Student Name: Bahjat Kawar Student ID: 206989105
Student Name: Mori Levinzon Student ID: 308328467

Software Design H.W 2 – Dry Part

1. Implementing the Observable as Monad:

```kotlin
package il.ac.technion.cs.softwaredesign
class ObservableMonad<T> constructor(private val value: T) {
    private val observers = mutableListOf<(T) -> Any >()
    companion object {
        @JvmStatic
        fun <T> of(value: T): ObservableMonad<T> = ObservableMonad(value)
    }

    fun <S> flatMap(functor: (T) -> ObservableMonad<S>): ObservableMonad<S> = functor(value)

    fun addObserver(callback: (T) -> Any): ObservableMonad<T> {
        observers.add(callback)
        return this
    }
    fun removeObserver(callback: (T) -> Any): ObservableMonad<T> {
        observers.remove(callback)
        return this
    }
    fun notify(x: T): Unit {
        observers.forEach { observer -> observer(x) }
    }
    override fun equals(other: Any?): Boolean {
        if (this === other)
            return true
        if (other !is ObservableMonad<*>)
            return false
        if (value != other.value)
            return false
        return true
    }
    override fun hashCode(): Int {
        var result = value?.hashCode() ?: 0
        result = 31 * result + observers.hashCode()
        return result
    }
}
```

ObservableMonad is an object that has observers. These observers are represented by callbacks that are saved in a List, and can be added or removed from it. When observableMonad finish calculating the value that was changed by the function it notifies each of the observers.

2. Proving that the monadic laws hold for this definition:
   We implemented Test to prove the 3 laws:
   *We used this functions to prove the identities:

```
private fun f(x: Long): ObservableMonad<Long> = ObservableMonad.of( value: x * x)
private fun g(x: Long): ObservableMonad<Long> = ObservableMonad.of( value: x * x * x)
```

Left identity

The first monad law states that if we take a value, put it in a default context with return and then feed it to a function by using flatMap, it's the same as just taking the value and applying the function to it. To put it formally:

return ObservableMonad.of(x).flatMap(f) is the same thing as f(x)

```
@Test
fun leftIdentity() {
    val x = 5L
    val lhs = ObservableMonad.of<Long>(x).flatMap { x -> f(x) }
    val rhs = f(x)
    assert(lhs == rhs)
}
```

Right identity

The second law states that if we have a monadic value and we use flatMap to feed it to return, the result is our original monadic value. Formally:

m.flatMap{ ObservableMonad.of(it) } is no different than just m

```
@Test
fun rightIdentity() {
    val monadValue = ObservableMonad.of<Long>( value: 5)
    val rhs = monadValue.flatMap { value -> ObservableMonad.of<Long>(value) }
    assert(monadValue == rhs)
}
```

Associativity

The final monad law says that when we have a chain of monadic function applications with flatMap, it shouldn't matter how they're nested. Formally written:

m.flatMap(f).flatMap(g) is just like doing m.flatMap{ f(it).flatMap(g) }

```
@Test
fun associativity() {
    val monad = ObservableMonad.of<Long>( value: 5)
    val lhs = monad.flatMap { m -> f(m) }.flatMap { m -> g(m) }
    val rhs = monad.flatMap { f( x: 5).flatMap { x -> g(x) } }
    assert(lhs == rhs)
}
```