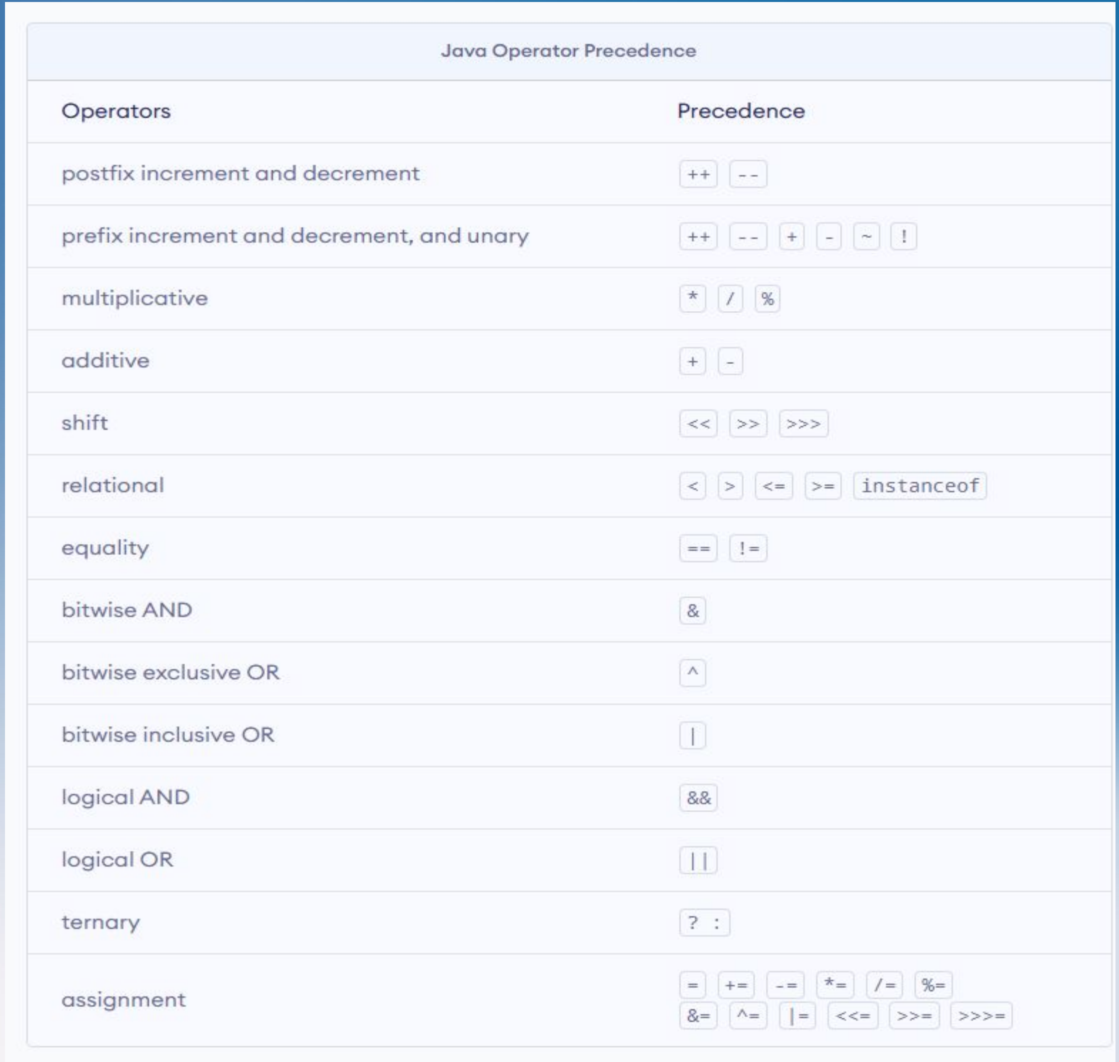
- The **operator precedence rules** for expression evaluation define the order in which “adjacent” operators of *different precedence levels* are evaluated.
- **Typical precedence levels**
    - Parentheses
    - unary operators
    - ** (if the language supports it)
    - *, /
    - +, -



Operators Precedence & Associativity Table

Operator	Meaning of operator	Associativity
()	Functional call	Left to right
[]	Array element reference	
->	Indirect member selection	
.	Direct member selection	
!	Logical negation	Right to left
~	Bitwise(1 's) complement	
+	Unary plus	
-	Unary minus	
++	Increment	
--	Decrement	
&	Dereference (Address)	
*	Pointer reference	
sizeof	Returns the size of an object	
(type)	Typecast (conversion)	
*	Multiply	Left to right
/	Divide	
%	Remainder	
+	Binary plus(Addition)	Left to right
-	Binary minus(subtraction)	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal to	Left to right
!=	Not equal to	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional Operator	Right to left
=	Simple assignment	Right to left
*=	Assign product	
/=	Assign quotient	
%=	Assign remainder	
+=	Assign sum	
-=	Assign difference	
&=	Assign bitwise AND	
^=	Assign bitwise XOR	
|=	Assign bitwise OR	
<<=	Assign left shift	
>>=	Assign right shift	
,	Separator of expressions	Left to right

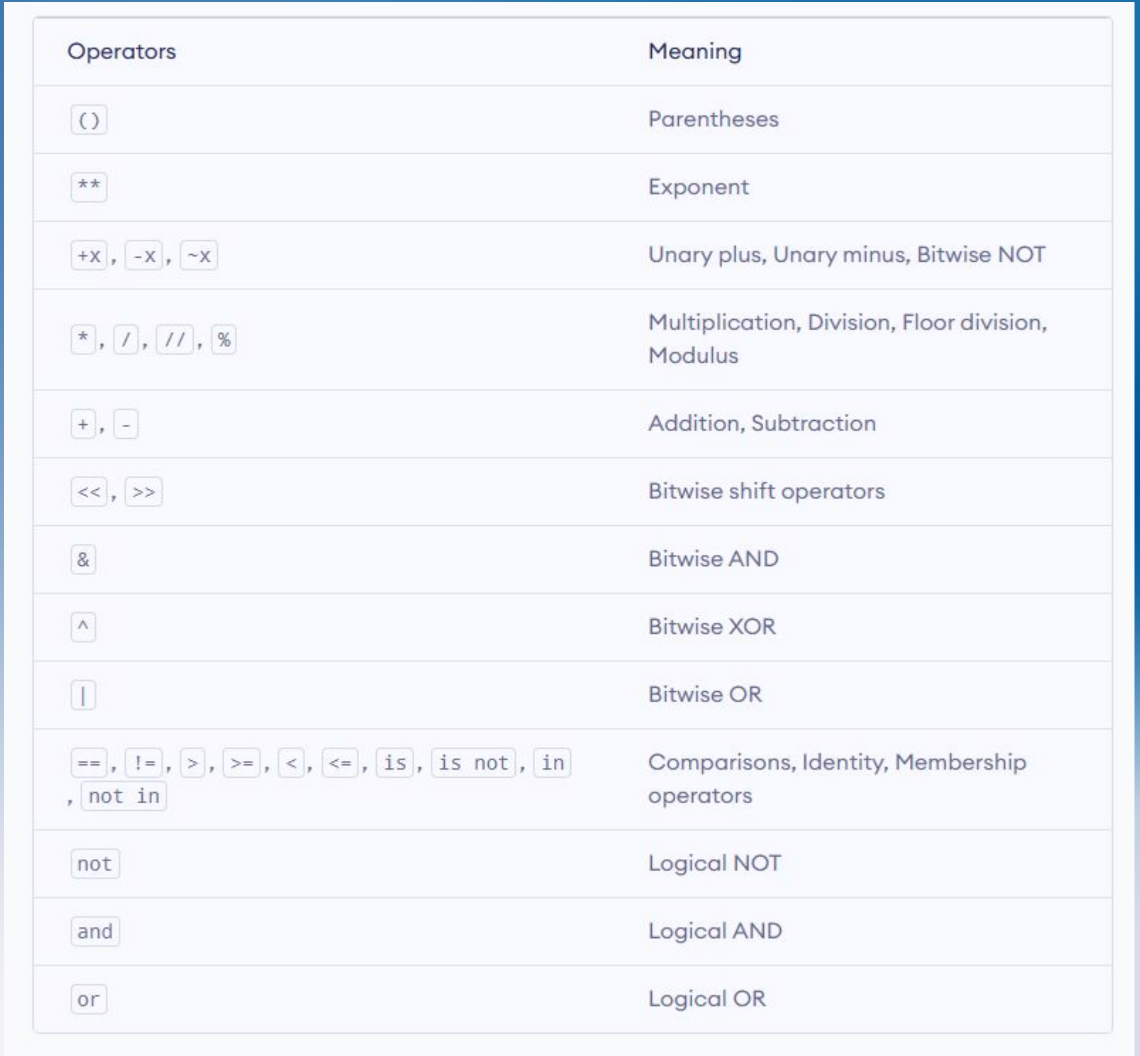
Arithmetic Expressions: Operator Precedence (Java)



Java Operator Precedence

Operators	Precedence
postfix increment and decrement	++ --
prefix increment and decrement, and unary	++ -- + - ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Arithmetic Expressions: Operator Precedence (Python)



Operators	Meaning
`()`	Parentheses
`**`	Exponent
`+x`, `-x`, `~x`	Unary plus, Unary minus, Bitwise NOT
`*`, `/`, `//`, `%`	Multiplication, Division, Floor division, Modulus
`+`, `-`	Addition, Subtraction
`<<`, `>>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

Arithmetic Expressions: Operator Precedence Rules

- Why does the operator precedence rules of the common imperative languages nearly the same?
- *Because they are based on mathematics*

- What are common programming languages that have the exponentiation operator?
- *Fortran, Ruby, Visual Basic, Ada and Python*

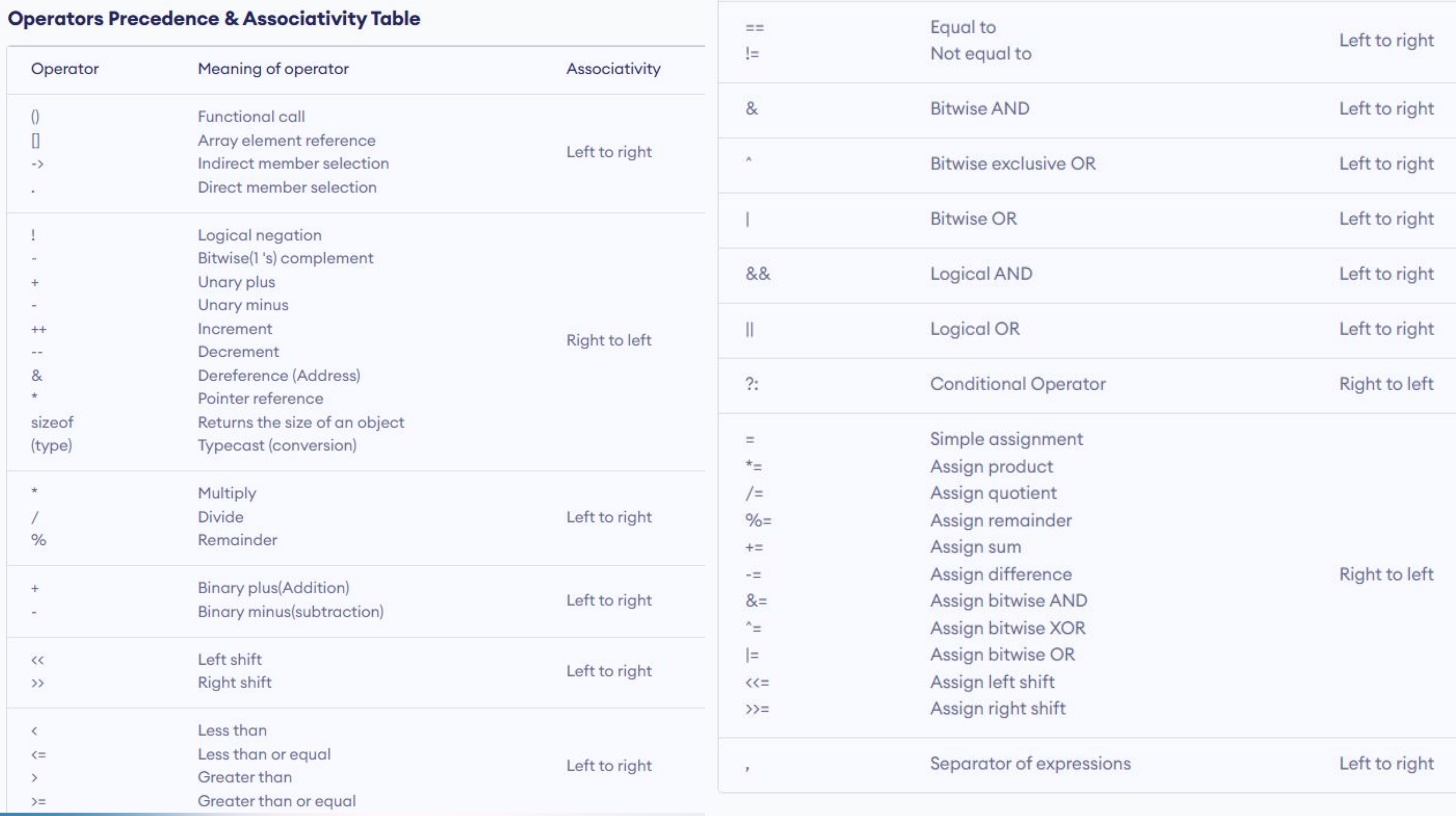
Arithmetic Expressions: Associativity

- Precedence accounts for only some of the rules of operator evaluation; **associativity rules** also affect it.
- **Given:**
  - `a - b + c - d`
  - The precedence rules does not apply to operators with the same precedence. Hence, the associativity rules will be used.

Arithmetic Expressions: Operator Associativity Rule

- 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 (e.g., in **FORTRAN**)



Operators Precedence & Associativity Table

Operator	Meaning of operator	Associativity
()	Functional call	Left to right
[]	Array element reference	
->	Indirect member selection	
.	Direct member selection	
!	Logical negation	Right to left
~	Bitwise(1 's) complement	
+	Unary plus	
-	Unary minus	
++	Increment	
--	Decrement	
&	Dereference (Address)	
*	Pointer reference	
sizeof	Returns the size of an object	
(type)	Typecast (conversion)	
*	Multiply	Left to right
/	Divide	
%	Remainder	
+	Binary plus(Addition)	Left to right
-	Binary minus(subtraction)	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal to	Left to right
!=	Not equal to	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional Operator	Right to left
=	Simple assignment	Right to left
*=	Assign product	
/=	Assign quotient	
%=	Assign remainder	
+=	Assign sum	
-=	Assign difference	
&=	Assign bitwise AND	
^=	Assign bitwise XOR	
|=	Assign bitwise OR	
<<=	Assign left shift	
>>=	Assign right shift	
,	Separator of expressions	Left to right

Arithmetic Expressions: Operator Associativity Rule

- What programming language wherein **all of its operators have equal precedence** and all of its operators associate **from right to left ?**
  - APL

- Precedence and associativity rules can be overriden with ____________?
  - parenthesis

Arithmetic Expressions: Python

```
2 + 3 ** 2 * 4
```

```
10 / 2 ** 2 + 3
```

```
2 ** 3 ** 2
```

```
2 + 3 ** 2 * 4
2 + 9 * 4
2 + 36
38
```

```
10 / 2 ** 2 + 3
10 / 4 + 3
2.5 + 3
5.5
```

```
2 ** 3 ** 2
2 ** 9
512
```

Expressions in Ruby

Ruby

- Everything in Ruby is an object including its operators which is implemented as **methods**
- Can be **overriden** by **application programs**
- For example, the ‘**+**’ operator is actually a method named + that belongs to the **Numeric** class.
- Similarly, the ‘[ ]’ operator for array indexing is actually a method named [ ] that belongs to the **Array** class.

Expressions in Ruby

```ruby
class MyClass
  def +(other)
    "Hello, #{other}!"
  end
end

obj = MyClass.new
puts obj + "world" #=> "Hello, world!"
```

This instance demonstrates that we have created a method named + specifically for objects of the MyClass class. If we utilize the + operator on an object of MyClass, it will execute our personalized method in place of its default action.

Expressions in Scheme

Scheme (and Common Lisp)

- All arithmetic and logic operations are by explicitly called subprograms
- For example, to specify the C expression `a + b * c` in Lisp `(+ a (* b c))`
- In this expression, + and * are the names of functions

Arithmetic Expressions: Conditional Expressions

Conditional Expressions

C-based languages (e.g., C, C++)

`expression_1 ? expression_2 : expression_3`

Example:

```
if (count == 0)
         average = 0
else
          average = sum /count
```

Can be written as:

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

Arithmetic Expressions: Operand Evaluation Order

Operand evaluation Order

- **Variables**: fetch the value from memory
- **Constants**: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
- **Parenthesized expressions**: evaluate all operands and operators first
- When an **operand is a function call**

Arithmetic Expressions: Potentials for Side Effects

Functional side effects: when a function changes a **two-way parameter** or **a non-local variable**

Problem with functional side effects:

- when a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(&a);
```

Arithmetic Expressions: Potentials for Side Effects

```
a = 10;
b = a + fun(&a);   /* assume fun returns 10 and
changes the value of its parameter to 20 */
```

```
/* variable is evaluated first */
b = 10 + fun(&a)
b = 10 + 10
b = 20
```

```
/*function call is evaluated first */
      b = a + 10
      b = 20 + 10
      b = 30
```

Functional Side Effects

Two possible solutions to the problem

1. Write the language definition to disallow functional side effects
   - No two-way parameters in functions
   - No non-local references in functions

Advantage: it works!
Disadvantage: inflexibility of one-way parameters and lack of non-local references

Functional Side Effects

Two possible solutions to the problem

2. Write the **language definition** to demand that operand evaluation order be fixed

Disadvantage: limits some compiler optimizations
Java requires that operands appear to be evaluated in left-to-right order

Referential Transparency

A program has the property of **referential transparency** if any two expressions in the program that have the **same value can be substituted for one another** anywhere in the program.

```
    result1 = (fun(a) + b) / (fun(a) - c);
    temp = fun(a);
    result2 = (temp + b) / (temp - c);
```

If fun has no side effects, result1 = result2
Otherwise, not, and referential transparency is violated

Referential Transparency cont.

- Advantage of referential transparency
  - **Semantics** of a program is much easier to understand if it has referential transparency

- Programs in **pure functional languages** (do not have variables) are referentially transparent
  - Functions cannot have state, which would be stored in local variables

  - The value of a function depends only on its parameters

Overloaded Operators

- Use of an operator for more than one purpose is called ***operator overloading***

- Some are common (e.g., + for **int** and **float**)

- Some are potential trouble (e.g., * in C and C++)
    - Loss of compiler error detection (omission of an operand should be a detectable error)
    - Some loss of readability

Overloaded Operators

In **Python**, we can change the way operators work for user-defined types.

For example, the + operator will perform **arithmetic addition** on two numbers, **merge two lists**, or **concatenate two strings**.

Overloaded Operators (continued)

- When sensibly used it can aid to **readability**
- Potential problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense

Programming Language	Operator Overloading	
Java	• doesn't allow user defined operator overloading	❌
C	• C doesn't support any form of overloading	❌
Python	• Can overload all existing operators but we can't create a new operator	✅

List of Python Operators that can be overloaded:

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

How does python overload its operators?

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3)  # Output: (4, 6)
```

Type Conversions

- A **narrowing conversion**
  - is one that converts an object to a type that cannot include all of the values of the original type
    - e.g., float to int

- A **widening conversion**
  - is one in which an object is converted to a type that can include at least approximations to all of the values of the original type
    - e.g., int to float

Type Conversions

Narrowing conversion

Python: float to int:

```
x = 10.5
y = int(x)  # Narrowing conversion from float to int
```

Java: double to int:

```
double x = 10.5;
int y = (int) x;  // Narrowing conversion from double to int
```

Type Conversions

Widening conversion

Python: int to float:

```
x = 10
y = float(x)  # Widening conversion from int to float
```

Java: byte to short:

```
byte x = 10;
short y = x;  // Widening conversion from byte to short
```

Type Conversions

In **C, Python and Java**, **widening conversions** can be done *implicitly* by the interpreter or compiler, while **narrowing conversions** require *explicit* conversion or casting.

Type Conversions: Mixed Mode

- A **mixed-mode expression** is one that has operands of different types.

- A **coercion** is an implicit type conversion while an explicit type is called **casting.**

Disadvantage of coercions:

- They decrease in the type error detection ability of the compiler

In most languages, all numeric types are coerced in expressions, using **widening conversions**

In ML and F#, there are no coercions in expressions

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - C: (**int**)angle
  - F#: **float**(sum)

Note that F#’s syntax is similar to that of function calls

Errors in Expressions

- Causes
  - Inherent limitations of arithmetic
    e.g., division by zero
  - Limitations of computer arithmetic
    e.g. overflow
- Often ignored by the run-time system

Relational Expressions

- **Relational expression -** has two operands and one relational operator
  - Value is *Boolean* type, true or false
  - Except when Boolean is not a type included in the language
    - Lisp - empty list for false, any other value for true
    - C - integer, 1 or non-zero if true and 0 if false
  - Types of operands that can be used for relational operators
    - Numeric, strings, enumeration

Relational Expressions

- **Relational operator** - operator that compares the values of its two operands
  - Always have lower precedence than arithmetic operators
  - C Relational Operators:
    - - == , equal to
    - - > , greater than
    - - < , less than
    - - >= , greater than or equal to
    - - <= , less than or equal to
    - - != , not equal to

Relational Expressions

- Inequality throughout languages differ
  - C-based , !=
  - Fortran 95+ , .NE. or <>
  - ML & F# , <>
- JavaScript and PHP have two additional relational operators
  - === and !==
  - Prevent their operands from being coerced

Boolean Expressions

- Boolean expressions consist of:
  - Boolean variables
  - Boolean constants
  - Relational expressions
  - Boolean operators
    - Usually include AND, OR, NOT, exclusive OR
    - Usually only take Boolean operands
    - Produce Boolean values
- In the mathematics of Boolean algebras, OR and AND operators have EQUAL PRECEDENCE, but in C, AND has a higher precedence than OR

Boolean Expressions

- When there is no Boolean type, ***readability suffers***
  - In other imperative languages, any non-Boolean expression used as an operand of a Boolean operator is detected as an error

Short Circuit Evaluation

- An expression in which the result is determined without evaluating all of the operands and/or operators
- Example: `(13 * a) * (b / 13 – 1)`
  - If a is zero, there is no need to evaluate (b /13 - 1)
- Problem with non-short-circuit evaluation

```
index = 0;
while ((index < listlen) && (list[index] != key))
index = index + 1;
```

When `index == listlen`, `LIST[index]` will cause an indexing problem (assuming `LIST` is `listen - 1` long)

Short Circuit Evaluation

- If program correctness depends on the side effect, short-circuit evaluation can result in a serious error

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

- b changes only when a <= b, so if the programmer assumes b to change every time this expression is evaluated during execution, the program will fail

Assignment Statements

- one of the central constructs in imperative languages since it is what dictates the values of variables

```
<target_var> <assign_operator> <expression>
```

- The assignment operator
  - = Fortran, BASIC, the C-based languages
  - := Ada
- = can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

```
($flag ? $total : $subtotal) = 0
```

which is equivalent to

```
if ($flag) {
    $total = 0
} else {
    $subtotal = 0
}
```

Assignment Statements: Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C and the C-based languages
- Example

```
a = a + b
```

can be written as

```
a += b
```

Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
- Examples

```
sum = ++count   (count incremented, then assigned to sum)
sum = count++   (count assigned to sum, then incremented)
count++         (count incremented)
-count++        (count incremented then negated)
```

Assignment as an Expression

- In the C-based languages, Perl, and JavaScript, the assignment statement produces a result and can be used as an operand

```
while ((ch = getchar())!= EOF){…}
```

```
ch = getchar() is carried out; the result (assigned to ch)
is used as a conditional value for the while statement
```

Multiple Assignments

- Perl and Ruby allow multiple-target multiple-source assignments

```
($first, $second, $third) = (20, 30, 40);
```

- Also, the following is legal and performs an interchange:

```
($first, $second) = ($second, $first);
```

Assignment in Functional Languages

- Identifiers in functional languages are only names of values
- ML:

  Names are bound to values with val

  ```
  val fruit = apples + oranges;
  ```

  If another `val` for fruit follows, it is a new and different name

CHAPTER 7

Expressions & Assignments

Hannah Ruth Labana
Marc Nathaniel Valeros

CHAPTER 7

Standing Questions

Hannah Ruth Labana
Marc Nathaniel Valeros

Does Java support operator overloading?

- Java does not allow operator overloading.
    - String concatenation with the plus operator.
    - Aside from that, Java does not allow you to design your own operators.

Type Conversion in Java

- When you assign the value of one data type to another, you should be aware of the compatibility of the data type.

- **Widening (automatically)**
  - smaller data type to the larger type size
  - `byte -> short -> char -> int -> long -> float -> double`
- **Narrowing (manually)**
  - larger data type to a smaller size type
  - double -> float -> long -> int -> char -> short -> byte

Widening Conversion

- **Implicit Conversion (Automatic)**
    - two data types are compatible
    - value of a smaller data type to a larger data type
    - numeric data types are compatible with each other
    - no implicit conversion (automatic) is supported from numeric type to char or boolean

Widening Casting

```
public class Conversion{
public static void main(String[] args)
{
int i = 200;

//automatic type conversion
long l = i;

//automatic type conversion
float f = l;

System.out.println("Int value "+i);
System.out.println("Long value "+l);
System.out.println("Float value "+f);
}
}
```

Output:

```
Int value 200
Long value 200
Float value 200.0
```

Narrowing Conversion

- **Explicit Conversion (Manual)**
  - for incompatible data types
  - value of larger data type to a smaller data type

Narrowing Casting

```
//Java program to illustrate explicit type
conversion
public class Narrowing
{
public static void main(String[] args)
{
double d = 200.06;

//explicit type casting
long l = (long)d;

//explicit type casting
int i = (int)l;
System.out.println("Double Data type value "+d);

//fractional part lost
System.out.println("Long Data type value "+l);

//fractional part lost
System.out.println("Int Data type value "+i);
}
}
```

Output:

```
Double Data type value 200.06
Long Data type value 200
Int Data type value 200
```