Programming Paradigms Tutorials
Lazy evaluation

Lazy evaluation

Lazy evaluation, or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations.

The benefits of lazy evaluation include:

- The ability to define control flow (structures) as abstractions instead of primitives.
- The ability to define potentially infinite data structures. This allows for more straightforward implementation of some algorithms.
- Performance increases by avoiding needless calculations and avoiding error conditions when evaluating compound expressions.

In functional programming, lazy evaluation means efficiency. Laziness lets us separate the description of an expression from the evaluation of that expression. This gives us a powerful ability—we may choose to describe a "larger" expression than we need, and then evaluate only a portion of it. There are many ways to achieve lazy evaluation in Scala i.e using lazy keyword, views, streams, LazyList etc.

Example:
```
scala> val num = 1
num: Int = 1

scala> lazy val lazyNum = 1
lazyNum: Int =

scala> lazyNum
res0: Int = 1
```

Streams and LazyList
A lot of examples show Streams as lazy collection, but they have lazy tail only and are now marked as deprecated. There is recommendation to use LazyList which is fully lazy instead of Stream.

The main idea:
Avoid computing the tail of a sequence until it is needed for the evaluation result (which might be never).

LazyLists are similar to lists, but their elements are evaluated only on demand. LazyList supports almost all methods of List.
For example:

```
val naturals : LazyList[Int] = 1 #:: naturals.map(_ + 1)
```

Important:
Operator :: always produces a list, never a lazy list.
There is an alternative operator #:: which produces a lazy list.

In Lazy structures the values are evaluate only once

```
naturals(5)
naturals(7)
```

For above code first line will evaluate values 1,2,3,4,5 and the second one only missing 6 and 7, as the rest of them will be taken from cache.

View
A view is a special kind of collection that represents some base collection but implements all methods lazily.
For example,

```
scala> (1 to 1000000000).filter(_ % 2 != 0).take(20).toList
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Here, I have tried to create a list of million elements and taking first 20 odd numbers. OOPS! I got OOM error. But with view,

```
scala> (1 to 1000000000).view.filter(_ % 2 != 0).take(20).toList
res2: List[Int] = List(1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21,
23, 25, 27, 29, 31, 33, 35, 37, 39)
```

This time there is no OOM error, the reason is stream has never allocated memory for all million elements. Memory is allocated only for needed elements.

Exercises

1. (3pt.) Define a function which for each number k (1…n) converts lazy list to lazy list in which eachelement of input list occurs k times
   ```
   lrepeat : [A](k: Int)(lxs: LazyList[A])LazyList[A]
   ```

2. (3pt.) Define function which generate Fibonacci Sequence in lazy way
   ```
   fib : LazyList[Int]
   ```

3. (4pt.) For lazy binary tree:

   ```
   trait lBT[+A]
   case object LEmpty extends lBT[Nothing]
   case class LNode[+A](elem:A, left:()=>lBT[A],
   right:()=>lBT[A]) extends lBT[A]
   ```

   define function which generate infinity tree which as root has number n (parameter) and sub trees: Tree (2*n) and Tree(2*n+1).