

## Lista zagadnień nr 3

### Przed zajęciami

Zakładamy znajomość tematów z pierwszych dwóch tygodni zajęć, a więc zrozumienia podstaw abstrakcji proceduralnej przedstawionych w pierwszej części podręcznika (Rozdziały 1.1–1.3). W szczególności należy rozumieć zasady składniowego **wiązania zmiennych**, rozróżniać **procesy rekurencyjne** i **iteracyjne**, a także potrafić obliczyć wartość kombinacji w której występują **procedury anonimowe** i wskazać przykłady **cukru składniowego** w programach.

### Na zajęciach

W tym tygodniu zajmować się będziemy przede wszystkim podstawami abstrakcji danych i programowaniem z użyciem list, a także testowaniem i dowodzeniem własności takich programów.

#### Ćwiczenie 1.

Wektory zaczepione na płaszczyźnie możemy reprezentować jako pary punktów: odpowiednio początek i koniec. Zdefiniuj konstruktor `make-vect`, predykat `vect?` i selektory `vect-begin` i `vect-end`, definiujące reprezentację wektora zaczepionego za pomocą punktów. Punkt możemy zdefiniować podobnie, jako parę współrzędnych kartezjańskich. Podobnie jak w przypadku wektorów, zdefiniuj konstruktor `make-point`, predykat `point?` i selektory `point-x` i `point-y`.

Następnie, dbając o zachowanie abstrakcji danych, zdefiniuj następujące procedury:

- `vect-length`, obliczającą długość wektora,
- `vect-scale`, taką że `(vect-scale v k)` znajduje wektor o początku w punkcie `(vect-begin v)`, ale który jest  $k$ -krotnie dłuższy,
- `vect-translate`, taką że `(vect-translate v p)` znajduje wektor o początku w punkcie `p` który jest przesunięciem wektora `v`.

Mogą Ci się przydać następujące procedury:

```
(define (display-point p)
  (display "(")
  (display (point-x p))
  (display ", ")
  (display (point-y p))
  (display "))")

(define (display-vect v)
  (display "[")
  (display-point (vect-begin v))
  (display ", ")
  (display-point (vect-end v))
  (display "]"))
```

### Ćwiczenie 2.

Wektory zaczepione możemy reprezentować również jako trójki składające się z punktu początkowego, kierunku (kąta z przedziału  $[0, 2\pi)$ ) i długości. Zaimplementuj taką reprezentację wektorów i zdefiniuj dla niej procedury z poprzedniego zadania.

### Ćwiczenie 3.

Zdefiniuj jednoargumentową procedurę `reverse`, która przyjmuje listę i zwraca listę elementów w odwrotnej kolejności. Podaj dwie implementacje, rekurencyjną i iteracyjną, i udowodnij ich równoważność. Jaką złożoność mają obie implementacje?

### Ćwiczenie 4.

Zdefiniuj procedurę `insert`, taką że dla posortowanej (rosnąco) listy liczb `xs` wywołanie `(insert xs n)` obliczy posortowaną listę w której `n` zostanie wstawione na właściwe miejsce. Wykorzystaj procedurę `insert` do zaimplementowania algorytmu sortowania przez wstawianie.

**Ćwiczenie 5.**

Udowodnij że `append` wraz z listą pustą `null` tworzą monoid, tj:

- Dla dowolnych `xs` i `ys`, jeśli `(list? xs)` i `(list? ys)` to `(list? (append xs ys))`;
- dla dowolnych `xs`, `ys` i `zs`, jeśli `(list? xs)`, `(list? ys)` i `(list? zs)`, to `(append (append xs ys) zs) ≡ (append xs (append ys zs))`
- dla dowolnego `xs`, jeśli `(list? xs)` to `(append xs null) ≡ xs`
- dla dowolnego `xs`, `(append null xs) ≡ xs`.

**Ćwiczenie 6.**

Permutacją ciągu  $x_1, x_2, \dots, x_n$  nazywamy dowolny ciąg  $x_{i_1}, x_{i_2}, \dots, x_{i_n}$ , gdzie  $i_1, \dots, i_n$  są parami różnymi liczbami z przedziału  $1, \dots, n$ . Permutacje powstają zatem przez przestawienie kolejności elementów na liście.

Zdefiniuj procedurę `perm` zwracającą listę wszystkich permutacji danej listy, działającą zgodnie z poniższym schematem:

- Jediną permutacją listy pustej jest lista pusta.
- Aby wygenerować permutację niepustej listy wygeneruj permutację jej ogona i wstaw jej głowę w dowolne miejsce uzyskanej permutacji.

**Ćwiczenie 7.**

Uogólnij procedurę `append` z wykładu tak żeby mogła przyjmować dowolnie dużo list jako argumenty (obliczenie `(append)` powinno dawać w wyniku listę pustą).

**Zadania domowe****Ćwiczenie 8.**

Lepszym niż wstawianie algorytmem sortowania list jest sortowanie przez scalanie. Esencją tego algorytmu jest procedura scalająca *posortowane* listy w posortowaną listę zawierającą wszystkie elementy znajdujące się na listach wejściowych. Zaimplementuj dwuargumentową procedurę `merge` scalającą dwie listy, oraz procedurę `split` dzielącą daną listę na połowy (wynikiem powinna być para list), a następnie wykorzystaj je do zaimplementowania procedury `mergesort` sortującej daną listę. Zdefiniuj również zestaw testów sprawdzający poprawność każdej z trzech funkcji, zgodnie z metodologią przedstawioną na wykładzie.

**Ćwiczenie 9.**

Niech  $xs$  i  $ys$  będą dowolnymi listami (tj.  $(list? xs)$  i  $(list? ys)$  są prawdą).  
Czemu jest wtedy równe:

- $(map f (append xs ys))$  (dla pewnej procedury jednoargumentowej  $f$ ),
- $(filter p? (append xs ys))$  (dla pewnego predykatu jednoargumentowego  $p?$ )?

Sformułuj odpowiednie twierdzenia i udowodnij je. Dowód powinien być spisany czytelnie w pliku tekstowym o nazwie `imie_nazwisko.txt`, bez specjalnego formatowania.

**Uwaga:** Rozwiązania obydwu zadań należy przesłać w systemie SKOS w nieprzekraczalnym terminie 18 marca 2019, godz. 06.00.