Bilkent University
Fall 2023-2024

CS315
Programming Languages

Homework 1
Boolean Expressions

Section 003

Yasemin Akın
22101782

November 27, Monday

Contents

1.0. Part A – Design Decisions in Each Language

1.1. Dart

- In Dart, there are three Boolean operators. These are AND (&&), OR (||), and NOT (!) operators. AND outputs true if both operands represent the logical value true; otherwise, AND outputs false. OR outputs false only if both operands represent false; otherwise, OR outputs true. NOT is a unary operator that reverses its only operand's logical value. Here is an illustration of how Boolean operators are used and evaluated in Dart and the code segment's output:

```
bool a = true;
bool b = false;
print("1. Boolean operators in Dart:");
print("AND '&&' (a && b): ${a && b}");
print("OR '||' (a || b): ${a || b}");
print("NOT '!' (!a): ${!a}");
```

Outputs:

```
1. Boolean operators in Dart:
AND '&&' (a && b): false
OR '||' (a || b): true
NOT '!' (!a): false
```

- Dart directly uses true and false Boolean values with Boolean operators and does not implement truthy or falsy values. Therefore, it strictly uses (has strict typing for) bool type for Boolean expressions because Dart is a statically typed language, and Non-Boolean values cannot be directly used as operands without explicit conversion. The following code segment will be commented out in the source code file because all of the lines are generating compile-time errors due to number and string type usage in Boolean contexts:

```
print("Number in boolean context:", 5);
print("String in boolean context:", "hello");
print("String in Boolean context: " + ("string" && a));
```

```
print("Number in Boolean context: " + (0 && a));
```

Here is the correct way to use non-boolean values in the boolean context in Dart:

```
int number = 5;
String str = "hello";
print("Is number non-zero?: ${number != 0}");
print("Is string non-empty?: ${str.isNotEmpty}");
```

Outputs:

Is number non-zero?: true
Is string non-empty?: true

- In Dart, the Boolean operators' precedence rule is as follows: ! > && > ||. This means that when used in the same Boolean expression, the NOT operator will be evaluated first, then the AND operator, then the OR operator.

```
bool a = true;
bool b = false;
bool c = false;
bool d = true;
print("a || b && !c: ${a || b && !c}");
  print("a || b == c && d != a: ${a || b == c && d != a}");
print("!a && b || c == d && d != a: ${!a && b || c == d && d != a}"); //
Equivalent to ((!a) && b) || ((c == d) && (d != a))
```

Outputs:

a || b && !c: true
a || b == c && d != a: true
!a && b || c == d && d != a: false

Above, the first expression is evaluated as: a || b && !c → a || b && true → a || false → true

The second expression (== or != has higher precedence than Boolean operators in Dart): a || b == c && d != a → a || true && false → a || false → true
The third expression: !a && b || c == d && d != a → !a && b || false && false → false && b || false && false → false || false → false

- Boolean operators '&&' and '||' in Dart are left-associative, meaning that the evaluation starts from the left-most operation in an expression where only one is used. Since '!' is a unary operator, associativity is irrelevant for this operator. Below is an example code segment showcasing this concept and its output. The operation between a and b is evaluated first, while the operation between the upcoming result from the previous expressions and d is evaluated last:

```
print("\n4. Operator associativity rules in Dart: left-associative");
print("a && b && c && d: ${a && b && c && d}"); // Evaluates as (((a && b) && c) && d)
```

Outputs:

```
4. Operator associativity rules in Dart: left-associative
a && b && c && d: false
```

- Operands in a Boolean context are evaluated from left to right in Dart.

The following examples display this rule. As you can understand from the outputs, the left-most operands (that are functions returning bool type and printing identifying statements) are called initially:

```
print("\n5. Operand evaluation order in Dart: left-to-right");
bool result1 = firstMethod() && secondMethod();
print(result1);
bool result2 = secondMethod() || firstMethod();
print(result2);
bool firstMethod() {
print("First method called");
return true;
}
```

```
bool secondMethod() {
print("Second method called");
return false;
}
```

Outputs:

5. Operand evaluation order in Dart: left-to-right
First method called
Second method called
false
Second method called
First method called
true

- Dart utilizes short-circuit evaluation. Short-circuit evaluation in programming is a method where the second operand in a logical operation (like AND &&, OR ||) is evaluated only if the first operand does not suffice to determine the value of the expression. Dart uses this method while operating with && or ||. In the case of &&, if the first operand is false, the second operand is never evaluated because the first one is sufficient to determine the result. In the case of ||, if the first operand is true, the second operand is never evaluated because the first one is sufficient to determine the result:

```
print("\n6. Short-circuit evaluation in Dart:");
print("false && secondMethod(): ${false && secondMethod()}"); //
secondMethod is not called
print("true || secondMethod(): ${true || secondMethod()}");   //
secondMethod is not called
 bool result = b && (a || check());
print("Result of b && (a || check()): $result"); //check() is not called because
value of the second operand for the or operator does not matter at this
example.
```

```
bool check() {
        print("check() function called");
        return true;
}
```

Outputs:

```
6. Short-circuit evaluation in Dart:
false && secondMethod(): false
true || secondMethod(): true
Result of b && (a || check()): false
```

1.2. Go

- Go supports three primary Boolean operators: AND (&&), OR (||), and NOT (!). The AND operator returns true if both operands are true and false otherwise. The OR operator returns true if at least one operand is true and false if both are false. The NOT operator inverts the boolean value of its operand. In the table below, the left column illustrates Boolean operators in Go, and the right column is the output of that code segment:

a := true b := false fmt.Println("1. Boolean operators in Go:") fmt.Println("AND '&&' (a && b):", a && b) fmt.Println("OR '||' (a || b):", a || b) fmt.Println("NOT '!' (!a):", !a)	1. Boolean operators in Go: AND '&&' (a && b): false OR '||' (a || b): true NOT '!' (!a): false

- Go, like Dart, strictly uses bool type for Boolean expressions. It does not implement 'truthy' or 'falsy' values. Non-Boolean values cannot be directly used as operands without explicit conversion. Uncommenting any of the following commented lines will result in a compilation error in Go:

```
var number int = 5
var str string = "hello"
var ptr *int = &number
var slice []int = []int{1, 2, 3}
// fmt.Println("Number in boolean context:", bool(number))
// fmt.Println("String in boolean context:", bool(str))
// fmt.Println("Pointer in boolean context:", bool(ptr))
// fmt.Println("Slice in boolean context:", bool(slice))
```

- In Go, the operator precedence is NOT > AND > OR. When used in the same Boolean expression, the NOT operator is evaluated first, followed by the AND operator, then the OR operator. Below, complex expressions illustrate this concept. The first expression is equivalent to (a || (b && (!c))), and the third expression is equal to (a || (b == c) && (d != a)). For the second expression, you can see that if one tries to calculate the result of the expression without obeying the Boolean operator precedence rule, the result comes out as wrong.

```
a := true
b := false
c := false
d := true
 fmt.Println("a || b && !c:", a || b && !c)
 fmt.Println("a || b == c && d != a:", a || b == c && d != a)
 fmt.Println("!a && b || c == d && d != a:", !a && b || c == d && d != a)
```

Outputs:

```
a || b && !c: true
a || b == c && d != a: true
!a && b || c == d && d != a: false
```

Second expression evaluation (without obeying the precedence rule): a || b == c && d != a → a || true && false → false && false → false

Second expression evaluation (with obeying the precedence rule): a || b == c && d != a → a || true && false → a || false → true

- Boolean operators are left-associative in Go, except the NOT operator since it is a unary operator. Talking about associativity for NOT is unnecessary. Go evaluates the first example like there are parentheses placed like this: (((a && b) && c) && d). For the second expression, it will be like this: ((x > y || y > x) || y == x):

```
fmt.Println("a && b && c && d:", a && b && c && d)
x := 0
y := 1
fmt.Println("x > y || y > x || y == x:", x > y || y > x || y == x)
```

Outputs:

```
a && b && c && d: false
x > y || y > x || y == x: true
```

- Go evaluates operands from left to right in a sentence. Example code segments show that by placing function calls that return Boolean type as the operands, initially, the left-most function call is made:

```
result1 := firstMethod() && secondMethod()
fmt.Println(result1)
result2 := secondMethod() || firstMethod()
fmt.Println(result2)
```

Outputs:

```
First method called
Second method called
false
Second method called
First method called
true
```

- Go utilizes short-circuit evaluation for && and ||. In &&, if the first operand is false, the second operand is not evaluated; in ||, if the first operand is true, the second operand is not considered. Below, in both expressions, secondMethod() remains uncalled, showing how short-circuit evaluation works in Go briefly:

```
fmt.Println("false && secondMethod():", false && secondMethod())
fmt.Println("true || secondMethod():", true || secondMethod())
```

Output:

```
false && secondMethod(): false
true || secondMethod(): true
```

1.3. Javascript

- JavaScript supports three primary Boolean operators: AND (&&), OR (||), and NOT (!). The AND operator (&&) returns true only if both operands are true; otherwise, it returns false. The OR operator (||) outputs true if at least one of its operands is true and false only if both operands are false. The NOT operator (!) is a unary operator that reverses the logical value of its operand. The code demonstrates the use of these operators, with a being true and b being false. As a result, a && b outputs false, a || b outputs true, and !a outputs false:

```
let a = true;
let b = false;
console.log(`a = ${a}`);
console.log(`b = ${b}`);
console.log("1. Boolean operators in JavaScript:");
console.log(`AND '&&' (a && b): ${a && b}`);
console.log(`OR '||' (a || b): ${a || b}`);
console.log(`NOT '!' (!a): ${!a}`);
```

Outputs:

```
a = true
b = false
1. Boolean operators in JavaScript:
AND '&&' (a && b): false
OR '||' (a || b): true
NOT '!' (!a): false
```

- JavaScript has a concept of 'truthy' and 'falsy' values. A 'falsy' value is something that evaluates to false when converted to a boolean and includes values such as false, 0, -0, 0n (BigInt), "" (empty string), null, undefined, and NaN. Conversely, 'truthy' values are true in a Boolean context, like non-empty strings, non-zero numbers, arrays, objects, and functions. The code illustrates various examples of 'truthy' and 'falsy' values. For instance, !!"false" (double negation converts the string "false" to a boolean) evaluates to true, demonstrating it is a 'truthy' value:

`console.log(!!false);`	false
`console.log(!!0);`	false
`console.log(!!-0);`	false
`console.log(!!0n);`	false
`console.log(!!"");`	false
`console.log(!!null);`	false
`console.log(!!undefined);`	false
`console.log(!!NaN);`	true
	true
`console.log(!!"0");`	true
`console.log(!!"false");`	true
`console.log(!![]);`	true
`console.log(!!{});`	true
`console.log(!!function(){});`	
`console.log(!!42);`	

- The precedence of Boolean operators in JavaScript is NOT (!) > AND (&&) > OR (||). This means that in an expression with mixed operators, the NOT operator is evaluated first, followed by the AND, and then the OR operator. In the example, a || b && !c is evaluated as a || (b && (!c)), with the NOT operator taking precedence, then AND, and finally, OR:

```
let c = false;
console.log(`a || b && !c: ${a || b && !c}`);
let d = true;
console.log(`a || b === c && d != a: ${a || b === c && d != a}`);
console.log(`!a && b || c === d && d != a: ${!a && b || c === d && d != a}`);
```

Outputs:

```
a || b && !c: true
a || b === c && d != a: true
!a && b || c === d && d != a: false
```

- In JavaScript, the Boolean operators && and || are left-associative, meaning that the evaluation proceeds from left to right in expressions with these operators. The ! operator, being unary, doesn't have associativity concerns. The provided code a && b && c && d is evaluated as (((a && b) && c) && d), showing the left associativity:

```
console.log(`a && b && c && d: ${a && b && c && d}`);
let x = 0;
let y = 1;
console.log(`x > y || y > x || y === x: ${x > y || y > x || y === x}`);
```

Outputs:

```
a && b && c && d: false
x > y || y > x || y === x: true
```

- JavaScript evaluates operands in a Boolean context from left to right. This is evident when functions or expressions with side effects are used as operands. In the example, firstMethod() && secondMethod() calls firstMethod first and then secondMethod. However, firstMethod() || secondMethod() only calls firstMethod() because the result of the OR expression is determined after the first operand is evaluated to be true:

```
function firstMethod() {
   console.log("First method called");
   return true;
}

function secondMethod() {
   console.log("Second method called");
   return false;
}

firstMethod() && secondMethod();
firstMethod() || secondMethod();
```

Outputs:

```
First method called
Second method called
First method called
```

- JavaScript uses short-circuit evaluation for && and ||. In an && expression, if the first operand is false, the second operand is not evaluated since the result can already be determined. Similarly, in an || expression, if the first operand is true, the second operand is not evaluated. The code false && secondMethod() does not call secondMethod, and true || secondMethod() does not call secondMethod either, showcasing the short-circuit behavior in JavaScript:

```
console.log(false && secondMethod());
console.log(true || secondMethod());
let result = b && (a || check());
console.log(`Result of b && (a || check()): ${result}`); // check() is not called
function check() {
   console.log("check() function called");
   return true;
}
```

Outputs:

```
true
Result of b && (a || check()): false
```

1.4. Lua

- In Lua, the basic Boolean operators are AND (and), OR (or), and NOT (not). The AND operator (and) returns true if both operands are true; otherwise, it returns false. The OR operator (or) returns true if at least one of its operands is true and false only if both operands are false. The NOT operator (not) is a unary operator that inverts the logical value of its operand. The provided code demonstrates these operators, with a being true and b being false. Hence, a and b outputs false, a or b outputs true, and not a outputs false:

```
local a = true
local b = false
print("a = ", a)
print("b = ", b)

print("1. Boolean operators in Lua:")
print("AND 'and'(a and b): ", a and b)
print("OR 'or' (a or b): ", a or b)
print("NOT 'not' (not a): ", not a)
```

Outputs:

```
a =   true
b =   false
1. Boolean operators in Lua:
AND 'and'(a and b):   false
OR 'or' (a or b):   true
NOT 'not' (not a):  false
```

- Lua has a unique approach to truthy and falsy values: only false and nil are considered falsy. All other values, including 0 and "" (empty string), are truthy. The code showcases that 1, 0, and an empty string are all truthy in Lua, as they evaluate to true in Boolean contexts. In contrast, nil is falsy, as it evaluates to false:

```
print("\n2. Data types for operands of boolean operators, Boolean values (Truthy and Falsy values) in Lua: in Lua, only 'false' and 'nil' are 'falsy'. Everything else is 'truthy'.")
print("Boolean value of 1: ", 1 and true or false) -- true ('truthy' value)
print("Boolean value of nil: ", nil and true or false) -- false ('falsy' value)
print("Boolean value of 0: ", 0 and true or false) -- true (0 is 'truthy')
print("Boolean value of an empty string: ", "" and true or false) -- true (empty string is 'truthy')
```

Outputs:

```
2. Data types for operands of boolean operators, Boolean values (Truthy and Falsy values) in Lua: in Lua, only 'false' and 'nil' are 'falsy'. Everything else is 'truthy'.
Boolean value of 1:   true
Boolean value of nil:   false
Boolean value of 0:   true
Boolean value of an empty string:   true
```

- Lua's precedence for Boolean operators is NOT (not) > AND (and) > OR (or). This means that in a mixed expression, NOT is evaluated first, followed by AND, then OR. The NOT operator has the highest precedence in the example a or b and not c, so the expression is evaluated as a or (b and (not c)), leading to true. This demonstrates how operator precedence affects the evaluation of Boolean expressions:

```
local c = false
print("a or b and not c: ", a or b and not c)
local d = true
print("a or b == c and d ~= a", a or b == c and d ~= a) -- Equivalent to (a or ((b == c) and (d ~= a)))
```

```
print("not a and b or c == d and d ~= a: ", not a and b or c == d and d ~= a) -- Equivalent to
(((not a) and b) or ((c == d) and (d ~= a)))
-- ((false and b) or ((c == d) and (d != a)))
-- ((false and b) or (false and false))
-- (false or false)
-- false
```

Outputs:

```
a or b and not c:   true
a or b == c and d ~= a  true
not a and b or c == d and d ~= a:   false
```

- In Lua, Boolean operators ‘and’ and ‘or’ are left-associative. This means that the leftmost operations are evaluated first in expressions with multiple same operators. The code a and b and c and d is evaluated as (((a and b) and c) and d), indicating the left-to-right associativity in Lua:

```
print("a and b and c and d: ", a and b and c and d) -- Evaluates as (((a and b) and c) and d)

local x = 0
local y = 1
print("x > y or y > x or y == x: ", x > y or y > x or y == x) -- Evaluates as ((x > y or y > x) or y
== x)
```

Outputs:

```
a and b and c and d:  false
x > y or y > x or y == x:   true
```

- Lua evaluates operands in Boolean expressions from left to right. This order is important, especially in expressions where operands have side effects, such as function calls. The example demonstrates this: first_method()

and second_method() calls first_method first and then second_method, while second_method() or first_method() calls second_method first and then first_method, adhering to the left-to-right evaluation order:

```
local function first_method()
  print("First method called")
  return true
end

local function second_method()
  print("Second method called")
  return false
end

if first_method() and second_method() then end
if second_method() or first_method() then end
```

Outputs:

```
First method called
Second method called
Second method called
First method called
```

- Lua uses short-circuit evaluation for and and or. For and, if the first operand is false, the second operand is not evaluated. For or, if the first operand is true, the second operand is not evaluated. In the code, b and second_method() does not call second_method since b is false. Similarly, a or second_method() does not call second_method as a is true. This demonstrates Lua's use of short-circuit evaluation to optimize performance and avoid unnecessary evaluations:

```
local result = b and second_method()
print("Result of b and second_method(): ", result)
```

```
result = a or second_method()
print("Result of a or second_method(): ", result)
```

Outputs:

```
Result of b and second_method():  false
Result of a or second_method():   true
```

1.5. Python

- Python includes AND (and), OR (or), and NOT (not) as its Boolean operators. The and operator returns True only if both operands are True. Thus, a and b results in False since b is False. The or operator yields True if at least one operand is True, making a or b return True as a is True. The not operator inverses the boolean value, so not a becomes False as a is True.

```
a = True
b = False
print(f"a = {a}")
print(f"b = {b}")
print("\n1. Boolean operators in Python:")
print(f"AND 'and' (a and b): {a and b}")
print(f"OR 'or' (a or b): {a or b}")
print(f"NOT 'not' (not a): {not a}")
```

Outputs:

```
a = True
b = False

1. Boolean operators in Python:
AND 'and' (a and b): False
```

```
OR 'or' (a or b): True
NOT 'not' (not a): False
```

- Python treats False, None, 0, empty sequences, and collections as falsy. In contrast, Truthy values include non-zero numbers, non-empty strings, and other non-null objects. This behavior allows non-Boolean types like "Yasemin", -5, [], or "" to be used in Boolean contexts, evaluated based on their truthiness or falsiness:

```
# In Python, non-boolean types can be operands in boolean expressions
print("0 as Falsy:", bool(0))  # False, because 0 is considered Falsy
print("-5 as Truthy:", bool(-5)) # True, because non-zero numbers are considered Truthy
print("[1,4,6] as Truthy", bool([1,4,6]))
print("Yasemin as Truthy", bool("Yasemin"))
print("Empty string as Falsy", bool(""))
print("[] as Falsy", bool([]))
print("None as Falsy", bool(None))

non_bool = "I am not a boolean"
if non_bool:
    print("In Python, non-boolean types can be used in boolean contexts. This will print.")
number = 5
if number:
    print("Non-zero numbers are truthy in Python. This will print.")

print("Non-boolean operands:", 5 and 0)  # 0, because 0 is falsy in Python
print("Non-boolean operands:", "Yasemin" and "")  # "", because "" is falsy in Python
```

Outputs:

```
0 as Falsy: False
-5 as Truthy: True
[1,4,6] as Truthy True
```

```
Yasemin as Truthy True
Empty string as Falsy False
[] as Falsy False
None as Falsy False
In Python, non-boolean types can be used in boolean contexts. This will print.
Non-zero numbers are truthy in Python. This will print.
Non-boolean operands: 0
Non-boolean operands:
```

- Python's precedence for Boolean operators is not > and > or. This order affects the evaluation of expressions like a or b and not c, which is interpreted as a or (b and (not c)), indicating the prioritization of not, followed by and, then or. Additionally, Python's precedence places == and != higher than Boolean operators:

```
c = False
print(f"a or b and not c: {a or b and not c}")  # Equivalent to (a or (b and (not c)))

d = True
print(f"a or b == c and d != a: {a or b == c and d != a}") # Equivalent to (a or ((b == c) and (d != a)))
print(f"not a and b or c == d and d != a: {not a and b or c == d and d != a}") # Equivalent to (((not a) and b) or ((c == d) and (d != a)))
# (((not a) and b) or (false and false))
# ((false and b) or (false and false))
# (false or false)
# false
```

Outputs:

```
a or b and not c: True
a or b == c and d != a: True
not a and b or c == d and d != a: False
```

- Boolean operators in Python are left-associative, meaning expressions with multiple same operators are processed from left to right. For example, a and b and c and d is interpreted as (((a and b) and c) and d), showing Python's consistent approach to evaluating left-to-right in chained Boolean operations.

```
print(f"a and b and c and d: {a and b and c and d}") # Evaluates as (((a and b) and c) and d)

x = 0
y = 1
print(f"x > y or y > x or y == x: {x > y or y > x or y == x}") # Evaluates as ((x > y or y > x) or y == x)
```

Outputs:

```
a and b and c and d: False
x > y or y > x or y == x: True
```

- Python evaluates operands from left to right, which is particularly noticeable when functions with side effects are involved. In the example, first_method() and second_method() first calls first_method, followed by second_method. On the other hand, second_method() or first_method() evaluates second_method first, maintaining the left-to-right sequence:

```
def first_method():
    print("First method called")
    return True

def second_method():
    print("Second method called")
    return False

first_method() and second_method() # Outputs: "First method called" then "Second method called"
```

```
second_method() or first_method() # Outputs: "Second method called" then "First method called"
```

Outputs:

```
First method called
Second method called
Second method called
First method called
```

- Python employs short-circuit evaluation in and and or operations. If the first operand in an and expression is False, the second operand is not evaluated, as seen when False and second_method() does not trigger second_method. In an or expression, the evaluation stops if the first operand is True. Hence, True or second_method() does not execute second_method. This optimization prevents unnecessary computations:

```
# For 'and', if the first operand is false, the second operand is not evaluated.
print(False and second_method())   # "Second method called" is not printed

# For 'or', if the first operand is true, the second operand is not evaluated.
print(True or second_method())    # "Second method called" is not printed

def check():
    print("check() function called")
    return True

result = b and (a or check())
print(f"Result of b and (a or check()): {result}")  # check() is not called
```

Outputs:

```
False
True
```

```
Result of b and (a or check()): False
```

1.6. Ruby

- Ruby supports the Boolean operators AND (&&), OR (||), and NOT (!). The && operator returns true only if both operands are true; thus, a && b results in false since b is false. The || operator yields true if at least one operand is true, so a || b outputs true as a is true. The ! operator reverses the Boolean value, turning !a into false, given that a is true.

```
# 1. Boolean operators
a = true
b = false
puts "a = #{a}"
puts "b = #{b}"

puts "1. Boolean operators in Ruby:"
puts "AND '&&'(a && b): #{a && b}" # false
puts "OR '||' (a || b): #{a || b}" # true
puts "NOT '!' (!a): #{!a}" # false
```

Outputs:

```
a = true
b = false
1. Boolean operators in Ruby:
AND '&&'(a && b): false
OR '||' (a || b): true
NOT '!' (!a): false
```

- In Ruby, all values are considered truthy except for false and nil, which are falsy. Demonstrating with various examples, the code confirms that while 1 and 0 are truthy, nil is falsy. This feature allows Ruby to use non-boolean types like strings or numbers in Boolean contexts, such as in if conditions, where a

string like "I am not a boolean" or a non-zero number like 5 is evaluated as true:

```
# 'true' and 'false' are the only boolean values. Everything else is truthy, except 'nil'.
puts !!1 # true ('truthy' value)
puts !!nil # false ('falsy' value)
puts !!0 # true (0 is 'truthy')
puts !"" # false
non_bool = "I am not a boolean"
if non_bool
    puts "In Ruby, non-boolean types can be used in boolean contexts. This will print."
end
number = 5
if number
    puts "Numbers other than 0 are also truthy in Ruby. This will print."
end
```

Outputs:

```
true
false
true
false
In Ruby, non-boolean types can be used in boolean contexts. This will print.
Numbers other than 0 are also truthy in Ruby. This will print.
```

- Ruby's precedence for Boolean operators is ! > && > ||. This ordering dictates the evaluation of expressions like a || b && !c, interpreted as a || (b && (!c)). The precedence rule is crucial in understanding how complex Boolean expressions are evaluated, with negation taking priority over logical conjunction and disjunction.

```
# 3. Operator precedence rules
```

```
puts "\n3. Operator precedence rules in Ruby: ! > && > ||"
c = false
puts "a || b && !c: #{a || b && !c}"  # Equivalent to a || (b && (!c))

d = true
puts "a || b == c && d != a: #{a || b == c && d != a}" # Equivalent to (a || ((b == c) && (d != a)))

puts "!a && b || c == d && d != a: #{!a && b || c == d && d != a}" # Equivalent to ((!a) && b) ||
((c == d) && (d != a))
```

Outputs:

```
3. Operator precedence rules in Ruby: ! > && > ||
a || b && !c: true
a || b == c && d != a: true
!a && b || c == d && d != a: false
```

- In Ruby, the Boolean operators && and || are left-associative, meaning in expressions with multiple same operators, the leftmost operations are evaluated first. This is illustrated by the expression a && b && c && d, which is processed as (((a && b) && c) && d). The left-associative nature ensures a predictable evaluation order in chained Boolean operations.

```
# 4. Operator associativity rules
puts "\n4. Operator associativity rules in Ruby: left-associative"
puts "a && b && c && d: #{a && b && c && d}" # Evaluates as (((a && b) && c) && d)

x = 0
y = 1
puts "x > y || y > x || y == x: #{x > y || y > x || y == x}" # Evaluates as ((x > y || y > x) || y == x)
```

Outputs:

```
4. Operator associativity rules in Ruby: left-associative
a && b && c && d: false
x > y || y > x || y == x: true
```

- Ruby evaluates operands from left to right, as showcased when functions with side effects are used as operands. The execution of first_method && second_method first triggers first_method, then second_method. However, in the first_method || second_method scenario, first_method is evaluated first, and since it returns true, second_method is not executed, adhering to the left-to-right evaluation sequence.

```ruby
# 5. Operand evaluation order
puts "\n5. Operand evaluation order in Ruby: left-to-right"
def first_method
  puts "First method called"
  true
end

def second_method
  puts "Second method called"
  false
end

first_method && second_method  # Output: "First method called" then "Second method called"
first_method || second_method  # Output: "First method called" then "Second method called"
```

Outputs:

```
5. Operand evaluation order in Ruby: left-to-right
First method called
Second method called
First method called
```

- Ruby employs short-circuit evaluation for && and ||. In &&, if the first operand is false, the second operand is not evaluated, demonstrated when false && second_method does not call second_method. Conversely, in ||, if the first operand is true, the second operand is not evaluated, hence true || second_method does not trigger second_method. This short-circuit behavior is a key optimization feature in Ruby's evaluation of Boolean expressions:

```ruby
# 6. Short-circuit evaluation
puts "\n6. Short-circuit evaluation:"
# Ruby uses short-circuit evaluation for && and ||.

# For &&, if the first operand is false, the second operand is not evaluated.
puts false && second_method   # "Second method called" is not printed

# For ||, if the first operand is true, the second operand is not evaluated.
puts true || second_method    # "Second method called" is not printed

result = b && (a || check())
puts "Result of b && (a || check()): #{result}" # check() is not called

def check
    puts "check() function called"
    true
end
```

Outputs:

```
6. Short-circuit evaluation:
false
true
Result of b && (a || check()): false
```

1.7. Rust

- Rust, like other programming languages, includes basic Boolean operators: AND (&&), OR (||), and NOT (!). The && operator yields true only if both operands are true, which is why a && b returns false as b is false. The || operator produces true if at least one operand is true, leading to a || b being true because a is true. The ! operator inverts the boolean value, making !a equal to false as a is true:

```
// 1. Boolean operators:
let a = true;
let b = false;
println!("a = {}", a);
println!("b = {}", b);

println!("1. Boolean operators in Rust:");
println!("AND '&&'(a && b): {}", a && b); // false
println!("OR '||' (a || b): {}", a || b); // true
println!("NOT '!' (!a): {}", !a); // false
```

Outputs:

```
a = true
b = false
1. Boolean operators in Rust:
AND '&&'(a && b): false
OR '||' (a || b): true
NOT '!' (!a): false
```

- Rust strictly enforces the use of the bool type for Boolean expressions. Unlike some languages that allow "truthy" and "falsy" values from different data types, Rust requires explicit Boolean values or expressions that evaluate to bool. Attempting to use non-Boolean types like strings or integers directly in Boolean contexts will result in a compilation error.

```
// Using non-bool types as operands will cause a compilation error
// Incorrect usage: using a string in a boolean context
// let non_bool = "I am not a boolean";

// The following line will cause a compilation error
// Uncomment to see the error
// if non_bool {
//     println!("This won't compile");
// }

// Incorrect usage: using an integer in a boolean context
// let number = 5;

// The following line will also cause a compilation error
// Uncomment to see the error
// if number {
//     println!("This won't compile either");
// }
```

- In Rust, operator precedence is defined as ! > && > ||. This precedence order is crucial in determining the outcome of complex Boolean expressions. For instance, a || b && !c is evaluated as a || (b && (!c)), where the NOT operator (!) is evaluated first, followed by AND (&&), and then OR (||). This precedence ensures that logical expressions are evaluated in the intended order. Evaluations of the example expressions below are as shown by the comments next to the expressions:

```
let c = false;
    println!("a || b && !c: {}", a || b && !c); // Equivalent to a || (b && (!c))
    // a || (b && true)
    // a || false
    // true
    let d = true;
```

```
    println!("a || b == c && d != a: {}", a || b == c && d != a); // Equivalent to (a || ((b == c) && (d
!= a)))
    // (a || ((true) && (false))
    // (a || (false))
    // (true)
    println!("!a && b || c == d && d != a: {}", !a && b || c == d && d != a); // Equivalent to ((!a)
&& b) || ((c == d) && (d != a))
    // (false && b) || ((c == d) && (d != a))
    // (false && b) || (false && false)
    // false || false
    // false
```

Outputs:

```
a || b && !c: true
a || b == c && d != a: true
!a && b || c == d && d != a: false
```

- Rust's Boolean operators are left-associative, except for !, which is unary and thus does not have associativity concerns. This means that in expressions with multiple same operators, like a && b && c && d, the evaluation is performed from left to right, which is processed as (((a && b) && c) && d). This left-to-right associativity applies to all Boolean operators in Rust, ensuring a consistent and predictable evaluation order:

```
// 4. Operator associativity rules:
    println!("\n4. Operator associativity rules in Rust: all Boolean operators are left-associative
except '!', which is a unary operator so associativity is irrelevant here.");
    println!("a && b && c && d: {}", a && b && c && d); // Evaluates as (((a && b) && c) && d)
    // ((false && c) && d)
    // (false && d)
    // false
    let x = 0;
```

```
    let y = 1;
    println!("x > y || y > x || y == x: {}", x > y || y > x || y == x); // Evaluates as ((x > y || y > x) || y
== x)
    // (true || y == x)
    // true
```

Outputs:

```
4. Operator associativity rules in Rust: all Boolean operators are left-associative except '!',
which is an unary operator so associativity is irrelevant here.
a && b && c && d: false
x > y || y > x || y == x: true
```

- In Rust, operands are evaluated in a left-to-right order. This is particularly noticeable in expressions where operands have side effects, such as function calls. For example, in func1() && func2(), func1 is called first, followed by func2. This order is consistent across all operations in Rust, making it crucial for understanding how expressions will behave, especially when functions or operations with side effects are involved:

```
// 5. Operand evaluation order:
    println!("\n5. Operand evaluation order in Rust: left-to-right");
    let mut result = func1() && func2(); // func1 called, then func2 called
    println!("{}", result);
    result = func2() || func1(); // func2 called, then func1 called
    println!("{}", result);
fn func1() -> bool { println!("func1 called"); true }


fn func2() -> bool { println!("func2 called"); false }
```

Outputs:

```
5. Operand evaluation order in Rust: left-to-right
func1 called
```

```
func2 called
false
func2 called
func1 called
true
```

- Rust utilizes short-circuit evaluation for && and ||. This means if the first operand in an && expression is false, the second operand is not evaluated, as demonstrated when b && (a || check()) does not call check() because b is false. Similarly, in an || expression, if the first operand is true, the second operand is not evaluated, optimizing performance and preventing unnecessary computations:

```
// 6. Short-circuit evaluation.
    println!("\n6. Short-circuit evaluation:");
    result = b && (a || check());
    println!("Result of b && (a || check()): {}", result); // check() is not called
fn check() -> bool { println!("check() function called");true }
```

Outputs:

```
6. Short-circuit evaluation:
Result of b && (a || check()): false
```

2.0. Part B – Evaluation

Several factors come into play in evaluating the readability and writability of Boolean expressions across Dart, Go, JavaScript, Lua, Python, Ruby, and Rust, including syntax clarity, expressiveness, and ease of understanding. Below is an assessment of these languages in handling Boolean expressions.

Dart

Readability: Dart's syntax for Boolean expressions is quite clear and similar to other C-family languages, which can be familiar and readable for developers from those backgrounds. The selection of logical operators such as &&, || and !, precedence, associativity, evaluation order rules, and short-circuit functionality are usual among many programming languages. Also, the explicit requirement for Boolean expressions (no truthy or falsy values) enhances clarity.

Writability: Dart's strict typing means all Boolean expressions must resolve to bool, preventing errors due to unintentional truthy/falsy values. This enhances writability by reducing ambiguity, though it might require more code for type conversions.

Go

Readability: Go offers simplicity and clarity in its syntax, enhancing the readability of Boolean expressions. Like Dart, the lack of truthy and falsy values leads to a more predictable code.

Writability: The strict type requirements and simplicity in Go's design aid in preventing bugs but may limit expressiveness compared to more dynamic languages.

Javascript

Readability: Javascript's flexibility with truthy and falsy values can sometimes reduce readability due to less predictability. Expressions like if (a) where a could be non-Boolean might be confusing. However, standard operations like a && b are straightforward.

Writability: Javascript is highly expressive, especially in Boolean contexts, which can be advantageous for quick scripting and dynamic coding. Expressions like a || b can return the actual value of a or b, not just a Boolean, offering greater flexibility.

Lua

Readability: Lua's unique approach (where only ‘nil’ and ‘false’ are falsy) is simple but can be initially unintuitive for programmers coming from other languages. An expression like if a then is clear when a is a Boolean but less so for other data types.

Additionally, using self-explanatory keywords to represent logical operators (and, or, not) can make reading and understanding the Lua code easier, especially for beginners.

Writability: Lua's flexibility in treating all values except 'nil' and 'false' as true allows for compact and expressive Boolean expressions.

Python

Readability: Python's Boolean expressions are extremely readable due to the language's emphasis on simplicity and clarity, exemplified by explicit Boolean operators (and, or, not). For example, if a or not b stands out for itself.

Writability: Python is also very writable, allowing for expressive and compact code; although its dynamic typing can sometimes lead to less predictable behavior, mostly its handling of truthy/falsy values is predictable (0, None, empty collections are falsy), allowing for concise expressions like if my_list: to check for a non-empty list.

Ruby

Readability: Ruby's syntax is designed for readability, with a natural, almost English-like style. However, like JavaScript, its flexible truthy/falsy concept (where only nil and false are falsy) can cause readability issues in more complex expressions.

Writability: Ruby allows for very expressive Boolean expressions and offers significant flexibility, which can be both a strength and a potential source of complexity.

Rust

Readability: Rust's strict type system and similarity to C++ and C# make its Boolean expressions very readable for developers with a background in systems programming. For instance, if a && b || !c is clear and concise.

Writability: While Rust ensures safe and predictable Boolean expressions, its syntax can be verbose, so the learning curve can be steeper than in more dynamic languages.

Conclusion

According to the evaluation above, here's a table assigning points out of 5 for readability and writability for each of the seven programming languages and their brief justifications to sum up the evaluation:

Language	Readability (out of 5)	Writability (out of 5)	Overall Score (out of 10)
Dart	4	4	8
Go	4.5	3.5	8
JavaScript	3	4.5	7.5
Lua	3.5	4	7.5
Python	5	4.5	9.5
Ruby	4	4.5	8.5
Rust	4	3	7

Justifications:

- Dart: Offers clear syntax similar to C-family languages (4/5 in readability), but strict typing can sometimes limit writability (4/5).

- Go: High readability due to simplicity and explicit typing (4.5/5), but somewhat less writable due to strict typing (3.5/5).

- Javascript: Flexibility leads to high writability (4.5/5), but dynamic typing can sometimes affect readability (3/5).

- Lua: Fairly readable with its unique approach to truthiness (3.5/5) and quite writable due to its flexibility (4/5).

- Python: Highly readable thanks to its explicit and clear syntax (5/5) and very writable due to its expressiveness (4.5/5).

- Ruby: Very readable due to its natural, English-like syntax (4/5) and highly writable with great expressiveness (4.5/5).

- **Rust:** Readable for those familiar with systems programming languages (4/5) but less writable due to verbosity and complexity (3/5).

According to the score evaluation, Python leads the other programming languages in the context of readability and the writability of Boolean expressions.

3.0. Part C – Learning Strategy

Material and Tools Used

- Online Compilers/Interpreters: For each language (Dart, Go, Javascript, Lua, Python, Ruby, and Rust), I utilized online compilers/interpreters. This allowed me to write, execute, and test code snippets efficiently, ensuring the behaviors observed were accurate and up to date with the latest language versions.
  URLs of the online compiler/interpreters:
  - Dart: https://dart.dev/#try-dart
  - Go: https://go.dev/play/
  - Javascript: https://www.programiz.com/javascript/online-compiler/#google_vignette
  - Lua: https://www.jdoodle.com/execute-lua-online/
  - Python: https://www.online-python.com
  - Ruby: https://www.jdoodle.com/execute-ruby-online
  - Rust: https://play.rust-lang.org/?version=stable&mode=debug&edition=2021

- Official Language Documentation: I referred to the official documentation of Dart [1], Go [2], Javascript [3], Lua [4], Python [5], Ruby [6], and Rust [7] to understand each language's nuances, especially concerning Boolean operations. This was crucial for grasping the specifics of operator precedence, associativity, and other language-specific details.

- Programming Forums and Communities: Websites like Stack Overflow [8], GeeksforGeeks [9], GitHub [10], and other language forums [11] were instrumental in clarifying doubts and understanding common practices and idiomatic ways of using Boolean expressions in each language.

- Video Tutorials: Preparing source code with languages I had never implemented or worked on before was difficult. To understand the general syntaxes of Lua [12], Ruby [13], Rust [14], and Dart [15], I consulted a few videos prepared for this purpose. It has always been very useful for me to visually see how source codes are written and how they progress, and I think it also makes it easier for me.

- Text Editor: WebStorm, Visual Studio Code, and Xcode were used as simple text editors to draft and organize the code snippets before running them in online compilers. This helped in keeping the code organized and easily accessible.

Experiments Performed

For each programming language, I created and tested source code to explore the following aspects:

- Boolean Operators: Demonstrated using AND, OR, and NOT operators. Code snippets were crafted using AND, OR, and NOT operations across all languages. This approach was instrumental in understanding the fundamental logical operations in each environment. For instance, variables a and b were used to explore combinations like a && b, a || b, and !a.

- Data Types and Boolean Values: Examined how different data types are treated as Boolean values, focusing on what each language considers as truthy and falsy. This was achieved by experimenting with different data types such as integers, strings, and null values, and converting them to Boolean where applicable. This experiment was eye-opening in revealing the subtle differences across languages. For example, while languages like Python have a clear set of falsy values (like 0, False, None, empty collections), others like JavaScript and Ruby consider almost everything as truthy except a few falsy values.

- Operator Precedence Rules: Complex Boolean expressions involving various operators were constructed to observe which operations take precedence over others. This was essential in understanding how different languages prioritize operations, which can significantly influence the outcome of expressions.

- Operator Associativity Rules: Tested to determine how expressions with the same operator are grouped and evaluated, particularly in the absence of parentheses. This involved creating chained operations like a && b && c to test if the language evaluates them left-to-right or right-to-left, which is crucial for understanding how expressions are parsed and executed.

- Operand Evaluation Order: particularly focused on scenarios where operands have side effects, such as function calls. Using function calls as operands in Boolean expressions made it possible to determine the sequence in which these operands are evaluated and whether they follow a consistent pattern across languages.

- Short-Circuit Evaluation: Designed tests to see if and how each language implements short-circuit logic in Boolean expressions. Using expressions like false && func() and true || func(), the experiment tested whether the second operand (function call) gets executed, providing insights into how languages optimize performance by avoiding unnecessary evaluations. This aspect of the experiment highlighted the efficiency and optimization strategies used by different programming languages to evaluate Boolean expressions.

Personal Communication

For this homework assignment, talking to others really helped me out. Whenever I got stuck, especially on the trickier parts of Boolean operators in different programming languages, I contacted the Teaching Assistant (TA). They were great at explaining things in a way that made more sense to me, clearing up areas I was confused about. I also had some good discussions with my friends about the assignment. We shared our ways of doing things and discussed our challenges. Hearing how they handled certain parts of the homework gave me new ideas and helped me see things from different angles. These chats with the TA and friends were super helpful. They didn't just help me get past the tough spots but made me feel more confident about my understanding of the topic. It was like we were all figuring it out together, which made the whole process a lot more engaging and informative.

References

[1] https://dart.dev

[2] https://go.dev

[3] https://www.javascript.com

[4] https://www.lua.org

[5] https://www.python.org

[6] https://www.ruby-lang.org/tr/

[7] https://www.rust-lang.org

[8] https://stackoverflow.com

[9] https://www.geeksforgeeks.org

[10] https://github.com

[11] https://forum.freecodecamp.org

[12] https://youtu.be/kgiEF1frHQ8?feature=shared

[13] https://youtu.be/8wZ2ZD--VTk?feature=shared

[14] https://youtu.be/br3GIIQeefY?feature=shared

[15] https://youtu.be/Ej_Pcr4uC2Q?feature=shared