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.

The general idea of let in scheme is to create a block where local variables can be bound

For example

(let ((a 2) (b 3)) (+a b))

When we enter the block, storage is allocated for local variables, and the storage is initialized with appropriate initial values. In other words, when we enter the block(scope), we bind names to storage and create associations between names and places where we put values. Inside this example, all references to the variable a and b refer to these new local variable bindings, and a,b are ‘visible’ for the rest of the let body(expressions). When we leave, these variable binding cease to exist and references to a and b will be refer to something else outside the block.

It is possible to rewrite let by lambda

( (lambda (a b) (+ a b)) 2 3)

By the following that (let ((name val)) exp) equivalent to ((lambda (name) exp) val)

In lambda case, we pass in the values as arguments of the lambda expression, and lambda expression will create a scope where these bound variables exist.

Although such transformation works, it is still worthy for scheme to have added let. The most obvious reason is that let, as a syntactic sugar, will give us a clear understanding of what we are going to use and what is it value. In let version, it expresses intent through syntax expressions, but the lambda expressions, it expresses intent through the semantics of its construction. Therefore, it is easier for us to read and understand code with let. Another reason, may be just an extension on the previous one, is that it is much easier to write code where we want bound variables to exist.

For example, if I change a little bit of above let

To (let ((a 2) (b a)) (+ a b))

As you can see, where b is bound to a and a is bound to 2. It is simple and quick to build this kind of binding with let where the binding and scope can be easily build sequentially.

However, if we want equivalent Lambda

((lambda (a)

((lambda (b)

(+ a b))

a))

2)

It will become much more complicated. Since we want b to be bound to a, we must build this bind in the scope where bound variable a exists. Therefore, we need to nest another lambda into the original one where it binds a, and this is just a simple example to demonstrate, if we have a more complicated, nested, longer let, it will be easier to make mistake.

**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.

According to the definition the Option<> is a type which represents an optional value: either Some and contains a value, or None, and does not. There are many different situations that we can apply Option, like initial values, return values for functions, optional function arguments, etc.. The general principle of using Option is when we are not sure if a value may be or may not be presented, and we want to explicitly handle the value. Null is a value conveys that it is currently invalid and unavailable, which is similar to the None variant of Option. As a language guarantee safety, rust will force us at compile time to deal with situation where we might encounter None(null) via Option<>.

For example, the following code demonstrate a simple get() process from vector

// Method to print the get value

fn value(n:Option<&char>)

{

match n

{

Some(n)=>println!("Element of vector {}",n),

None=>println!("None"),

}

}

fn main() {

let v = vec!['G','E','E','K','S'];

// here index is the non negative value which is

// smaller than the size of the vector

let index: usize = 3;

// getting value at given index value

let ch: Option<&char> = v.get(index);

value(ch);

}

As you can see, we want to get an element from the vector at a given index. Since it is possible that we would have a index > 4 where there is no element. Thus, we need to create a Option type variable either a Some or None, and we are forced to handle different situation through patterning(match) explicitly.

The positive implication is that we can make our program safer without applying null. Just like the inventor of Null, Sir Tony Hoare suggests “I call it my billion-dollar mistake. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years”(retrieved from https://medium.com/@knoldus/options-in-rust-9f28761b94b0). It is obvious that null can generate complicated error where value is unexpected absent.

The negative implication is that it increases the complexity, for the reason that we need to explicitly handle the possible outcomes every time. Additionally, in a language with strict typing like Rust, we lose a certain ‘flexibility’ to do some operations when we are dealing with possible not valid value. For instance, in java, if we want to combine 2 string(1 might be the return value from function) and one of them might be invalid. With null, we can simply determine the condition and add them. However, without null just Option, even we can still determine the condition with pattern, but we can’t do further combine operation as the one that might be valid is Option type. Therefore, it increase the complexity of writing code.