CSE 262: Quiz #4  
Due November 4th, 2022 at 11:59 PM

The quiz has TWO questions. Please submit your answer by updating this file in the quizzes folder of your Bitbucket account, and then committing and pushing. You should use as much space as you want for each answer. Please be detailed in your answers. Remember: this quiz is worth 9% of your grade, and you will not receive very many points if you do not give detailed answers.

**Question 1:** In Scheme, let is syntactic sugar. Explain how it is possible to get the equivalent of scoped variables without let. Give a code example with let, and then re-write it to the equivalent code without let. Then discuss whether you think it is worth it for Scheme to have added let.

Syntactic sugar is designed to make syntax easier to read or express within a programming language. In the programming language Scheme, let is syntactic sugar to get equivalent scoped variables that could have also been gotten using lambda. Lambda’s body has syntax that is similar to let, but the structure in which the variables, expressions, and bodies are paired and ordered are very different. To start, I would like to talk about the form of the lambda and the let expressions.

**The lambda expression form is: The let expression form is:**

((lambda (<var1> …<varn>) (let ((<var1> <exp1>)

<body>) (<var2> <exp2>)

<exp1> …

… (<varn> <expn))

<expn>) <body>)

As you can see, in the lambda expression, the variables are listed first, then the body, then the expression that the variables contain. On the other hand, the let expression lists the pairs of the variables and the expressions they contain right next to each other first, then the body. Evidently, the two structures are very different. Next, I would like to show an example of a let and lambda function that are identical to further show their difference in syntax. The two functions below are identical in functionality, but different in syntax because of the difference in the structures of let and lambda as discussed previously.

**Let function (Divide the sum of two numbers by 2)** **Lambda function (Divide the sum of two numbers by 2)**

(define (foo1 a b) (define (foo2 a b)

(let ((c (+ a b))) ((lambda (c)

(/ c 2))) (/ c 2)

(+ a b)

In my opinion, I believe that it was worth it for Scheme to add let because it serves well as syntactic sugar in many cases. For example, if one were to have a function with a ton of variables, it would be hard to keep track of which variable is paired with which expression. If one were to use lambda, they would have to tediously match the number in which the expression and variable was declared. This could get messy. On the other hand, let makes it easier to read by allowing one to see the variable right next to the expression it represents. This leads to less human error. Overall, I find myself using let much more often than lambda because it is easy to read.

**Question 2:** It has been said that null is a “billion dollar mistake”. Modern functional languages (and also languages like Rust) do not have null; instead they use things like Option<>. How does this work? How does it solve the problem(s) with null? Give a code example, and discuss the implications of **not** having null, both the positive and the negative.

Although the null reference is widely used because it is easy to write down, it has led to many problems within the coding industry because of its design flaw. The null reference is a bad design because it can be assigned to many different objects, while containing no value. This is a problem because the null reference lacks specificity. A lack of specificity can make it hard to find where and what the problem is in your code, let alone what something represents, and is one of the worst software principles you can break. Many people have used null to represent an empty object, but this lack of specificity may lead to a rippling effect where the null is not handled. Null can cause types to break when invoking methods of an object since null cannot access those methods.

In order to address this problem, modern functional languages, like Java, have introduced things like Optional<>, a generic class in Java. The Optional<> class is designed to be like a container that holds objects and deals with null references. To explain it, the purpose of the optional class is to explicitly tell the user that the container may or may not contain an object, and if it does not contain an object, they must deal with it. This addresses the problem with the null because it makes an empty object more specific. The coder must handle when the object is empty and return a default value that makes more sense for when the object is empty rather than just null. This improvement in specificity leads to not only better code but prevents snowballing errors because the null object is handled and can return default values that makes sense for when the object is empty. Furthermore, the optional<> class contains methods that handles an empty object very nicely.

To demonstrate the advantages of the Optional<> class, below is two snippets of Java code. One with null, to handle an empty Panda object, and another with Optional<> to handle an empty Panda object.

// Here is the panda class’s constructor and methods

public class Panda(){

int weight;

int age;

public Panda(int age, int weight)

this.age = age;

this.weight = weight;

public int getAge(){

return age;

}

public int getAge(){

return weight;

}

}

Now lets say we want to call a method that can either return a panda, or no panda.

Example with null:

Private static Panda findPanda(String name){

If(name.equals(“Bob”)){

Panda found = new Panda(10,20)

return found;

}

else

return null;

}

Example with Optional:

Private static Optional<Panda> findPanda(String name){

If(name.equals(“Bob”)){

Panda found = new Panda(10,20)

}

else

found = null;

}

// The ofNullable method returns an empty option if found is null, and a option with panda if found is a panda.  
 return Optional.ofNullable(found);

As you can see, in both cases of having an empty optional and null, they have to be handled. If the findPanda method returns null and we try to access a Panda method, it will return a NullPointer Exception, if the findPanda method returns an empty optional than it will throw a NoSuchElementException. Now the null and an empty optional may seem similar, but they are different in the way they can be handled. For the null reference, we would have to create an if else statement that checked if there was a panda. If there is a panda return the panda’s method (Ex. Get their age), otherwise return a certain value. On the other hand, the optional class has a specific method that can handle it all in one line which is using the or else method.

Null Check: Optional

Panda isBob = findPanda(“Bob”) Optional<Panda> optionalPanda = findPanda(“Bob”)

If(isBob != null){ Panda isBob = optionalPanda.orElse(new Panda (0,0))

System.out.println(isBob.getAge())

}

else return 0;

As you can see, the optional class handles the empty Panda much more nicely. The orElse method in the Optional class allows us to have a default Panda object for when there is an empty optional. On the other hand, the NullCheck is very unspecific, and we are returning a default value. The optional<> class provided by Java evidently addresses many of the problems associated with null, but there are also some problems associated with it.