CS 315 - Programming Languages
Homework 1

Ghulam Ahmed 22101001

Boolean Expressions in Dart, Go, Javascript, Lua, Python, Ruby, and Rust

Design decisions in each language

1. Dart

Boolean Operators: Dart uses **&&**, | |, **!** to show logical AND, OR, and NOT operations.
Data Types and Boolean Values: Dart treats all values other than false and null as truthy.
Operator Precedence: **!** > **&&** > | |.
Associativity: **&&** and | | are left-associative, **!** is right-associative.
Evaluation Order: Left-to-right.
Short-Circuit Evaluation: Implemented for **&&** and | |.

Here is the source code explaining the design features above:

22101001_ahmed_ghulam.dart

```dart
void main() {
  // Boolean operators
  print('Boolean Operators:');
  print('true && false: ${true && false}'); // AND
  print('true || false: ${true || false}'); // OR
  print('!true: ${!true}'); // NOT

  // Data types and Boolean values
  print('\nData Types and Boolean Values:');
  var truthyValue = 1; // Non-zero numbers are considered truthy in Dart
  var falsyValue = 0; // Zero is considered falsy
  print('1 (Truthy) && 0 (Falsy): ${truthyValue != 0 && falsyValue == 0}');
  print('null (Falsy): ${null}');

  // Operator precedence
  // The precedence of && over || is shown.
  print('\nOperator Precedence:');
  print('false || true && false: ${false || true && false}');
  // Equivalent to false || (true && false)

  // Associativity
  // Shows that && and || are left-associative.
  print('\nAssociativity:');
  print('true && false || true && true: ${true && false || true && true}');
  // Equivalent to ((true && false) || true) && true

  // Evaluation order
  // Shows left-to-right evaluation, with the sideEffect() function not being called due to short-circuiting in logical expressions.
  print('\nEvaluation Order:');
  print('false && (sideEffect() || true): ${false && (sideEffect() || true)}');

  // Short-circuit evaluation
  // Shows how Dart does not evaluate the second operand if the first operand of && or || is sufficient to determine the result.
  print('\nShort-Circuit Evaluation:');
  print('true || (sideEffect() && false): ${true || (sideEffect() && false)}'); // sideEffect is not called due to short-circuit
}

bool sideEffect() {
  print('Side effect function called');
  return true;
}
```

2. Go

Boolean Operators: Go uses **&&**, **||**, **!** for logical AND, OR, and NOT operations.
Data Types and Boolean Values: Go uses the bool type for boolean expressions. No concept of truthy/falsy as in dynamically typed languages.
Operator Precedence: **!** > **&&** > **||**.
Associativity: All operators are left-associative.
Evaluation Order: Left-to-right.
Short-Circuit Evaluation: Short-circuit implemented for **&&** and **||**.

Here is the source code explaining the design features above:

22101001_ahmed_ghulam.go

```go
package main

import (
    "fmt"
)

func main() {
    // Boolean operators
    fmt.Println("Boolean Operators:")
    fmt.Printf("true && false: %v\n", true && false) // AND
    fmt.Printf("true || false: %v\n", true || false)       // OR
    fmt.Printf("!true: %v\n", !true)                       // NOT

    // Data types and Boolean values
    // Non-boolean values are not implicitly converted to boolean.
    fmt.Println("\nData Types and Boolean Values:")
    var truthyValue int = 1
    // In Go, only boolean values are considered in boolean expressions
    var falsyValue int = 0
    fmt.Printf("Non-boolean values (1, 0): %v, %v\n", truthyValue, falsyValue)

    // Operator precedence
    // It shows how && takes precedence over ||
    fmt.Println("\nOperator Precedence:")
    fmt.Printf("false || true && false: %v\n", false || true && false)  // Equivalent to false || (true && false)

    // Associativity
    // Shows that && and || are left-associative.
    fmt.Println("\nAssociativity:")
    fmt.Printf("true && false || true && true: %v\n", true && false || true && true) // Equivalent to ((true && false) || true) && true

    // Evaluation order
    // Shows left-to-right evaluation, with the sideEffect() function not being called due to short-circuiting in logical expressions.
    fmt.Println("\nEvaluation Order:")
    fmt.Printf("false && sideEffect() || true: %v\n", false && sideEffect() || true) // sideEffect is not called due to short-circuit

    // Short-circuit evaluation
    //  Shows how Go does not evaluate the second operand if the first operand of && or || is sufficient to determine the result.
    fmt.Println("\nShort-Circuit Evaluation:")
    fmt.Printf("true || sideEffect() && false: %v\n", true || sideEffect() && false) // sideEffect is not called due to short-circuit
}

func sideEffect() bool {
    fmt.Println("Side effect function called")
    return true
}
```

3. JavaScript

Boolean Operators: JavaScript uses **&&**, ||,**!** for logical AND, OR, and NOT operations.
Data Types and Boolean Values: Dynamically typed. Truthy values include non-zero numbers, non-null objects, etc.; falsy values include **0**, **null**, **undefined**, **NaN**, "", and **false**.
Operator Precedence:**!** > **&&** > ||.
Associativity: All operators are left-associative.
Evaluation Order: Left-to-right.
Short-Circuit Evaluation: Implemented for **&&** and ||.

Here is the source code explaining the design features above:

```javascript
function booleanDemo() {
    // Boolean operators
    console.log("Boolean Operators:");
    console.log("true && false:", true && false); // AND
    console.log("true || false:", true || false); // OR
    console.log("!true:", !true); // NOT

    // Data types and Boolean values
    // JavaScript is dynamically typed, and this part shows how different types (numbers, strings) behave in boolean contexts.
    console.log("\nData Types and Boolean Values:");
    console.log("1 (Truthy) && 0 (Falsy):", 1 && 0); // Non-zero numbers are truthy, 0 is falsy
    console.log("'string' (Truthy) || '' (Falsy):", 'string' || ''); // Non-empty strings are truthy, empty string is falsy

    // Operator precedence
    // Shows how && has higher precedence than ||.
    console.log("\nOperator Precedence:");
    console.log("false || true && false:", false || true && false); // Equivalent to false || (true && false)

    // Associativity
    // Shows that both && and || are left-associative.
    console.log("\nAssociativity:");
    console.log("true && false || true && true:", true && false || true && true); // Equivalent to ((true && false) || true) && true

    // Evaluation order
    // Shows the left-to-right evaluation, with the sideEffect() function not being called due to short-circuiting in logical expressions.
    console.log("\nEvaluation Order:");
    console.log("false && sideEffect() || true:", false && sideEffect() || true); // sideEffect is not called due to short-circuit

    // Short-circuit evaluation
    // Shows how JavaScript doesn't evaluate the second operand of && or || if the first operand is sufficient to determine the result.
    console.log("\nShort-Circuit Evaluation:");
    console.log("true || sideEffect() && false:", true || sideEffect() && false); // sideEffect is not called due to short-circuit
}

function sideEffect() {
    console.log("Side effect function called");
    return true;
}

booleanDemo();
```

4. Lua

Boolean Operators: Lua uses **`and`**, **`or`**, **`not`** for logical operations.
Data Types and Boolean Values: Dynamically typed. Everything is truthy except **`false`** and **`nil`**.
Operator Precedence: **`not`** > **`and`** > **`or`**.
Associativity: and and or are left-associative, not is right-associative.
Evaluation Order: Left-to-right.
Short-Circuit Evaluation: Short-circuit implemented for **`and`** & **`or`**.

Here is the source code explaining the design features above:

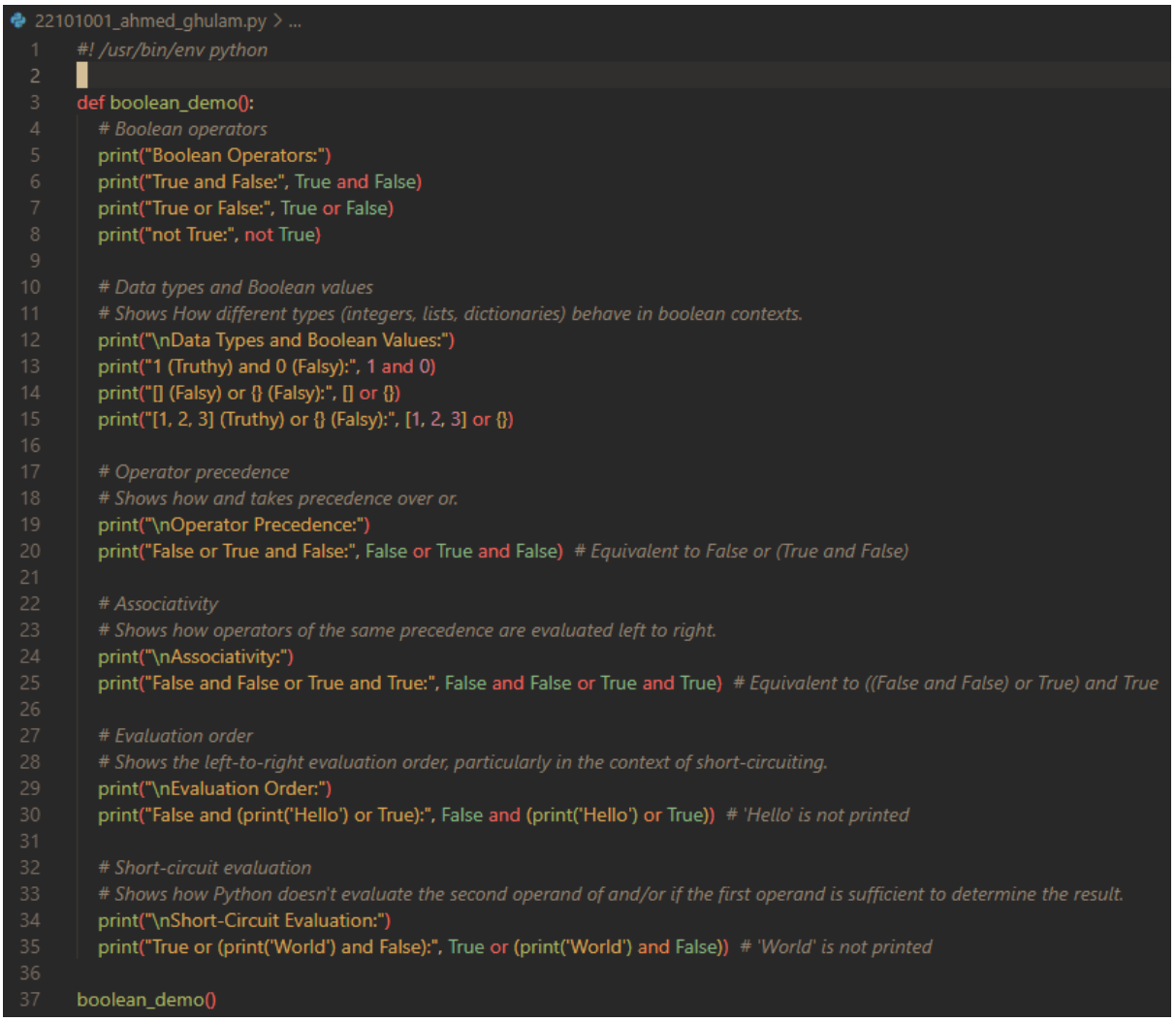
22101001_ahmed_ghulam.lua

```lua
function booleanDemo()
    -- Boolean operators
    print("Boolean Operators:")
    print("true and false:", true and false) -- AND
    print("true or false:", true or false)     -- OR
    print("not true:", not true)               -- NOT

    -- Data types and Boolean values
    -- This part demonstrates how Lua handles different values in boolean contexts.
    print("\nData Types and Boolean Values:")
    print("1 (Truthy) and 0 (Falsy):", 1 and 0) -- In Lua, all values except false and nil are truthy
    print("'string' (Truthy) or '' (Falsy):", 'string' or '') -- Strings are always truthy

    -- Operator precedence
    -- Shows how and takes precedence over or
    print("\nOperator Precedence:")
    print("false or true and false:", false or true and false) -- Equivalent to false or (true and false)

    -- Associativity
    -- Shows that and & or are left-associative.
    print("\nAssociativity:")
    print("true and false or true and true:", true and false or true and true) -- Equivalent to ((true and false) or true) and true

    -- Evaluation order
    -- Shows left-to-right evaluation, with the sideEffect() function not being called due to short-circuiting.
    print("\nEvaluation Order:")
    print("false and sideEffect() or true:", false and sideEffect() or true) -- sideEffect is not called due to short-circuit

    -- Short-circuit evaluation
    -- Shows how Lua does not evaluate the second operand of and or or if the first operand is sufficient to determine the result.

    print("\nShort-Circuit Evaluation:")
    print("true or sideEffect() and false:", true or sideEffect() and false) -- sideEffect is not called due to short-circuit
end

function sideEffect()
    print("Side effect function called")
    return true
end

booleanDemo()
```

5. Python

Boolean Operators: Python uses **`and`**, **`or`**, **`not`** for logical operations.
Data Types and Boolean Values: Dynamically typed. Truthy values include non-zero numbers, non-empty collections, etc., and falsy values include **`0`**, **`None`**, **`""`**, empty collections, **`False`**.
Operator Precedence: **`not`** > **`and`** > **`or`**.
Associativity: All operators are left-associative.
Evaluation Order: Left-to-right.
Short-Circuit Evaluation: Short-circuit implemented for **`and`** & **`or`**.

Here is the source code explaining the design features above:



```python
#! /usr/bin/env python

def boolean_demo():
    # Boolean operators
    print("Boolean Operators:")
    print("True and False:", True and False)
    print("True or False:", True or False)
    print("not True:", not True)

    # Data types and Boolean values
    # Shows How different types (integers, lists, dictionaries) behave in boolean contexts.
    print("\nData Types and Boolean Values:")
    print("1 (Truthy) and 0 (Falsy):", 1 and 0)
    print("[] (Falsy) or {} (Falsy):", [] or {})
    print("[1, 2, 3] (Truthy) or {} (Falsy):", [1, 2, 3] or {})

    # Operator precedence
    # Shows how and takes precedence over or.
    print("\nOperator Precedence:")
    print("False or True and False:", False or True and False)  # Equivalent to False or (True and False)

    # Associativity
    # Shows how operators of the same precedence are evaluated left to right.
    print("\nAssociativity:")
    print("False and False or True and True:", False and False or True and True)  # Equivalent to ((False and False) or True) and True

    # Evaluation order
    # Shows the left-to-right evaluation order, particularly in the context of short-circuiting.
    print("\nEvaluation Order:")
    print("False and (print('Hello') or True):", False and (print('Hello') or True))  # 'Hello' is not printed

    # Short-circuit evaluation
    # Shows how Python doesn't evaluate the second operand of and/or if the first operand is sufficient to determine the result.
    print("\nShort-Circuit Evaluation:")
    print("True or (print('World') and False):", True or (print('World') and False))  # 'World' is not printed

boolean_demo()
```

6. Ruby

Boolean Operators: Ruby uses **&&**, ||, **!** for logical AND, OR, and NOT operations.
Data Types and Boolean Values: being dynamically typed, Ruby treats almost all values as truthy except false and nil.
Operator Precedence: **!** > **&&** > ||.
Associativity: All operators are left-associative.
Evaluation Order: Left-to-right.
Short-Circuit Evaluation: Short-circuit implemented for **&&** and ||.

Here is the source code explaining the design features above:

22101001_ahmed_ghulam.rb

```ruby
def boolean_demo
  puts "Boolean Operators:"
  puts "true && false: #{true && false}" # AND
  puts "true || false: #{true || false}"        # OR
  puts "!true: #{!true}"                        # NOT

  # Data types and Boolean values
  puts "\nData Types and Boolean Values:"
  puts "1 (Truthy) && 0 (Falsy): #{1 && 0}" # In Ruby, 0 is truthy
  puts "'string' (Truthy) || '' (Falsy): #{'string' || ''}" # Non-empty string is truthy, empty string is also truthy

  # Operator precedence
  # Shows how && has higher precedence than ||.
  puts "\nOperator Precedence:"
  puts "false || true && false: #{false || true && false}" # Equivalent to false || (true && false)

  # Associativity
  # Shows that both && and || are left-associative.
  puts "\nAssociativity:"
  puts "true && false || true && true: #{true && false || true && true}" # Equivalent to ((true && false) || true) && true

  # Evaluation order
  # Shows left-to-right evaluation, with the side_effect() method not being called due to short-circuiting in logical expressions.
  puts "\nEvaluation Order:"
  puts "false && side_effect() || true: #{false && side_effect() || true}" # side_effect is not called due to short-circuit

  # Short-circuit evaluation
  # Shows how Ruby doesn't evaluate the second operand of && or || if the first operand is sufficient to determine the result.
  puts "\nShort-Circuit Evaluation:"
  puts "true || side_effect() && false: #{true || side_effect() && false}" # side_effect is not called due to short-circuit
end

def side_effect
  puts "Side effect function called"
  return true
end

boolean_demo
```

7. Rust

Boolean Operators: Rust uses **&&**, ||,**!** for logical AND, OR, and NOT operations.
Data Types and Boolean Values: Rust uses the bool type for boolean expressions with **`true`** & **`false`** operators. Non-boolean values do not automatically convert to boolean types; they require explicit conversion.
Operator Precedence: **!** > **&&** > ||.
Associativity: All operators are left-associative.
Evaluation Order: Left-to-right.
Short-Circuit Evaluation: Short-circuit implemented for **&&** and ||.

Here is the source code explaining the design features above:

22101001_ahmed_ghulam.rs

```rust
fn main() {
    println!("Boolean Operators:");
    println!("true && false: {}", true && false); // AND
    println!("true || false: {}", true || false);       // OR
    println!("!true: {}", !true);                       // NOT

    // Data types and Boolean values
    println!("\nData Types and Boolean Values:");
    // In Rust, only booleans are used in boolean expressions; other types need explicit conversion
    let truthy_value = 1; // Non-zero integers are not automatically considered truthy in Rust
    let falsy_value = 0;
    println!("Non-boolean values (1, 0): {}, {}", truthy_value, falsy_value);

    // Operator precedence
    // Shows how && has higher precedence than ||.
    println!("\nOperator Precedence:");
    println!("false || true && false: {}", false || true && false); // Equivalent to false || (true && false)

    // Associativity
    // Shows that both && and || are left-associative.
    println!("\nAssociativity:");
    println!("true && false || true && true: {}", true && false || true && true); // Equivalent to ((true && false) || true) && true

    // Evaluation order
    // Highlights left-to-right evaluation, with the side_effect() function not being called due to short-circuiting in logical expressions.
    println!("\nEvaluation Order:");
    println!("false && side_effect() || true: {}", false && side_effect() || true); // side_effect is not called due to short-circuit

    // Short-circuit evaluation
    // Shows Rust doesn't evaluate the second operand of && or || if the first operand is sufficient to determine the result.
    println!("\nShort-Circuit Evaluation:");
    println!("true || side_effect() && false: {}", true || side_effect() && false); // side_effect is not called due to short-circuit
}

fn side_effect() -> bool {
    println!("Side effect function called");
    true
}
```

Evaluation

Readability and Writability of List Operations

1. Dart:

Readability: Dart's syntax for list operations is straightforward and similar to other C-style languages, making switching between languages easier.
Writability: Dart supports spread operators and collection `if` and `for`, which add to its writability for complex list operations. It has comprehensive support for list manipulation with methods like `map`, `filter`, and `forEach`, which enhances writability.

2. Go:

Readability: Go's approach to lists (slices) is straightforward, but its lack of built-in high-level functions (like `map`, `filter`) can make some operations less readable. The Go community emphasizes simplicity and readability, often leading to straightforward but lengthy list operations.
Writability: Writing list operations can be more lengthy due to the need for manual implementation of common high-level operations.

3. JavaScript:

Readability: JavaScript's dynamic nature has pros and cons, it's easy to start with but can lead to complex, hard-to-understand code. Its array methods are expressive and widely used, making list operations readable.
Writability: High-level functions like `map`, `filter`, `reduce`, and array destructuring enhance writability.

4. Lua:

Readability: Lua's tables (used as lists) are flexible, but this flexibility can sometimes lead to less predictable patterns, affecting readability. Lua's simplicity makes learning easy, but its different approach (using tables) can initially confuse those from traditional list-based languages.
Writability: Lua's simplicity offers easy writability for basic operations, but more complex manipulations can be less straightforward.

5. Python:

Readability: Python is renowned for its readability, list comprehensions, and built-in functions that make list operations highly readable. Its syntax is designed to be intuitive, making it a go-to language for beginners.

Writability: Python's syntax and built-in methods like `map` and list comprehensions make list operations concise and easy to write. Python's iterators and generators provide additional tools for efficient and readable list processing, especially for large datasets.

6. Ruby:

Readability: Ruby's syntax is designed for readability. List operations using blocks and iterators are very easy to understand. Ruby's mix-ins and duck typing make handling lists and enumerable collections flexible.
Writability: Ruby excels in writability due to its elegant syntax and powerful enumerable methods.

7. Rust:

Readability: Rust's learning curve is steep due to its unique ownership model and borrowing rules, directly impacting how lists are managed. Rust's approach to listing operations is straightforward, though its strictness around ownership and borrowing can complicate the readability for beginners.
Writability: Rust offers comprehensive iterator methods, but the strictness can make writing more complex.

Best Language for Boolean Expressions

In my opinion, Python stands out in terms of simplicity and readability in Boolean expressions, especially for those new to programming or coming from a non-technical background. Its syntax is intuitive, and the language's philosophy emphasizes readability, a crucial aspect of Boolean logic. However, for projects requiring more strict type safety and performance, languages like Rust or Go might be more suitable.

Learning strategy

I began by exploring Dart Documentation, noting its similarity to other C-style languages. Moving to Go, I consulted The Go Programming Language Specification and learned about the language's straightforward Boolean logic. I turned to MDN Web Docs for JavaScript and learned the language's Boolean expressions. I saw Lua's approach to handling Boolean expressions by reviewing the Lua Reference Manual. I was already familiar with Python, but I consulted its documentation to fill the gaps in my knowledge. For Ruby, I used its official documentation. Finally, Rust was studied through rust-lang.org's documentation.

Throughout this time, I employed a strategic learning approach. I started with the official documentation to get a fundamental grasp of each language, followed by additional resources if the documentation was not enough.

Here are the URLs for the online compilers/interpreters I used to compile the above source codes:

1. Dart: https://dartpad.dev/
2. Go: https://go.dev/play/
3. Js: https://playcode.io/new
4. Lua: https://www.tutorialspoint.com/execute_lua_online.php
5. Python: https://www.programiz.com/python-programming/online-compiler/
6. Ruby: https://onecompiler.com/ruby/
7. Rust: https://play.rust-lang.org/