Labsheet 6

Exercise I – Higher Order Functions

1. Given List

```scala
val values = List(2, 4, 5, 6, 7, 8)
```

a. Use `partition` to separate even and odd numbers

```scala
val (evens, odds) = values.partition(_ % 2 == 0)
println(s"Evens: $evens, Odds: $odds")
```

```
Evens: List(2, 4, 6, 8), Odds: List(5, 7)
```

b. Explanation of `partition`

`partition` applies the predicate (`_ % 2 == 0`) to each element. It returns a tuple of two lists: one for elements satisfying the predicate (even numbers), and another for the rest (odd numbers).

---

c. Use `span` to separate even and odd numbers

```scala
val (prefixEvens, rest) = values.span(_ % 2 == 0)
println(s"Evens: $prefixEvens, Rest: $rest")
```

```
Evens: List(2, 4), Rest: List(5, 6, 7, 8)
```

d. Explanation of `span`

`span` splits the list into two parts: a prefix that satisfies the condition continuously, and the remaining list. It stops at the first element that does **not** match.

---

2. Given Code

```
val data = List(1, 2, 3)
val result = data.flatMap(x => List(x, x * 2))
println(result)
```

```
List(1, 2, 2, 4, 3, 6)
```

a. Output of `result`

Each element `x` is mapped to a list `[x, x * 2]`, then `flatMap` flattens the result into a single list.

b. `flatMap` vs `map`

- `map` returns a nested list of lists.
- `flatMap` flattens one level of nesting automatically, ideal for combining map + flatten.

---

3. Given List

```
val numbers = List(5, 10, 15, 20)
val filtered = numbers.filter(_ > 10)
println(filtered)
```

```
List(15, 20)
```

a. Output

`List(15, 20)` – elements greater than 10.

b. Explanation

`filter` uses a predicate function to test each element. Only those returning `true` (greater than 10) are kept.

Exercise II – Tail Recursion

1. Tail-recursive factorial

```
def factorial(n: Int): BigInt = {
  @annotation.tailrec
  def loop(acc: BigInt, n: Int): BigInt = {
    if (n <= 1) acc
    else loop(acc * n, n - 1)
  }
  loop(1, n)
}
println(factorial(5))
```

```
120
```

---

2. Tail-recursive sum of list

```
def sumList(lst: List[Int]): Int = {
  @annotation.tailrec
  def loop(acc: Int, remaining: List[Int]): Int = remaining match {
    case Nil => acc
    case head :: tail => loop(acc + head, tail)
  }
  loop(0, lst)
}
println(sumList(List(1, 2, 3, 4)))
```

```
10
```

---

3. Tail-recursive nth Fibonacci

```
def fibonacci(n: Int): Int = {
  @annotation.tailrec
  def loop(a: Int, b: Int, count: Int): Int = {
    if (count == 0) a
    else loop(b, a + b, count - 1)
  }
  loop(0, 1, n)
}
println(fibonacci(7))
```

```
13
```

---

4. Tail-recursive reverse of list

```
def reverseList[T](lst: List[T]): List[T] = {
  @annotation.tailrec
  def loop(remaining: List[T], acc: List[T]): List[T] = remaining
match {
    case Nil => acc
    case head :: tail => loop(tail, head :: acc)
  }
  loop(lst, Nil)
}
println(reverseList(List(1, 2, 3, 4)))
```

```
List(4, 3, 2, 1)
```