COMP 333
Summer 2024

Higher-Order Functions and Traits in Rust (Answers)

1.) Consider the following `enum` definition, representing a custom definition of `Option`:

```
enum MyOption<A> {
  Some(A),
  None
}
```

1.a.) Define a `get_or_else` method, which will take ownership over a `MyOption` instance, and will also take a function. `get_or_else` will return the contained `A` value if it's a `Some`, or will call the function to get back an `A` value in case it's a `None`. Example usage is below:

```
let v1: i32 = MyOption::Some(5).get_or_else(|| 8);
let v2: i32 = MyOption::None.get_or_else(|| 12);
println!("{}", v1);
println!("{}", v2);

---OUTPUT---
5
12
```

```
impl<A> MyOption<A> {
    fn get_or_else<F: Fn() -> A>(self, f: F) -> A {
        match self {
            MyOption::Some(a) => a,
            MyOption::None => f()
        }
    }
}
```

1.b.) Define a `map` method, which will take ownership over a `MyOption` instance, and will also take a function. If the `MyOption` instance is a `Some`, `map` will apply the given function to the value inside of the `Some`, and wrap it into a new `Some`. Otherwise, `map` will return `None` without calling the function. Example usage is below:

```
let v3: MyOption<i32> = MyOption::Some(6).map(|x| x + 1);
// v3 = MyOption::Some(7)

let v4: MyOption<usize> =
  MyOption::Some("foo").map(|s| s.len());
// v4 = MyOption::Some(3)

let v5: MyOption<bool> =
  MyOption::None.map(|b: bool| b == false);
// v5 = MyOption::None
```

```
impl<A> MyOption<A> {
    fn map<B, F: Fn(A) -> B>(self, f: F) -> MyOption<B> {
        match self {
            MyOption::Some(a) => MyOption::Some(f(a)),
            MyOption::None => MyOption::None
        }
    }
}
```

1.c.) Consider the following trait and impls:

```
trait Number {
  fn zero() -> Self;
  fn add(self, other: Self) -> Self;
}

impl Number for i32 {
  fn zero() -> i32 { 0 }
  fn add(self, other: i32) -> i32 { self + other }
}

impl Number for u64 {
  fn zero() -> u64 { 0 }
  fn add(self, other: u64) -> u64 { self + other }
}
```

Define a generic `add` method for `MyOption`, which takes some value to add. For a `Some`, `add` will add the given amount, and return a new `Some` holding the result of the addition. You will need to call `Number`'s `add` method. For a `None`, `add` will return `0`; you'll need to call the `zero` function associated with the specific kind of number. The signature has been provided for you.

```
impl<A> MyOption<A> {
    fn add(self, amount: A) -> A where A: Number {
        match self {
            MyOption::Some(value) => value.add(amount),
            MyOption::None => A::zero()
        }
    }
}
```