Anirudha Joshi – SEC01 (NUID 002991365)

Big Data System Engineering with Scala
Spring 2023
Assignment No. 2 (Lazy)

GitHub - https://github.com/anirudhajoshi2808/Anirudha_Joshi_CSYE7200

- Questions:

1. Implemented the *form* method of MyLazyList.scala
2. Find answers to the following questions:

- - (a) what is the chief way by which MyLazyList differs from LazyList (the built-in

  Scala class that does the same thing). Don't mention the methods that MyLazyList does or doesn't implement--I want to know what is the structural difference.

  (b) Why do you think there is this difference?

- - Explain what the following code actually does and why is it needed?

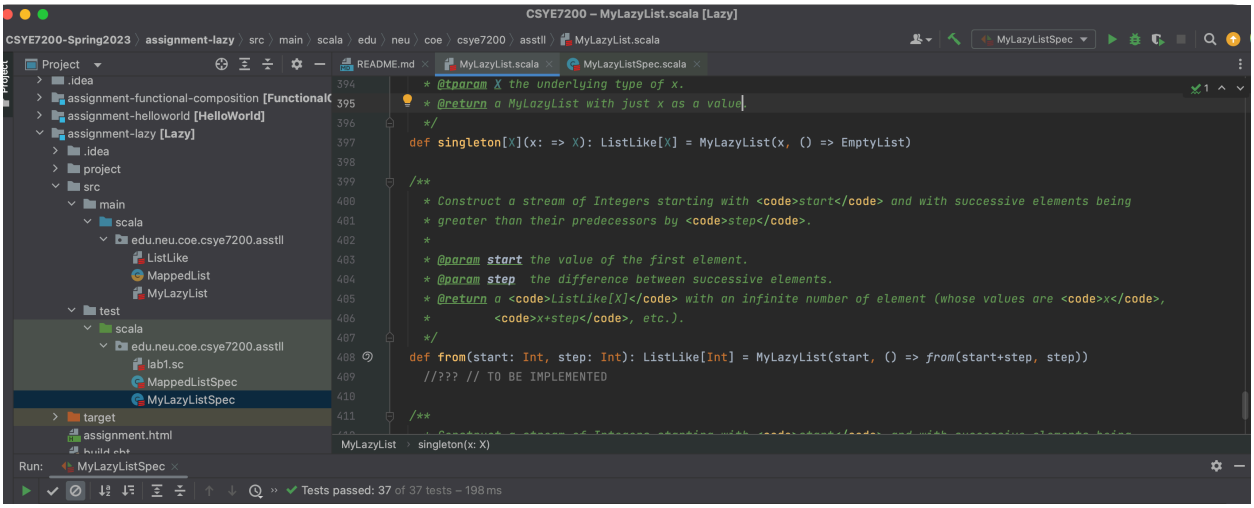
  def tail = lazyTail()

- - List all of the recursive calls that you can find in MyLazyList (give line numbers).
- - List all of the mutable variables and mutable collections that you can find in

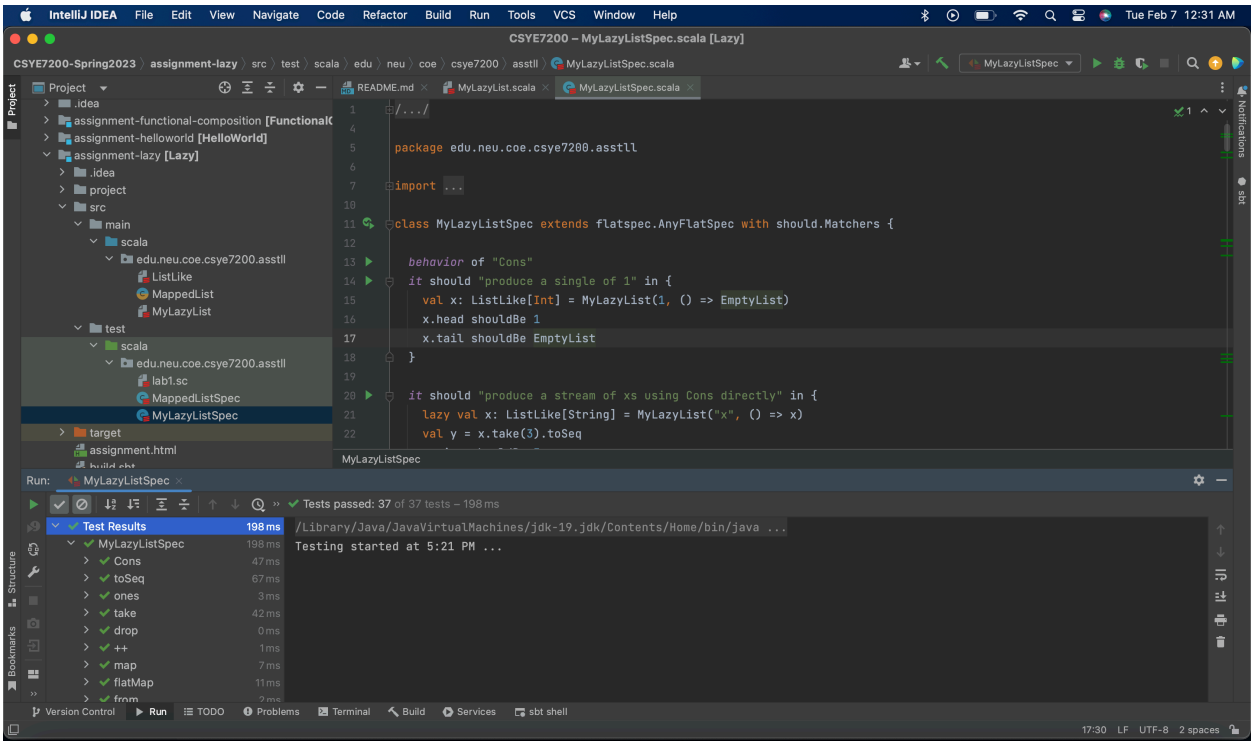
  MyLazyList (give line numbers).

- - What is the purpose of the zip method?
- - Why is there no length (or size) method for MyLazyList?

- Code



- Unit tests



- Answers:

1. (a) What is the chief way by which *MyLazyList* differs from *LazyList* (the built-in Scala class that does the same thing). Don't mention the methods that *MyLazyList* does or doesn't implement--I want to know what is the *structural* difference.

The private, implementation-defined technique for lazy evaluation of a list is provided by LazyList, whereas the actual, subclassable implementation of a lazy list is provided by MyLazyList. LazyList is also intended to be used as a base class, but MyLazyList is intended to be used as a robust implementation.

(b) Why do you think there is this difference?

While MyLazyList can be built directly by calling its apply function, LazyList can be defined using Scala's stream class, which implements LazyList as a base class.

2. Explain what the following code actually does and why is it needed? def tail = lazyTail()

This line defines a function called "tail" that produces a "ListLike[X]" object as the outcome of the "lazyTail()" method's lazy evaluation. The evaluation of the list's tail can be delayed until it is actually required by utilizing the function lazyTail in place of a concrete list object. This makes it possible to create lists that are infinite or extremely big so that they can be handled slowly, one element at a time, as opposed to having to analyze the full list at once.

3. List all of the recursive calls that you can find in *MyLazyList* (give line numbers). –

Line 26: def ++[Y >: X](ys: ListLike[Y]): ListLike[Y] = MyLazyList[Y](x, () => lazyTail() ++ ys)

Line 43: MyLazyList(y.head, () => y.tail ++ lazyTail().flatMap(f))

Line 60: def +:[Y >: X](y: Y): ListLike[Y] = MyLazyList(y, () => this)

Line 70: if (p(x)) MyLazyList(x, tailFunc) else tailFunc()

Line 82: case MyLazyList(y, g) => MyLazyList((x, y), () => lazyTail() zip g())

Line 98: case MyLazyList(h, f) => MyLazyList(h, () => f() take n - 1)

Line 116: case MyLazyList(_, f) => f().drop(n - 1)

Line 131: def inner(rs: Seq[X], xs: ListLike[X]): Seq[X] = xs match {

Line 164: case h :: t => MyLazyList(h, t)

Line 170: case MyLazyList(x, f) => Some(x -> f())

Line 298: case h :: t => MyLazyList(h, t)

Line 361: case h :: t => MyLazyList(h, () => apply(t))

Line 372: def apply[X](x: X, xs: Seq[X]): ListLike[X] = MyLazyList(x, () => apply(xs))

Line 383: def continually[X](x: => X): ListLike[X] = MyLazyList(x, () => continually(x))

Line 388: lazy val ones: ListLike[Int] = MyLazyList(1, () => ones)

Line 396: def singleton[X](x: => X): ListLike[X] = MyLazyList(x, () => EmptyList)

Line 408: def from(start: Int, step: Int): ListLike[Int] = MyLazyList(start, () => from(start+step, step))

Line 418: def from(start: Int): ListLike[Int] = from(start, 1)

4. List all of the mutable variables and mutable collections that you can find in *MyLazyList* (give line numbers).

Since no var is used or no mutable collections are imported, there are no mutable variables or collections in MyLazyList

5. What is the purpose of the *zip* method?

Instead of iterating over each list separately, the zip approach enables the processing of the components of numerous lists simultaneously and effectively. As it offers a practical and legible approach to couple corresponding elements from each collection, this is especially helpful for processing data from many collections. Additionally, because the items are only generated as needed rather than being loaded into memory all at once, the lazy evaluation of the resulting list enables efficient processing.

6. Why is there no *length* (or *size*) method for *MyLazyList*?

A lazy, endless sequence is how MyLazyList is implemented. Given that it is impossible to calculate the length of an endless sequence, a length or size approach would not be acceptable in this situation.