Lista zagadnień nr 3

Zamiast zajęć

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

Zdefiniuj jednoargumentową procedurę `select-min`, która z listy liczb wybiera element najmniejszy i zwraca parę składającą się ze znalezionego elementu i listy elementów pozostałych (z zachowaniem oryginalnej kolejności). Następnie użyj tej procedury do zaimplementowania algorytmu sortowania przez selekcję.

Ćwiczenie 6.

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)` $\equiv$ `(append xs (append ys zs))`)
- dla dowolnego `xs`, jeśli `(list? xs)` to `(append xs null)` $\equiv$ `xs`
- dla dowolnego `xs`, `(append null xs)` $\equiv$ `xs`.

Ćwiczenie 7.

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

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

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

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.

Ćwiczenie 9.

Jednym z praktycznych algorytmów sortowania list jest sortowanie szybkie. Podstawowa procedura, `partition` skomplikowana przy implementacji w miejscu w języku niskopoziomowym, staje się relatywnie prosta w języku wyższego poziomu: przyjmuje ona dwa argumenty, liczbę `n` i listę `xs`, a zwraca parę list, z których pierwsza zawiera te elementy `xs` które są mniejsze lub równe `n`, a druga — pozostałe. Aby posortować listę wystarczy teraz podzielić ją względem wybranego elementu procedurą `partition`, posortować otrzymane listy, i złączyć wyniki (zauważ że dowolny element pierwszej listy jest mniejszy niż dowolny element drugiej z nich). Zaimplementuj procedury `partition` i `quicksort`, implementujące powyższy algorytm.

Ćwiczenie 10.

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: Ponieważ to zadanie nie będzie sprawdzane automatycznie, rozwiązanie trzeba będzie wysłać w odpowiednim miejscu w systemie SKOS.