Paradygmaty programowania - ćwiczenia
Lista 3

We wszystkich językach funkcyjnych funkcje są wartościami pierwszej kategorii, więc bez żadnych ograniczeń mogą przyjmować funkcje jako argumenty.
Poniższe dwie funkcje (w językach OCaml i Scala) aplikują zadaną funkcję jednoargumentową f do zadanego argumentu x i zwracają parę, zawierającą wynik aplikacji f x oraz czas wykonania (w milisekundach).

OCaml

```
let estimateTime f x =
  let startTime = Sys.time()
  in let fx = f x
     and estimatedTime = int_of_float((Sys.time() -. startTime) *. 1e3)
     in (fx, estimatedTime)
;;
```

Scala

```
def estimateTime[A, B](f: A => B) (x: A) = {
  val startTime = System.nanoTime
  val fx = f(x)
  val estimatedTime = (System.nanoTime - startTime) / 1000000
  (fx, estimatedTime)
}
```

Paradygmaty programowania - ćwiczenia
Lista 3

W zadaniach 2, 3, 5 funkcje należy napisać w obu językach: OCaml i Scala (wykorzystując mechanizm dopasowania do wzorca!).

1. Podaj (**i uzasadnij!**) najogólniejsze typy poniższych funkcji (samodzielnie, bez pomocy kompilatora OCamla!) :
   a) let f1 x = x 2 2;; b) let f2 x y z = x (y ^ z);;

2. Zdefiniuj funkcje a) curry3 i b) uncurry3, przeprowadzające konwersję między zwiniętymi i rozwiniętymi postaciami funkcji od trzech argumentów. Podaj (i uzasadnij!) ich typy.
   Dla każdej funkcji napisz dwie wersje: z lukrem syntaktycznym i bez lukru syntaktycznego.

3. Przekształć poniższą rekurencyjną definicję funkcji sumProd, która oblicza jednocześnie sumę i iloczyn listy liczb całkowitych na równoważną definicję nierekurencyjną z jednokrotnym użyciem funkcji bibliotecznej fold_left (Scala – foldLeft), której argumentem jest odpowiednia funkcja anonimowa (literał funkcyjny).

OCaml

```
let rec sumProd xs =
  match xs with
    h::t -> let (s,p)= sumProd t
            in (h+s,h*p)
  | []   -> (0,1);;
```

Scala

```
def sumProd(xs: List[Int]): (Int,Int) =
  xs match
    case h::t => { val (s,p)=sumProd(t)
                   (h+s,h*p)
                 }
    case Nil  => (0,1)
```

4. Poniższe dwie wersje funkcji *quicksort* działają niepoprawnie. Dlaczego?

OCaml

a)
```
let rec quicksort = function
    []  -> []
  | [x] -> [x]
  | xs  -> let small = List.filter (fun y -> y < List.hd xs ) xs
           and large = List.filter (fun y -> y >= List.hd xs ) xs
           in quicksort small @ quicksort large;;
```

b)
```
let rec quicksort' = function
    [] -> []
  | x::xs -> let small = List.filter (fun y -> y < x ) xs
             and large = List.filter (fun y -> y > x ) xs
             in quicksort' small @ (x :: quicksort' large);;
```

5. Zdefiniuj funkcje sortowania

a) przez wstawianie z zachowaniem stabilności i złożoności $O(n^2)$
   insertionsort : ('a->'a->bool) -> 'a list -> 'a list .

b) przez łączenie (scalanie) z zachowaniem stabilności i złożoności $O(n \lg n)$
   mergesort : ('a->'a->bool) -> 'a list -> 'a list .

Pierwszy argument jest funkcją, sprawdzającą porządek. Podaj przykład testu sprawdzającego stabilność.

Uwaga! Przypominam, że funkcje List.append i List.length mają złożoność liniową!