Paradygmaty programowania - ćwiczenia
Lista 5

Jak widzieliśmy na wykładzie, ewaluacja leniwa umożliwia wygodne przetwarzanie struktur potencjalnie nieskończonych. Należy jednak uważać na niebezpieczeństwo obliczeń nieskończonych, na co zwrócono uwagę przy omawianiu funkcji lfilter. Jeśli żaden element nieskończonej listy leniwej nie spełnia predykatu, to nasza funkcja nie zakończy działania, natomiast dla list skończonych zwróci listę pustą. Ewaluacja leniwa wzbogaca też repertuar programisty o zupełnie inne sposoby konstruowania programów, co widzieliśmy na przykładzie generowania nieskończonej listy leniwej, zawierającej liczby Fibonacciego.
Jako kolejny przykład użytecznych funkcji dla list leniwych (znanych z wykładu 2 dla zwykłych list) poniżej jest zdefiniowana funkcja lzip, zsuwająca dwie listy (tworząca z pary list leniwych leniwą listę par) i funkcja lunzip, rozsuwająca listy (wykonująca operację odwrotną).

Scala

```
def lzip[A,B](lxs: LazyList[A],lys: LazyList[B]): LazyList[(A,B)] =
  (lxs,lys) match
        case (h1 #:: t1, h2 #:: t2) => (h1, h2) #:: lzip(t1, t2)
        case          _             => LazyList()

//def lzip[A, B](lxs: LazyList[A], lys: LazyList[B]): LazyList[(A, B)]
```

Oto przykładowe wywołania tej funkcji i odpowiedzi środowiska interakcyjnego.

```
scala> val plxs = lzip(LazyList.from(1), LazyList.from(10))
val plxs: LazyList[(Int, Int)] = LazyList(<not computed>)
scala> plxs.take(5).toList
val res0: List[(Int, Int)] = List((1,10), (2,11), (3,12), (4,13), (5,14))
scala> (lzip (LazyList.from(1), LazyList('a','b','c'))).force
val res1: LazyList[(Int, Char)] = LazyList((1,a), (2,b), (3,c))
```

```
def lunzip[A,B](plxs: LazyList[(A,B)]): (LazyList[A] ,LazyList[B]) =
  plxs match
     case (h1, h2) #:: t => (h1 #:: lunzip(t)._1, h2 #:: lunzip(t)._2)
     case LazyList()     => (LazyList(), LazyList())

//def lunzip[A, B](plxs: LazyList[(A, B)]): (LazyList[A], LazyList[B])
```

Oto przykładowe wywołanie tej funkcji i odpowiedź środowiska interakcyjnego.

```
scala> val (lxs1, lxs2) = lunzip(plxs);  (lxs1.take(5).toList, lxs2.take(8).toList)
val lxs1: LazyList[Int] = LazyList(<not computed>)
val lxs2: LazyList[Int] = LazyList(<not computed>)
scala> (lxs1.take(5).toList, lxs2.take(8).toList)
val res2: (List[Int], List[Int]) = (List(1, 2, 3, 4, 5),List(10, 11, 12, 13, 14, 15, 16, 17))
```

Paradygmaty programowania - ćwiczenia
Lista 5

OCaml

Poniżej te same funkcje są zdefiniowane w OCamlu. Na wykładzie sami zdefiniowaliśmy listy leniwe w OCamlu, decydując się na ewaluację leniwą ogona, natomiast głowa jest ewaluowana gorliwie (co widać w odpowiedziach środowiska interakcyjnego). Poniżej została powtórzona definicja list leniwych (wykorzystująca moduł Lazy) wraz z kilkoma użytecznymi funkcjami pomocniczymi.

```
type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t;;

let rec lfrom k = LCons (k, lazy (lfrom (k+1)));;

let rec toLazyList = function
    []       -> LNil
  | x :: xs -> LCons(x, lazy (toLazyList xs))
;;

let rec ltake = function
    (0, _)                    -> []
  | (_, LNil)                 -> []
  | (n, LCons(x, lazy xs)) -> x :: ltake(n-1, xs)
;;

let rec lzip (lxs, lys) =
  match (lxs, lys) with
     (LCons(h1, lazy t1), LCons(h2, lazy t2)) -> LCons((h1, h2), lazy (lzip (t1, t2)))
  |                                _                        -> LNil
;;
(* val lzip : 'a llist * 'b llist -> ('a * 'b) llist = <fun>  *)
```

Oto przykładowe wywołania tej funkcji i odpowiedzi środowiska interakcyjnego.

```
# let plxs = lzip(lfrom 1, lfrom 10);;
val plxs : (int * int) llist = LCons ((1, 10), <lazy>)
# ltake(5, p);;
- : (int * int) list = [(1, 10); (2, 11); (3, 12); (4, 13); (5, 14)]
# ltake(100, lzip(lfrom 1, toLazyList ['a'; 'b'; 'c']));;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]

let rec lunzip plxs =
  match plxs with
  | LCons((h1, h2), lazy t) -> (LCons(h1, lazy (fst(lunzip t))), LCons(h2,lazy (snd(lunzip t))))
  | LNil                          -> (LNil, LNil)
;;
(* val lunzip : ('a * 'b) llist -> 'a llist * 'b llist = <fun> *)
```

Oto przykładowe wywołanie tej funkcji i odpowiedź środowiska interakcyjnego.

```
# let (lxs1, lxs2) = lunzip plxs;;
val lxs1 : int llist = LCons (1, <lazy>)
val lxs2 : int llist = LCons (10, <lazy>)
# (ltake(5, lxs1),ltake(8 ,lxs2));;
- : int list * int list = ([1; 2; 3; 4; 5], [10; 11; 12; 13; 14; 15; 16; 17]
```

Paradygmaty programowania - ćwiczenia
Lista 5

Wszystkie zadania mają być wykonane w obu językach: OCaml i Scala.
W OCamlu należy wykorzystać poniższą definicję list leniwych:

```
type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t;;
```

1. Zdefiniuj funkcję lrepeat, która dla danej dodatniej liczby całkowitej *k* i listy leniwej $[x_0, x_1, x_2, x_3, \ldots]$ zwraca listę leniwą, w której każdy element jest powtórzony *k* razy, np.
   lrepeat 3 $[x_0, x_1, x_2, x_3, \ldots]$ => $[x_0, x_0, x_0, x_1, x_1, x_1, x_2, x_2, x_2, x_3, x_3, x_3, \ldots]$
   Uwaga. Dla zwiększenia czytelności zastosowano tu notację dla zwykłych list.
   OCaml: lrepeat : int -> 'a llist -> 'a llist
   Scala: lrepeat[A](k: Int)(lxs: LazyList[A]): LazyList[A]

Np. lrepeat(3)(LazyList('a','b','c','d')).toList == List('a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c', 'd', 'd', 'd')
lrepeat(3)(LazyList.from(1)).take(15).toList == List(1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5)
lrepeat(3)(LazyList()).take(15).toList == List()

2. Zdefiniuj (w inny sposób, niż na wykładzie) ciąg liczb Fibonacciego.
   a) (OCaml) lfib : int llist
   b) (Scala) lfib : LazyList[Int]

3. Polimorficzne leniwe drzewa binarne można zdefiniować następująco:
OCaml: type 'a lBT = LEmpty | LNode of 'a * (unit ->'a lBT) * (unit -> 'a lBT);;
Scala:

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

a) Napisz funkcję lBreadth, tworzącą leniwą listę, zawierającą wszystkie wartości węzłów leniwego drzewa binarnego.
   OCaml: lBreadth : 'a lBT -> 'a llist
   Scala: lBreadth[A](ltree: lBT[A]): LazyList[A]
   Wskazówka: zastosuj obejście drzewa wszerz, reprezentując kolejkę jako zwykłą listę.

b) Napisz funkcję lTree , 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 podpunktu a).
   Np. lBreadth(lTree(1)).take(20).toList == List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
   lBreadth(LEmpty).take(20).toList == List()