Programowanie funkcyjne i współbieżne
Lista 6

1. Jedna z pętli w języku Scala ma następującą składnię: while (*warunek*) *wyrażenie*, np.

```
var count = 0
while (count < 5) {
   println(count)
   count += 1
}
```

   Napisz funkcję whileLoop (**bez używania efektów obliczeniowych**), która pobiera dwa argumenty: warunek oraz wyrażenie i dokładnie symuluje działanie pętli while (również składniowo). Jakiego typu (i dlaczego) muszą być argumenty i wynik funkcji? Przetestuj jej działanie, symulując powyższą pętlę.

2. Napisz funkcję lrepeat [A] (k: Int) (xsl: LazyList[A]): LazyList [A], która dla danej dodatniej liczby całkowitej k i listy leniwej LazyList ($x_0, x_1, x_2, x_3, \ldots$) zwraca listę leniwą, w którym każdy element x_i jest powtórzony k razy, np.

```
(lrepeat(3) (LazyList('a','b','c','d'))).force == LazyList('a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c', 'd', 'd', 'd')
(lrepeat (3) (LazyList.from(1)) take 12).toList == List(1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4)
```

3. Polimorficzne leniwe drzewa binarne można zdefiniować następująco:

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

   a) Napisz funkcję lBreadth[A](ltree: lBT[A]): LazyList [A], która tworzy listę leniwą zawierającą wszystkie wartości węzłów leniwego drzewa binarnego.
   Wskazówka: zastosuj obejście drzewa wszerz, reprezentując kolejkę jako zwykłą listę.

   b) Napisz funkcjęlTree(n: Int): lBT[Int] , która dla zadanej liczby naturalnej n konstruuje nieskończone leniwe drzewo binarne z korzeniem o wartości n i z dwoma poddrzewami lTree (2*n) oraz lTree(2*n+1).
   To drzewo jest przydatne do testowania funkcji z poprzedniego podpunktu.