BILKENT UNIVERSITY
FALL 2023 – 2024

CS315
PROGRAMMING LANGUAGES

HOMEWORK 2
CLOSURES

SECTION 003

YASEMİN AKIN
22101782

10 DECEMBER 2023

Table of Contents

PART A – DESIGN ISSUES IN EACH LANGUAGE

1. DART

1.1. What is the syntax for creating closures? Is there any explicit syntax?

```
String Function(int) createAdder(int addBy) {
   return (int i) => 'Result: ${i + addBy}';
}

var addTwo = createAdder(2);
print(addTwo(3));
```

Output: Result: 5

Dart creates closures by means of a function syntax that may or may not be named or anonymous. Instead of defining a syntax just for closures, Dart allows any function to act as a closure. Functions allow for variable capture and modification, coming from their lexical scope. This is an example of createAdder being a function that yields a closure that summates a certain number on its argument. AddBy is taken out of its lexical scope by the returned function. The result shows that closure can access the addBy once createAdders is over.

1.2. What variables from the outer scope are accessible within the closure?

```
String outerFunction(String message) {
  // An outer scope variable
  String outerVariable = 'Outer: $message';

  // Closure definition
  String innerFunction() {
    // Accessing the outer scope variable
    return 'Inner accesses: $outerVariable';
  }

  return innerFunction(); // Executing the closure
}

void main() {
  var result = outerFunction('Hello');
  print(result);
}
```

Output: Inner accesses: Outer: Hello

When a closure is created in Dart, for example, variables from its lexical scope can be accessed. Hence, any variable defined in the same scope as the closure or any outer scope with respect to where the closure is defined may be accessed by the closure. Such include local variables of the function in which the closure was defined and global variables. Such a scenario can be explained by an example of how innerFunction, a closure, refers to outerVariable, a variable originating from outerFunction. A particular message results in the setting of an outer variable when the outer function is called. Closure uses this variable in its scope, thereby showing how Dart makes it possible for a closure to relate and use variables that are outside its immediate lexical scope.

1.3. Are the variables captured from outer scope mutable or immutable?

```
var globalVar = 10;

void main() {
var outerVar = 10;
final outerFinal = 10;
   Function closure = () {
         // outerFinal = 20; // This line would cause an error, as final variables can't be modified.
         print(outerFinal);
         outerVar = 20;
         print(outerVar);
         globalVar = 20;
         print(globalVar);
   };
   closure();
}
```

```
Output:      10
             20
             20
```

The mutability of variables extracted from enclosing scope in closures is dependent upon their declaration in Dart. outerFinal is final and therefore can not be changed in the closure. This will be a compile time error if one tries to alter it. The outerVar variable is mutable. This causes the closure to update its value from 10 to 20 and upon printing the variable. Within closures, global variables such as globalVar are both acceptable and modifiable. This statement closes and updates globalVar to twenty that gets printed.

1.4. Are nested closures allowed?

```
Function outerClosure() {
```

```
  var outerVar = 'Outer';

  Function innerClosure() {
    var innerVar = 'Inner';
    print(outerVar); // Accesses variable from the outer closure
    return () {
      print('$outerVar $innerVar'); // Accesses variables from both scopes
    };
  }

  return innerClosure();
}

void main() {
  var nestedClosure = outerClosure();
  nestedClosure(); // Executes the nested closure
}
```

Output: Outer
Outer Inner

Nested closures are valid in Dart. This Dart code defines the innerClosure function inside the outerClosure function. innerVar is a private variable of the innerClosure, while the outerVar is shared externally by the outerClosure. It is illustrative of Dart's closures capturing and holding references to any variables from containing scopes, in case of nesting. This code will print out the values for both outerVar and innerVar to show that nested closures are indeed functional and can access variables in different scopes.

2. GO

2.1. What is the syntax for creating closures? Is there any explicit syntax?

```
func main() {
    adderFunc := func(x int) func(int) int {
        return func(y int) int {
            return x + y
        }
    }

    addFive := adderFunc(5)
    fmt.Println(addFive(10)) // Output: 15
}
```

Output: 15

Closures are created in Go using anonymous functions (named functions can also be used, but typically, it is preferable to utilize anonymous ones for this purpose in Go). Go allows for the declaration of anonymous functions inside a function with access capture to environment. Here, adderFunc is a function returning a function. This returned function is used to add a particular number to an x value provided when adderFunc was called. The inner function captures the x value from its surrounding environment, a critical attribute of closures. This output "15" confirms that the closure keeps the value of x (which is 5) and includes this into the argument addedFive(10).

2.2. What variables from the outer scope are accessible within the closure?

```
func outerFunction() func() {
   outerVar := 10
   packageLevelVar := &globalVar

   closure := func() {
      fmt.Println("Outer variable:", outerVar)
      fmt.Println("Package-level variable:", *packageLevelVar)
   }

   return closure
}

var globalVar = 100

func main() {
   myClosure := outerFunction()
   myClosure()  // Access the outer and package-level variables
}
```

Output:
Outer variable: 10
Package-level variable: 100

A closure in Go allows the closure to access the variables defined in its surrounding area. These comprise of locally scoped variables of their functions or global package scope variables. In this case, outerVar belongs to outerFunction while globalVar is a package level variable. OuterVar and globalVar are accessed by the closure closure defined inside outerFunction. In main when myClosure it shows that closure has an access to these variables.

2.3. Are the variables captured from outer scope mutable or immutable?

```
func main() {
   outerVar := 10
   closure := func() {
      outerVar += 5 // Modifying the outer variable
      fmt.Println("Inside Closure:", outerVar)
   }

   fmt.Println("Before Closure:", outerVar)
   closure()
   fmt.Println("After Closure:", outerVar)
}
```

Output:
Before Closure: 10
Inside Closure: 15
After Closure: 15

Captured variables in a closure from an outer scope in of a Go are mutable. As a consequence, modifications of captured variables from enclosing scopes are allowed by closures in Go. The following is Go code for closures in the main function that grabs and alters the outerVar variable. For instance, every time the closure function is invoked, it increments outerVar by 5. The result reveals the initial value of outerVar and the new one that is obtained post-call of a closure which proves the change.

2.4. Are nested closures allowed?

```
func outerClosure() func() func() string {
   outerVar := "Outer"

   return func() func() string {
      innerVar := "Inner"
      fmt.Println(outerVar) // Accesses outer closure variable

      return func() string {
         return outerVar + " " + innerVar // Accesses both outer and inner variables
      }
   }
}

func main() {
   innerClosure := outerClosure()()
```

```
    fmt.Println(innerClosure()) // Executes the nested closure
}
```

Output:
Outer
Outer Inner

Nested, or inner, closures are allowed in go. In this case, an inner scope's closure has access to the variables defined within the scope of the external layer or the upper closure in Go language. This is Go code that yields an outer closure that returns another function in turn that returns a closure. The deepest closure can utilize innerVar from its own environment, while referring to outerVar from the outer closure. This code shall show that a nested closure can manipulate the data inside it as well as the outer one.

3. JAVASCRIPT

3.1. What is the syntax for creating closures? Is there any explicit syntax?

```
function outerFunction(outerVariable) {
    function innerFunction(innerVariable) {
        console.log(outerVariable, innerVariable);
    }
    return innerFunction;
}

const newFunction = outerFunction('outside');
newFunction('inside');
```

Output: outside inside

Closures in Java script are commonly formed when one defines a function inside the other function and the internal function can read or change variables of the external function. Closures are not syntactically marked in JavaScript — instead, being just byproducts of the function scoping rule. In this case, innerFunction is a closure that references outerVariable from outerFunction. Whenever, a new function, newFunction, gets invoked with inside, it has an access to both outside (caught from outerFunction) and inside (its own argument). This demonstrates the essence of closures in JavaScript: preserve the reference to their lexical environment.

3.2. What variables from the outer scope are accessible within the closure?

```
var globalVar = "Global";
```

```
function outerFunction2(outerParam) {
  var outerVar = "Outer";

  function innerFunction2() {
    var innerVar = "Inner";
    console.log(globalVar);   // Accesses global variable
    console.log(outerParam);  // Accesses parameter of the outer function
    console.log(outerVar);    // Accesses variable from the outer function
    console.log(innerVar);    // Accesses variable from its own scope
  }

  innerFunction2();
}

outerFunction2("OuterParam");
```

Output:
Global
OuterParam
Outer
Inner

A closeure can access global variables available in the outer scope or local variables present in the surrounding function or any parameter provided to this surrounding function in Javascript. InnerFunction is a closure used here that gets value of globalVar, outerParam, and outerVar from outerFunctions. Furthermore, it intercepts outerVar, a variable that was similarly declared in its own scope.

3.3. Are the variables captured from outer scope mutable or immutable?

```
function createCounter() {
  let count = 0;

  return {
    increment: function() {
      count++;
      console.log(count);
    },
    decrement: function() {
      count--;
      console.log(count);
    }
  };
```

```
}

let counter = createCounter();
counter.increment();
counter.decrement();
```

Output:
1
0

Closure captures variables from the outer scope in JavaScript and these variables are mutable that is they can be modified by the closure. CreateCounter in this JavaScript example gives you an object with two methods, one is increment while the other is decrement. These techniques alter the value of the count variable that is declared within the outer scope of createCounter. Such changes illustrate how captured variables in JavaScript's closures are mutable, as count is not a local variable.

3.4. Are nested closures allowed?

```
function outerFunction3(outerVar) {
   return function innerFunction3(innerVar) {
      return function innermostFunction() {
         console.log('Outer Variable:', outerVar);
         console.log('Inner Variable:', innerVar);
      };
   };
}

const newFunction2 = outerFunction3('outer')( 'inner');
newFunction2();
```

Output:
Outer Variable: outer
Inner Variable: inner

Nested closures are allowed in JavaScript. Nested Closure is a function defined in another closure with an access to variables in own scope, in scope of outer function and in global scope in JavaScript. In this case, innermostFunction is nested inside innerFunction that has been nested in outerFunction. This permits innermostFunction to utilize both innerVar from innerFunction and outerVar from outerFunction.0 This output shows that nested closures retain access to variable of multiples enclosing scopes.

4. LUA

4.1. What is the syntax for creating closures? Is there any explicit syntax?

```
function outerFunction(x)
   return function(y)
      return x + y
   end
end

local addFive = outerFunction(5)
print(addFive(3))
```

Output: 8

Closures in Lua are defined through anonymous functions (although it is possible to define them using named ones; however, one typically opts for the anonymous option). This Lua example, outerFunction returns an anonymous function that is part of a closure. The internal operation retrieves x from external scope. Invoking of outer function with five yields a fresh function that adds five on its argument. this shows Luas's closures syntax in which functions can incorporate and apply variables within its surrounding scope. When this code is run the result is a value of eight which indicates that a different function has the right of way of the x variable whose value is set as five that was added to three.

4.2. What variables from the outer scope are accessible within the closure?

```
function outerFunction2()
   local localVar = 10
   globalVar = 20
   local tableVar = { value = 30 }

   function innerFunction()
      return localVar, globalVar, tableVar.value
   end

   return innerFunction
end

local closure = outerFunction2()
print(closure())
```

Output: 10 20 30

All varieties of variables from the outer scope are accessible by a closure in Lua i.e., local variables, table elements, and global variables. This is a Lua example where innerFunction is a closure with a local variable(localVar), a global variable(globalVar) and an element inside a table (tableVar.value).

4.3. Are the variables captured from outer scope mutable or immutable?

```
function outerFunction3()
   local outerVar = 10
   function innerFunction2()
      outerVar = outerVar + 5
      return outerVar
   end
   return innerFunction2
end

local myClosure = outerFunction3()
print(myClosure())
print(myClosure())
```

Output:
15
20

Variables captured from the outside scope within a closure in Lua are mutable changing will affect outer scope. Here, outerVar is a local variable in outerFunction. outerVar is modified by an innerFunction, which is a closure. every calling to myClose adds 5 to outerVar. OuterVar is mutable and preserves its status across invocations of the closure in the outputs 15 and then 20.

4.4. Are nested closures allowed?

```
function outerFunction4(outerVar2)
   function innerFunction3(innerVar2)
      print(outerVar2, innerVar2)
      return function(mostInnerVar)
         print(outerVar2, innerVar2, mostInnerVar)
      end
   end
   return innerFunction3
end

local nested = outerFunction4("Outer")("Inner")
nested("Most Inner")
```

Output:
Outer Inner
Outer Inner Most Inner

In Lu, nested closures are supported such that an inner function can access the scope variables of outer function. This Lua code demonstrates nested closures with three levels of functions: outerFunction, innerFunction, and the most internal anonymous function. The inner function can get its own variables from its outer scope. The deepest function, which prints variables from all levels.

5. PYTHON

5.1. What is the syntax for creating closures? Is there any explicit syntax?

```
def outer_function(x):
    def inner_function(y):
        return x * y
    return inner_function

multiplier_by_3 = outer_function(3)
print(multiplier_by_3(5))
```

Output: 15

Closure in python is made when nesting a function inside another function or through lambdas. They are able to have access to variables of an enclosing function's scope through a nested function. Closure is defined through natural functionality rules and it does not necessitate a syntax specific for closure formation in Python. This code has an inner function called closure which takes up the value of the outer function's variable x. When multiplier_by_3(5), the stored x value (3) is used to multiply it by three therefore giving 15.

5.2. What variables from the outer scope are accessible within the closure?

```
def outer_function(x):
    y = 10
    global z
    z = 5

    def inner_function():
        nonlocal y
        y += 1
        print(f"x (local to outer): {x}, y (non-local): {y}, z (global): {z}")
```

```
    return inner_function

closure = outer_function(3)
closure()
```

Output: x (local to outer): 3, y (non-local): 11, z (global): 5

Closure in python can access non-local, global and enclosed function's variables. Here, inner_function is a closure that has access to x which is a local variable from outer_function, y which is a non-local variable whose value was modified using the 'nonlocal' keyword, and z which is a global variable.

5.3. Are the variables captured from outer scope mutable or immutable?

```
def outer():
    x = [10]

    def inner():
        nonlocal x
        x[0] += 1  # Modifies the list element, which is mutable
        return x[0]

    return inner

my_closure = outer()
print(my_closure())
print(my_closure())
```

Output:
11
12

Depending on how the variables captured from the outer scope are applied in Python, they could either be mutable or immutable as a closure. However, outer scope variables should remain unchanged in a closure unless specifically stated as "nonlocal". This code defines x as a list that comes up from the outside function. Inner function, as a closure, augments x's initial element. Using the nonlocal keyword, inner can change x directly. The closure may have access to variables external to it but mutability in python closures is done via special keywords such as, non-local which facilitates mutable interaction to the captured variables.

5.4. Are nested closures allowed?

```
def outer_function(x):
   def middle_function(y):
      def inner_function(z):
         return x * y * z
      return inner_function
   return middle_function

nested_closure = outer_function(2)(3)
print(nested_closure(4))
```

Output: 24

Python has the ability to create nested closures where an interior function has access to the variables within its encircling scoop. This python code portrays that inner_function is located inside the middle_function and that middle_function is located into outer_function. Every function creates a closure that includes variables of an immediate outer scope. Both x from outer_function and y from middle_function have inner_function access. Output “24” indicates that nested closures in Python are capable of handling more than one scope variable at a time.

6. RUBY

6.1. What is the syntax for creating closures? Is there any explicit syntax?

```
def block_example
  yield "Hello"
end

block_example { |msg| puts "#{msg}, Block!" }

proc_example = Proc.new { |msg| puts "#{msg}, Proc!" }
proc_example.call("Hello")

lambda_example = ->(msg) { puts "#{msg}, Lambda!" }
lambda_example.call("Hello")
```

Output:
Hello, Block!
Hello, Proc!
Hello, Lambda!

Blocks, procs, and lambas play a pivotal role in Ruby’s syntax for closure creation. These examples will construct blocks using yield, procs through the creation of Proc.new, and lambdas with ->. The most basic form of closure is “blocks” which can pass code to a method. They do a better job

at being more explicit (lambdas enforce argument count and return statements behave differently), especially procs and lambdas.

6.2. What variables from the outer scope are accessible within the closure?

```
CONST_VAR = 100

class Example
  def initialize
    @instance_var = 10
  end

  def create_closure
    local_var = 20
    lambda {
      global_var = 30  # Global variable within closure
      puts "Local variable: #{local_var}"     # Accesses local variable
      puts "Instance variable: #{@instance_var}"  # Accesses instance variable
      puts "Constant: #{CONST_VAR}"          # Accesses constant
      puts "Global variable: #{global_var}"  # Declares and accesses global variable
    }
  end
end

example = Example.new
closure = example.create_closure
closure.call
```

Output:
Local variable: 20
Instance variable: 10
Constant: 100
Global variable: 30

Any closed type of variable in the outer scope comprise parameters like local variables, constant, and instance variables. Here's how to close off any Ruby code based on a sample class. closure accesses a local variable, an instance variable @instance_var and a constant CONST_VAR. In addition, it also defines and accesses a globale variable known as global_var in its scope. when called upon on closure.call they will print these variable values.

6.3. Are the variables captured from outer scope mutable or immutable?

```
def outer_function
```

```
  x = 10
  inner_function = lambda { x += 5 }
  inner_function.call
  puts x
end

outer_function()
```

Output: 15

Mutability of outer-scope variables captured in a closure is a characteristic feature of Ruby. An outer_function in this Ruby code has a variable x inside it and a lambda (which is a closure type in Ruby). Lambda inner_function increments five units to each occurrence of x. Calling inner_function.call alters the value of x in the outer scope. This is confirmed by the output whereby x becomes equal to fifteen upon calling the lambda

6.4. Are nested closures allowed?

```
def outer_function(text)
  inner_function = lambda do
    nested_function = lambda { puts text }
    nested_function.call
  end
  inner_function
end

my_closure = outer_function("Hello, Ruby!")
my_closure.call
```

Output: Hello, Ruby!

Nested closures are allowed in Ruby meaning that a closure can be defined inside another closure. This ruby program defines an inner function that is a lambda inside another program—outer_function. This makes a new lambda called nested_function, which captures the text variable from the outside scope. The result proves that nested_function got the text variable from outside_function.

7. RUST

7.1. What is the syntax for creating closures? Is there any explicit syntax?

```
let immediate_closure = || println!("This is an immediate closure.");
immediate_closure();
```

```
let add_closure = |a: i32, b: i32| -> i32 { a + b };
println!("5 + 3 = {}", add_closure(5, 3));
```

Output:
This is an immediate closure.
5 + 3 = 8

Closure is defined by using && pattern in Rust, for different complexity to be determined by what it captures for its environment. This Rust code defines immdiate_closure as a simple closure with no parameters, which does something. The parameters and return types for add_closure.

7.2. What variables from the outer scope are accessible within the closure?

```
// Ownership
let owned_var = String::from("Hello");
let closure = move || println!("{}", owned_var);
closure();

//Borrow mutability
let mut mutable_var = 10;
let mut closure2 = || mutable_var += 1;
closure2();
println!("{}", mutable_var);

//Borrow immutability
let immutable_var = 5;
//let closure3 = || immutable_var += 1; //error
//closure3();
let closure3 = || println!("{}", immutable_var);
closure3();
```

Output:
Hello
11
5

In Rust, closures can capture variables from their surrounding scope in three ways: it should include taking ownership, borrowing mutably, and borrowing immutable. In these cases, the first one takes owned_var as its own, so it is moved into closure and not available in outer scope anymore. Second, the value of mutable_var is modified as that occurs to a mutable quantity during the second closure borrowing mutable_var. To permanently lend immutable_var in the third closure and make use of it but not change it.

7.3. Are the variables captured from outer scope mutable or immutable?

```
let mut count = 0;
let mut increment = || {
        count += 1;
        println!("Count: {}", count);
};

increment();
increment();
```

Output:
Count: 1
Count: 2

The mutability of a variable caught in a closure in ruby is determined by how it is used and declared the closure. By default closure captures variables by reference, which are immutable if not declared otherwise. Accordingly, the increment closure modifies a counter that is shared in this scope. Closure holds a mut keyword in its declaration and it references a mutable count variable which it is able to manipulate.

7.4. Are nested closures allowed?

```
let outer_closure = |x: i32| {
        let y = x * 2;
        move |z: i32| y + z
};

let inner_closure = outer_closure(3);
println!("Result: {}", inner_closure(4));
```

Output: Result: 10

Nested closures may exist in rust and an inner closure is defined within an outer closure. This allows the inner closure to operate within the environment of the outer closure. For instance, outer_closure in this case is a closure which accepts an integer value, doubles it and then again creates a closure. It closes the program internally using a second integer (z) that is added to y (the doubled value). Y moves into the inner closure through the use of moving keyword.

PART B – EVALUATION

Evaluation of the readability and writability of closures for each of the seven languages:

DART

The closure syntax in Dart is understandable and similar to Javascript's. It is simple to comprehend for anyone with experience with other programming languages since it uses well-known function grammar. The writability of the scoping rules is improved by their clarity.

GO

Go's closures are simple and work well with the language's grammar. Less verbose syntax improves writability as well as readability. Go manages closure variables in a rather unusual but practical method.

JAVASCRIPT

Closures in Javascript may be written and read with their concise syntax, especially when using arrow functions. Its widespread application in asynchronous programming demonstrates its usefulness.

LUA

The closure syntax in Lua is clean and easy. Because closures are elemental to Lua, they provide excellent readability and writability because functions are important.

PHYTON

Python has easily readable and understandable closures. Even for beginners, writing and understanding closures is made simple by the concise and clear syntax.

RUBY

Ruby relies heavily on its closures, which include lambdas, procs, and blocks. The syntax looks strange initially, but it is expressive and strong. Therefore, Ruby has average readability and writability.

RUST

Rust's closure syntax is a little more complicated than others. Although this may frustrate beginners' reading ability, it can enhance writability for experienced users because it provides strong control over how closures depict their surroundings.

In conclusion, depending on personal preferences and programming experience, different languages work better for closures. JavaScript and Python are good for ease of use due to their simplicity and friendliness. Because of its complexity, Rust works well when strong control is needed. Ruby sticks out for its expressiveness, providing a distinct and effective method. Go and

Dart are well-balanced that combine functionality and ease of use. But if one had to be selected as the best overall, JavaScript might win out because of its extensive use, adaptability, and simplicity for various programmers.

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.

LINKS TO ONLINE COMPILERS:

1. Dart: https://dartpad.dev/?
2. Go: https://go.dev/play/
3. JavaScript: https://www.programiz.com/javascript/online-compiler/
4. Lua: https://www.mycompiler.io/new/lua
5. Python: https://www.programiz.com/python-programming/online-compiler/
6. Ruby: https://www.mycompiler.io/new/ruby
7. 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 closures. This was crucial for grasping the specifics of closures.

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 closures 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 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:

SYNTAX FOR CREATING CLOSURES

I developed code examples to understand the unique syntax used in each language for defining closures. This included examining any explicit or implicit syntax required for closure creation.

VARIABLES FROM THE OUTER SCOPE THAT ARE ACCESSIBLE WITHIN THE CLOSURE

I tested which variables from an outer scope could be accessed within a closure. This involved creating scenarios where outer scope variables were referenced inside closures.

ARE VARIABLES CAPTURED FROM OUTER SCOPE MUTABLE OR IMMUTABLE?

I explored whether the variables captured from the outer scope were mutable or immutable within the closure. This was achieved by attempting to modify these variables within the closure and observing the results.

NESTED CLOSURES

I created nested closures to determine if the languages allowed closures within closures and how the variable scope is managed in such a structure.

PERSONAL COMMUNICATION

For this homework assignment, talking to others really helped me out. Whenever I got stuck, especially on the trickier parts of closures in different programming languages, I looked to the “hw2-questions” forum, which the teaching assistant (TA) formed to answer our questions. He was 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. The forum with the TA and chats with friends were super helpful. They did not 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