





"ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej"

## Paradygmaty programowania - ćwiczenia Lista 1

Początkowe wykłady są poświęcone paradygmatowi funkcyjnemu, ponieważ sposób myślenia jest tu zupełnie inny niż w przypadku programów imperatywnych i jeszcze go nie znacie. Zaczniemy od pisania pojedynczych funkcji w obu językach. Każde zadanie z listy 1 (z wyjątkiem ostatniego) polega na napisaniu prostej funkcji w językach OCaml i Scala.

W programach można używać wyłącznie konstrukcji językowych przedstawionych na wykładach!

To wymaganie dotyczy również kolejnych list. Każdy program można napisać na nieskończenie wiele sposobów, a naszym celem jest opanowanie nowych (omawianych na wykładach) technik i paradygmatów programowania.

W "czystych" językach funkcyjnych nie ma zmiennych mutowalnych, więc nie ma też pętli. Zamiast pętli należy stosować rekursję. Co prawda zarówno Scala, jak i OCaml zawierają mechanizmy programowania imperatywnego, ale będzie o nich mowa później. **Na razie nie wolno ich używać.** 

Programowanie ma wiele punktów wspólnych z konstruktywnymi dowodami w matematyce (tzw. izomorfizm Curry'ego-Howarda). Załóżmy, że chcemy udowodnić, że każda liczba naturalna n ma własność P(n). Zasada indukcji matematycznej mówi, że taki wniosek możemy wyciągnąć, jeśli udowodnimy dwie przesłanki:

- 1. P(0), podstawa indukcji
- 2.  $P(n) \rightarrow P(n+1)$  dla każdego n. krok indukcyjny

Lewa strona implikacji (2) nosi w tym kontekście nazwę założenia indukcyjnego.

Na str. 46 wykładu 1 jest zdefiniowana funkcja, obliczająca silnię dla każdej liczby naturalnej n. let rec silnia n = if n = 0 then 1 else n \* silnia(n-1);;

Fragment if n = 0 then 1 jest odpowiednikiem podstawy indukcji, wywołanie rekurencyjne silnia(n-1) jest odpowiednikiem założenia indukcyjnego, a wyrażenie n \* silnia(n-1) jest odpowiednikiem kroku indukcyjnego (pokazujemy, jak obliczyć n! przy założeniu, że potrafimy obliczyć (n-1)!). Struktura prostych programów dla list jest analogiczna.

let rec f xs = if xs = [] then wynik dla listy pustej (\* odpowiednik podstawy indukcji \*) else zakładając, że znamy wynik dla ogona listy f(List.tl xs), wykorzystaj wartość głowy List.hd xs do skonstruowania wyniku dla całej listy xs

W zadaniach 1-3 oraz 5-6 należy napisać funkcje polimorficzne. W językach funkcyjnych funkcje są wartościami, ale w Scali wartości nie mogą być polimorficzne. Należy je więc zdefiniować jako metody, na co zresztą wskazują ich typy (wykład 1, str. 45). W zadaniu 4 funkcja jest monomorficzna, więc można ją zdefiniować w Scali technicznie jako funkcję lub metodę (z podanymi typami). W metodach polimorficznych należy używać omawianego na wykładzie polimorfizmu parametrycznego, a nie polimorfizmu inkluzyjnego, o którym będzie mowa później (nie używać typu Any).

## Program można zgłosić jako zrobiony, jeśli spełnione są poniższe warunki:

- 1. Program jest zgodny z podaną specyfikacją.
- 2. Program się kompiluje bez błędów i ostrzeżeń.
- 3. Został przetestowany.
- 4. Student go rozumie i potrafi wyjaśnić przy "tablicy".

## Paradygmaty programowania - ćwiczenia Lista 1

Wszystkie funkcje należy napisać w obu językach: OCaml i Scala. W specyfikacjach bedzie na ogół wykorzystywana notacja OCamla, ponieważ jest prostsza.

- 1. Zdefiniuj funkcję flatten1: 'a list list -> 'a list, która dla argumentu będącego listą list tworzy listę, złożoną z elementów wszystkich podlist z zachowaniem ich kolejności, np. flatten1 [[5;6];[1;2;3]] zwraca [5; 6; 1; 2; 3], czyli spłaszcza listę o jeden poziom. Scala flatten1[A] (xss: List[List[A]]): List[A]
- 2. Zdefiniuj funkcję count : 'a \* 'a list -> int obliczającą ile razy dany element występuje w danej liście, np. count ('a', ['a'; 'l'; 'a']) zwraca 2. Scala count[A] (x: A, xs: List[A]): Int
- 3. Zdefiniuj funkcję replicate: 'a \* int -> 'a list powtarzającą dany obiekt określoną liczbę razy i zwracającą wynik w postaci listy, np. replicate ("la",3) zwraca ["la"; "la"]. Scala replicate[A] (x: A, n: lnt): List[A]
- 4. Zdefiniuj funkcję sqrList: int list -> int list podnoszącą do kwadratu wszystkie elementy danej listy liczb, np. sqrList [1;2;3;-4] zwraca [1; 4; 9; 16].

Scala sqrList(xs: List[Int]): List[Int] (metoda)
i sqrList: List[Int] => List[Int] (funkcja)

- 5. Zdefiniuj funkcję palindrome: 'a list -> bool sprawdzającą, czy dana lista jest palindromem, tj. równa się sobie samej przy odwróconej kolejności elementów, np. palindrome ['a'; 'l'; 'a'] zwraca true.

  Scala palindrome[A] (xs: List[A]): Boolean
- 6. Zdefiniuj swoją funkcję listLength: 'a list -> int, obliczającą długość dowolnej listy (oczywiście bez użycia standardowej funkcji List.length).
  Scala listLength[A](xs: List[A]): Int
- 7. Rozwiąż układ równań rekurencyjnych (zakładając, że N jest potęgą dwójki):

$$T(1) = 1$$
  

$$T(N) = c(\lg N) + T(N/2) \qquad \text{dla } N \ge 2$$

Wykorzystaj technikę zilustrowaną na wykładzie 1 (Dodatek: Złożoność obliczeniowa. Podstawowe pojęcia).