CS 315 - Programming Languages
Homework 2

Ghulam Ahmed 22101001

Closures in Dart, Go, Javascript, Lua, Python, Ruby, and Rust

Design issues in each language

1. Dart

In Dart, closures are anonymous functions that capture and use variables from their lexical scope. The syntax for creating closures is implicit and doesn't require any special keywords. They can be defined using arrow (=>) syntax for single expressions or curly braces ({}) for more complex logic.

Variables from the outer scope are accessible within the closure. In the code below, `simpleClosure` accesses the `message` variable, `incrementClosure` accesses and modifies the `counter` variable, and `complexClosure` captures `part1` and `part2`. This accessibility demonstrates that closures in Dart can capture the environment in which they are created.

The mutability of captured variables depends on their declaration in the outer scope. In Dart, if a variable is mutable in the outer scope (like `counter`), it remains mutable within the closure. This is evident as `incrementClosure` modifies `counter` each time it is called.

Nested closures are also allowed and can be helpful. The `createMultiplier` function demonstrates this by returning a closure containing another function (`innerFunction`). The inner closure has access to its own local variables as well as variables from its parent closure's scope.

Additionally, Dart closures capture variables by reference, not by value. This means changes to a captured variable are reflected inside the closure. Furthermore, Dart supports both local and global variables capturing, enhancing the flexibility and power of closures in various contexts.

Closures in Dart are a powerful construct allowing for functional programming patterns, encapsulation, and creating factory functions. They are extensively used in event handling, callbacks, and functional manipulations of data structures.

Here is the source code explaining the design features above:

```
void main() {
  // Demonstrating basic closure
  String message = 'Hello';
  var simpleClosure = () => print(message);
  simpleClosure(); // Output: Hello

  // Demonstrating closure with mutability
  int counter = 0;
  var incrementClosure = () {
    counter += 1;
    print('Counter: $counter');
  };
  incrementClosure(); // Output: Counter: 1
  incrementClosure(); // Output: Counter: 2

  // Demonstrating closure with a returned function
  var adderClosure = createAdder(5);
  print(adderClosure(3)); // Output: 8

  // Demonstrating nested closures
  var nestedClosure = createMultiplier(2);
  print(nestedClosure(5)); // Output: 10

  // Demonstrating closure capturing multiple variables
  var complexClosure = createComplexClosure('Complex', 'Closure');
  complexClosure(); // Output: Complex Closure
}

// Function that returns a closure adding a specific number
Function(int) createAdder(int addBy) {
  return (int number) => number + addBy;
}

// Function that returns a nested closure
Function(int) createMultiplier(int multiplyBy) {
  return (int number) {
    int innerFunction(int innerNumber) {
      return innerNumber * multiplyBy;
    }
    return innerFunction(number);
  };
}

// Function that creates a closure capturing multiple variables
Function() createComplexClosure(String part1, String part2) {
  return () => print('$part1 $part2');
}
```

2. Go

In Go, closures are created using anonymous function syntax, which doesn't require any special keywords. They are defined either inline, as seen with `simpleClosure` and `incrementClosure`, or as return values from other functions, like `createAdder`, `createMultiplier`, and `createComplexClosure`. Closure capturing multiple variables in Go is demonstrated by `createComplexClosure, which` creates a closure that captures two variables, `part1,` and `part2`, from its outer scope.

Variables from the outer scope are accessible within the closure. For instance, `simpleClosure` captures the `message` variable from its outer scope, and `incrementClosure` accesses and modifies the `counter` variable. This demonstrates Go's ability to capture the environment in which closures are created.

The captured variables are mutable if they are mutable in the outer scope. This is evident in how `incrementClosure` modifies the `counter` variable. In Go, when a closure modifies a variable from its outer scope, it's modifying the same variable, not a copy. This is because closures in Go capture variables by reference.

Nested closures are allowed in Go, as shown by the `createMultiplier` function. In this example, the returned closure captures the `multiplyBy` variable from its outer scope.

Additionally, Go allows closures to capture multiple variables from their outer scope, as demonstrated by `complexClosure`. This feature helps maintain state across multiple function calls or create function factories.

Closures in Go can be used for implementing function literals, encapsulating state, and in scenarios where you need to pass around a piece of functionality with its own state. They are also useful in Go's concurrent programming patterns, such as goroutines and channels, where they can be used to encapsulate the state and logic for each concurrent task.

Here is the source code explaining the design features above:

```go
package main

import "fmt"

func main() {
    // Demonstrating basic closure
    message := "Hello"
    simpleClosure := func() {
        fmt.Println(message) // Captures 'message' from outer scope
    }
    simpleClosure() // Output: Hello

    // Demonstrating closure with mutability
    counter := 0
    incrementClosure := func() {
        counter++ // Mutates 'counter' variable
        fmt.Println("Counter:", counter)
    }
    incrementClosure() // Output: Counter: 1
    incrementClosure() // Output: Counter: 2

    // Demonstrating closure that returns a function
    adderClosure := createAdder(5)
    fmt.Println(adderClosure(3)) // Output: 8

    // Demonstrating nested closures
    nestedClosure := createMultiplier(2)
    fmt.Println(nestedClosure(5)) // Output: 10

    // Demonstrating closure capturing multiple variables
    complexClosure := createComplexClosure("Complex", "closure")
    complexClosure() // Output:  Complex closure
}

// Function that returns a closure adding a specific number
func createAdder(addBy int) func(int) int {
    return func(number int) int {
        return number + addBy
    }
}

// Function that returns a nested closure
func createMultiplier(multiplyBy int) func(int) int {
    return func(number int) int {
        return number * multiplyBy
    }
}

// Function that creates a closure capturing multiple variables
func createComplexClosure(part1 string, part2 string) func() {
    return func() {
        fmt.Println(part1, part2)
    }
}
```

3. JavaScript

In JavaScript, closures are created whenever a function is declared. This is apparent in the `simpleClosure`, which captures the `message` variable from its outer scope. The syntax for creating closures is the same as for defining any function. Arrow functions, as used for `simpleClosure`, are a common way to create closures due to their concise syntax.

Variables from the outer scope are accessible within the closure. This is demonstrated in `incrementClosure`, which accesses and modifies the `counter` variable. JavaScript closures capture variables by reference, so changes in the outer scope are reflected within the closure and vice versa.

Regarding mutability, captured variables in JavaScript closures follow the same mutability rules as in their original scope. If a variable is mutable in its original scope, like `counter`, it can be modified within the closure.

Nested closures are also supported in JavaScript. This is seen in `createMultiplier`, where a function returns another function, each having access to their respective lexical environments. This nesting can be as deep or complex as needed.

The ability to capture multiple variables is another feature of JavaScript closures, illustrated by `complexClosure`. This function captures `part1` and `part2` from its lexical scope and uses them when invoked. Closure returning a function in JavaScript is demonstrated by `createAdder`, which returns a closure that adds a given number to its argument.

Closures in JavaScript are not only a technical feature but a core part of the language's features, enabling powerful programming patterns. They can be used for encapsulating data (thus providing privacy), in callbacks and higher-order functions, and in event handling, particularly in asynchronous programming patterns like Promises and async/await.

Here is the source code explaining the design features above:

```
function main() {
    // Demonstrating basic closure
    let message = "Greetings";
    let simpleClosure = () => console.log(message);
    simpleClosure(); // Output: Greetings

    // Demonstrating closure with mutability
    let counter = 10;
    let incrementClosure = () => {
        counter += 5;
        console.log('Counter:', counter);
    };
    incrementClosure(); // Output: Counter: 15
    incrementClosure(); // Output: Counter: 20

    // Demonstrating closure that returns a function
    let adderClosure = createAdder(7);
    console.log(adderClosure(4)); // Output: 11

    // Demonstrating nested closures
    let nestedClosure = createMultiplier(3);
    console.log(nestedClosure(4)); // Output: 12

    // Demonstrating closure capturing multiple variables
    let complexClosure = createComplexClosure("Complex", "Closure");
    complexClosure(); // Output:  Complex Closure
}

// Function that returns a closure adding a specific number
function createAdder(addBy) {
    return function (number) {
        return number + addBy;
    };
}

// Function that returns a nested closure
function createMultiplier(multiplyBy) {
    return function (number) {
        return number * multiplyBy;
    };
}
```

```
// Function that creates a closure capturing multiple variables
function createComplexClosure(part1, part2) {
    return function () {
        console.log(part1, part2);
    };
}

main();
```

4. Lua

Closures in Lua are functions that carry with them the environment in which they were declared, allowing them to access local variables from that scope even after the outer function has finished executing.

In Lua, closures are created in the same way as regular functions, using the `function` keyword. This is evident in the examples `basicClosure`, `mutableClosure`, etc. The closure is essentially the anonymous function returned by these functions.

Variables from the outer scope are accessible within the closure, as shown by `basicClosure`, where the anonymous function returned can access the `message` variable. This accessibility demonstrates the concept of lexical scoping, where functions remember the environment in which they were created.

Concerning mutability, Lua closures capture local variables by reference, meaning that if a variable is mutable in the outer scope, like the `counter` in `mutableClosure`, it remains mutable within the closure. Thus, each call to `counterClosure` modifies the same `counter` variable.

Nested closures are also possible in Lua, as demonstrated by `createMultiplier`, capturing the `multiplyBy` variable. In this example, the returned function itself contains another function (`innerFunction`), showing the ability of Lua closures to capture and maintain multiple levels of scope.

Capturing multiple variables from the outer scope is another characteristic of Lua closures, as createComplexClosure shows. The closure created captures both `part1` and `part2` from its lexical environment. Closure returning a function is demonstrated by `createAdder`, which returns a closure that adds a specific number to its argument.

Closures in Lua are a fundamental part of the language's functional programming capabilities. They are commonly used in callback functions, encapsulating state within a function, in iterator functions, and in implementing object-oriented patterns.

Here is the source code explaining the design features above:

```lua
function basicClosure()
    local message = "Hello"
    return function()
        print(message)
    end
end

-- Demonstrating closure with mutability
function mutableClosure()
    local counter = 5
    return function()
        counter = counter + 2
        print("Counter:", counter)
    end
end

-- Demonstrating closure that returns a function
function createAdder(addBy)
    return function(number)
        return number + addBy
    end
end

-- Demonstrating nested closures
function createMultiplier(multiplyBy)
    return function(number)
        local function innerFunction(innerNumber)
            return innerNumber * multiplyBy
        end
        return innerFunction(number)
    end
end

-- Demonstrating closure capturing multiple variables
function createComplexClosure(part1, part2)
    return function()
        print(part1, part2)
    end
end

-- Main function to run the closures
function main()
    local simpleClosure = basicClosure()
    simpleClosure() -- Output: Hello

    local counterClosure = mutableClosure()
    counterClosure() -- Output: Counter: 7
    counterClosure() -- Output: Counter: 9

    local adderClosure = createAdder(10)
    print(adderClosure(5)) -- Output: 15

    local nestedClosure = createMultiplier(4)
    print(nestedClosure(3)) -- Output: 12

    local complexClosure = createComplexClosure("Complex", "closure")
    complexClosure() -- Output: Complex closure
end

main()
```

5. Python

Closures in Python are functions that remember the values from their enclosing lexical scope even when the program flow is no longer in that scope.

Closures in Python can be defined using nested functions or lambda expressions. The inner function (the closure) captures the variables from the enclosing (outer) function. This is evident in functions like `basic_closure`, `mutable_closure`, and `create_adder`.

Variables from the outer function's scope are accessible within the closure. For instance, the `closure` function inside `basic_closure` accesses the `message` variable defined in its outer scope. This access is a crucial feature of closures, allowing the inner function to remember the state of its environment when it was created.

Regarding mutability, Python closures can modify variables from the outer scope if declared `nonlocal`. This keyword is used in `mutable_closure`, `create_counter_closure`, and `create_accumulator` to modify the `counter`, `count`, and `total` variables, respectively, within the closure. Without `nonlocal`, these variables would be treated as local to the closure, and attempts to modify them would result in an error.

Nested closures are supported in Python, as demonstrated by `create_multiplier`, where a function returns another function, each capturing and using variables from their respective outer scopes.

Python closures, whether created using nested functions or lambda expressions, can also capture multiple variables from their lexical scope, as shown in `create_complex_closure`. This capability is beneficial for maintaining states and creating higher-order functions. To demonstrate closure returning a function, `create_adder` returns a closure that adds a specified number.

Closures in Python are extensively used in functional programming, callback mechanisms, decorators, and event handling. They allow for a functional programming style and help create cleaner, more modular, and maintainable code.

Here is the source code explaining the design features above:

```
def basic_closure():
    message = "Hello"
    def closure():
        print(message)
```

```python
    return closure

# Lambda version of the above function
basic_lambda_closure = lambda message="Hello": lambda: print(message)

def mutable_closure():
    counter = 3
    def closure():
        nonlocal counter
        counter += 4
        print("Counter:", counter)
    return closure

def create_adder(add_by):
    def adder(number):
        return number + add_by
    return adder

def create_multiplier(multiply_by):
    def multiplier(number):
        def inner_multiplier(inner_number):
            return inner_number * multiply_by
        return inner_multiplier(number)
    return multiplier

def create_complex_closure(part1, part2):
    def closure():
        print(part1, part2)
    return closure

# Lambda version of the above function
create_complex_closure_lambda = lambda part1, part2: lambda: print(part1, part2)

def create_counter_closure():
    count = 0
    def closure():
        nonlocal count
        count += 1
        return count
    return closure

def create_accumulator(initial_value):
    total = initial_value
    def accumulator(value):
        nonlocal total
        total += value
        return total
    return accumulator

def main():
    simple_closure = basic_closure()
    simple_closure() # Output: Hello

    simple_lambda_closure = basic_lambda_closure()
    simple_lambda_closure() # Output: Hello

    counter_closure = mutable_closure()
    counter_closure() # Output: Counter: 7
    counter_closure() # Output: Counter: 11

    adder_closure = create_adder(20)
    print(adder_closure(6)) # Output: 26

    multiplier_closure = create_multiplier(5)
    print(multiplier_closure(4)) # Output: 20

    complex_closure = create_complex_closure("Complex", "closure")
    complex_closure() # Output: Complex closure

    complex_closure_lambda = create_complex_closure_lambda("Complex", "closure")
    complex_closure_lambda() # Output: Complex closure

    count_closure = create_counter_closure()
    print(count_closure()) # Output: 1
    print(count_closure()) # Output: 2

    accumulator_closure = create_accumulator(50)
    print(accumulator_closure(5))  # Output: 55
    print(accumulator_closure(10)) # Output: 65

if __name__ == "__main__":
    main()
```

6. Ruby

In Ruby, closures are blocks of code that can be passed around and executed. They can capture variables from the context in which they were defined.

In Ruby, closures are commonly created using `Proc` objects or `lambda`. This is seen in the functions `basic_closure`, `mutable_closure`, and `create_adder`. The `Proc.new` and `lambda` methods are used to define these closures, encapsulating the code block and the environment in which they are defined.

Variables from the outer scope are accessible within the closure. For example, in `basic_closure`, the `Proc` object returned can access the `message` variable from its defining context. This capability allows closures to maintain a state.

Regarding mutability, Ruby closures can modify variables from their outer scope. This is demonstrated in `mutable_closure`, where the `Proc` object modifies the `counter` variable each time it is called. It's important to note that Ruby closures capture variables by reference, meaning changes to a captured variable are reflected inside the closure.

Nested closures are also supported in Ruby, as shown in `create_multiplier`. In this function, a lambda returns another lambda, each capturing and working with variables from their respective scopes.

Capturing multiple variables from the outer scope is another feature of Ruby closures, as illustrated in `create_complex_closure`. This function creates a `Proc` object that captures both `part1` and `part2`.

Closures in Ruby are a powerful and versatile tool. They are used in various contexts, such as in iterators, for passing around blocks of code, in lazy evaluation, and in implementing callbacks and event handlers. Ruby's implementation of closures is a key part of its support for functional programming paradigms and adds to the language's expressiveness and flexibility.

Here is the source code explaining the design features above:

```
def basic_closure
    message = "Hello"
    Proc.new { puts message }
  end

  def mutable_closure
    counter = 1
    Proc.new do
      counter += 2
      puts "Counter: #{counter}"
    end
  end

  def create_adder(add_by)
    lambda { |number| number + add_by }
  end

  def create_multiplier(multiply_by)
    lambda do |number|
      inner_multiplier = lambda { |inner_number| inner_number * multiply_by }
      inner_multiplier.call(number)
    end
  end

  def create_complex_closure(part1, part2)
    Proc.new { puts "#{part1} #{part2}" }
  end

  def create_counter_closure
    count = 0
    Proc.new do
      count += 1
      count
    end
  end

  def create_accumulator(initial_value)
    total = initial_value
    lambda do |value|
      total += value
      total
    end
```

```
end

def main
  simple_closure = basic_closure
  simple_closure.call # Output: Hello

  counter_closure = mutable_closure
  counter_closure.call # Output: Counter: 3
  counter_closure.call # Output: Counter: 5

  adder_closure = create_adder(15)
  puts adder_closure.call(7) # Output: 22

  multiplier_closure = create_multiplier(6)
  puts multiplier_closure.call(3) # Output: 18

  complex_closure = create_complex_closure("Complex", "closure")
  complex_closure.call # Output:  Complex closure

  counter = create_counter_closure
  puts counter.call # Output: 1
  puts counter.call # Output: 2

  accumulator = create_accumulator(40)
  puts accumulator.call(5) # Output: 45
  puts accumulator.call(10) # Output: 55
end

main if __FILE__ == $PROGRAM_NAME
```

7. Rust

Closures in Rust are anonymous functions that can capture variables from their surrounding environment.

In Rust, closures are defined using the `||` syntax. They can capture variables from their enclosing scope in three ways: by borrowing immutably (&T), borrowing mutably (&`mut T`), or taking ownership (T). The compiler infers how to capture each variable based on its use in the closure.

The basic closure in the example captures `message` by immutably borrowing it. This is because the closure only needs to read the value of `message`. In Rust, this kind of borrowing ensures that the closure does not alter the captured variable.

For mutable closures, as seen with `mutable_closure`, Rust allows the closure to mutate the captured variable. This is achieved by declaring `counter` as mutable (`mut`) in the outer scope and then capturing it mutably in the closure. This kind of capturing is powerful but must be used with caution to avoid data races in concurrent contexts.

The closure that returns a function, like `create_adder` and `create_multiplier`, demonstrates how closures can be used to create higher-order functions. These closures capture their parameters (`add_by` and `multiply_by`) and return a new closure that uses these captured values.

Nested closures are supported in Rust and can capture variables from the outer closure. This is shown in `create_multiplier`, which captures `multiply_by` and uses it in the returned closure.

The `create_complex_closure` function demonstrates capturing multiple variables (`part1` and `part2`). Rust closures can capture as many variables as needed, either by borrowing or by taking ownership, depending on the use case.

The `create_counter_closure` and `create_accumulator` functions use `Box<dyn FnMut()>` and `Box<dyn FnMut(i32) -> i32>`, respectively, to return closures. This is because closures in Rust are represented by traits (`Fn`, `FnMut`, and `FnOnce`), and each closure has its unique type. Using `Box<dyn Fn...>` allows for returning closures of different types from a function.

In summary, closures in Rust are versatile and powerful, allowing for capturing and manipulating the environment in which they are defined. They are used extensively in Rust for tasks like iterating over collections, handling events, and designing APIs that require custom logic. The ability of Rust closures to capture variables in different ways (by value, mutable borrow, or immutable borrow) offers excellent flexibility while aligning with Rust's focus on safety and concurrency.

Here is the source code explaining the design features above:

```
fn main() {
    // Basic closure
    let message = String::from("Hello");
    let basic_closure = || println!("{}", message);
    basic_closure(); // Output: Hello

    // Mutable closure
    let mut counter = 2;
    let mut mutable_closure = || {
        counter += 3; // Mutates counter
        println!("Counter: {}", counter);
    };
    mutable_closure(); // Output: Counter: 5
    mutable_closure(); // Output: Counter: 8

    // Closure returning a function
    let adder_closure = create_adder(10);
    println!("Result: {}", adder_closure(4)); // Output: Result: 14

    // Nested closures
    let multiplier_closure = create_multiplier(3);
```

```rust
    println!("Result: {}", multiplier_closure(6)); // Output: Result: 18

    // Closure capturing multiple variables
    let complex_closure = create_complex_closure("Complex", "closure");
    complex_closure(); // Output: Complex closure

    // Counter closure
    let mut counter_closure = create_counter_closure();
    println!("Count: {}", counter_closure()); // Output: Count: 1
    println!("Count: {}", counter_closure()); // Output: Count: 2

    // Accumulator closure
    let mut accumulator_closure = create_accumulator(30);
    println!("Total: {}", accumulator_closure(7)); // Output: Total: 37
    println!("Total: {}", accumulator_closure(8)); // Output: Total: 45
}

fn create_adder(add_by: i32) -> Box<dyn Fn(i32) -> i32> {
    Box::new(move |number| number + add_by)
}

fn create_multiplier(multiply_by: i32) -> Box<dyn Fn(i32) -> i32> {
    Box::new(move |number| number * multiply_by)
}

fn create_complex_closure<'a>(part1: &'a str, part2: &'a str) -> Box<dyn Fn() +
'a> {
    Box::new(move || println!("{} {}", part1, part2))
}

fn create_counter_closure() -> Box<dyn FnMut() -> i32> {
    let mut count = 0;
    Box::new(move || {
        count += 1;
        count
    })
}

fn create_accumulator(start: i32) -> Box<dyn FnMut(i32) -> i32> {
    let mut total = start;
    Box::new(move |value| {
        total += value;
        total
    })
}
```

Evaluation

Readability and Writability of the syntax for closures

1. Dart:

Readability: Dart's syntax for list operations is straightforward and similar to other C-style languages, making switching between languages easier. This familiarity eases readability, especially for those accustomed to similar syntax in other languages. Arrow functions and the ability to capture variables from the outer scope enhance the clarity of closures.

Writability: Dart offers great writability for closures, supporting both concise arrow functions for single expressions and traditional braces for complex logic. Its flexibility in capturing variables from the outer scope, including mutable variables, and the ability to create nested closures add to its writability.

2. Go:

Readability: Go's syntax for closures is straightforward, aligning with the language's overall emphasis on simplicity. Closures capture variables from their surrounding scope, but the lack of high-level functional constructs can sometimes make complex closure operations less readable.
Writability: While Go supports closures, writing them can be more verbose compared to languages with more functional features. The manual process of capturing variables and the absence of shorthand syntax for simple closures make it less concise.

3. JavaScript:

Readability: JavaScript's closure syntax is highly readable, thanks to its dynamic nature and the use of arrow functions. Closures in JavaScript are a fundamental part, and their use in callbacks and higher-order functions contributes to the language's expressiveness.

Writability: JavaScript's support for arrow functions and the ability to capture any variable from the outer scope effortlessly enhance writability. The language's flexible and dynamic nature allows for concise and powerful closure expressions.

4. Lua:

Readability: Lua's closure syntax is simple and readable, but its unique approach of using tables for everything, including functions and closures, might confuse those familiar with more traditional languages.

Writability: Lua closures are easy to write for basic operations. However, the language's unique characteristics and reliance on tables for closures might make more complex closures slightly less straightforward to implement.

5. Python:

Readability: Python's syntax for closures, using nested functions, is clear and readable, aligning with the language's overall philosophy of simplicity and readability. The ability to capture variables from the outer scope is straightforward and adds to its readability.

Writability: Python excels in writability for closures, with its clean syntax and the use of the `nonlocal` keyword to modify outer scope variables. Its support for nested functions and the ability to capture multiple variables from the outer scope make closures in Python both powerful and easy to write.

6. Ruby:

Readability: Ruby's closures, typically created with blocks, Procs, or lambdas, are very readable. The language's emphasis on simplicity and elegance extends to its closure syntax, making it easy to understand and follow.
Writability: Ruby's closures are not only readable but also writable, thanks to its expressive syntax and robust methods. The language's flexible approach to closures and powerful enumerable methods enhance its writability.

7. Rust:

Readability: Rust's syntax for closures can be initially challenging due to the language's focus on safety and performance. However, once accustomed, closures in Rust are quite readable, with a clear syntax that expresses their functionality and captures variables from their environment effectively.
Writability: Rust offers robust support for closures, including mutable closures and the ability to capture variables by value or reference. Writing closures in Rust can be more involved due to the language's strict ownership and borrowing rules, but it provides robust functionality, especially in concurrent and functional programming contexts.

Best Language for Closures

In my opinion, JavaScript and Python stand out for their balance of readability, writability, and flexibility in handling closures. JavaScript's lexical scoping and arrow functions are easy to understand, while Python's explicit scope handling via `nonlocal` is precise and predictable.

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 the documentation of rust-lang.org.

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://onecompiler.com/lua/
5. Python: https://www.programiz.com/python-programming/online-compiler/
6. Ruby: https://onecompiler.com/ruby/
7. Rust: https://play.rust-lang.org/